THE DESIGN OF MAPLE: A COMPACT, PORTABLE, AND POWERFUL COMPUTER ALGEBRA SYSTEM

Bruce W. Char
Keith O. Geddes
W. Morven Gentleman
Gaston H. Gonnet

Research Report CS-83-06
April, 1983

THE DESIGN OF MAPLE: A COMPACT, PORTABLE, AND POWERFUL COMPUTER ALGEBRA SYSTEM*

Bruce W. Char
Keith O. Geddes
W. Morven Gentleman
Gaston H. Gonnet

Department of Computer Science
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1

ABSTRACT

The Maple system has been under development at the University of Waterloo since December 1980. The kernel of the system is written in a BCPL-like language. A macro-processor is used to generate code for several implementation languages in the BCPL family (in particular, C). Maple provides interactive usage through an interpreter for the user-oriented, higher-level, Maple programming language.

This paper discusses Maple's current solution to several design issues. Maple attempts to provide a natural syntax and semantics for symbolic mathematical computation in a calculator mode. The syntax of the Maple programming language borrows heavily from the Algol family. Full "recursive evaluation" is uniformly applied to all expressions and to all parameters in function calls (with exceptions for only four basic system functions).

Internally, Maple supports many types of objects: integers, lists, sets, procedures, equations, and power series, among others. Each internal type has its own tagged data structure. "Dynamic vectors" are used as the fundamental memory allocation scheme. Maple maintains a unique copy of every expression and subexpression computed, employing hashing for efficient access. Another feature relying upon hashing is the "remembering" facility, which allows system and user-defined functions to store results in internal tables to be quickly accessed in later retrieval, thus avoiding expensive re-computation of functions.

The compiled kernel of the Maple system is relatively compact (about 100K bytes on a VAX under Berkeley Unix). This kernel includes the interpreter for the Maple language, basic arithmetic (including polynomial arithmetic), facilities for tables and arrays, print routines (including two-dimensional display), basic simplification, and basic functions (such as *coeff, degree, map,* and *divide*). Some functions (such as *expand, diff* (differentiation), and *taylor*) have a "core" in the kernel, and automatically load external user-language library routines for

* This work was supported in part by grants from the Natural Sciences and Engineering Research Council of Canada, and by the Academic Development Fund of the University of Waterloo.

extensions. The higher-level mathematical operations (such as *gcd, int* (integrate), and *solve*, are entirely in the user-language library and are loaded only when called.

The approach to portability of the Maple system is also discussed. Maple currently runs in C under Berkeley Vax/Unix, and B under a Honeywell GCOS operating system. Maple is currently being ported to Motorola 68000 microprocessor systems on "Unix-like" operating systems.

1. Motivation for Designing a New System

Maple is a language and system for symbolic mathematical computation, under development at the University of Waterloo since December, 1980. (The name "Maple" is not an acronym but rather it is simply a name with a Canadian identity.) The type of computation provided by Maple is known by various other names such as "algebraic manipulation" or "computer algebra". The Maple system can be used interactively as a mathematical calculator, and computational procedures can be written using the high-level Maple programming language.

With so many languages and systems already developed and being developed, the question arises: "Why develop yet another system?". We will explain our motivation for developing the Maple system and the goals we are trying to achieve with Maple.

The primary motivation can be described as *user accessibility.* This concept has several aspects. The state of the art in 1980 was such that in order to have access to a powerful system such as MACSYMA (or Vaxima)[Mos74a, Fod81a] it was necessary to have a large, relatively costly mainframe computer and then to dedicate it to a small number of simultaneous users. In the university setting, this meant it was not feasible to offer symbolic computation to large classes for student computing. In a broader context, this meant that a large community of potential users of symbolic mathematical computation remained non-users. The development of the MUMATH[Ric79a] and PICOMATH[Sto80a] systems showed that a significant symbolic computation capability could be provided on low-cost, small-address-space microcomputers. It seemed clear that it should be possible to design a symbolic system with a full range of capabilities for symbolic mathematical computation which was neither restricted by the small address space of the early microcomputers nor "inaccessible to the masses" because of unreasonable demands on computing resources. In particular, it seemed possible to design a modular system whose demands on memory would grow gracefully with the needs of the application program.

Portability was another of our earliest concerns, partly because we found ourselves users of a computing environment in transition, and partly because it was clear that a wide variety of computer systems would be coming onto the market in the decade of the 1980's. It was also recognized that "user accessibility" is greatly affected by the quality of user interface which a system provides.

Thus the primary design goals of the Maple system are: *compactness,* a *powerful set of facilities* for symbolic mathematical computation, *portability,* and

a *good user interface.* These issues are discussed in more detail in the following sections.

2. Syntax and Semantics

Part of our attempt to provide a good user interface has been to try to design a syntax which is mathematically natural. This goal is conditioned by our current assumption that most users will be accessing Maple from "ordinary" terminals using one-dimensional ASCII input. (An interesting direction for the future would be to address the design of a good user interface based upon more sophisticated peripherals, building upon previous work such as [Hof79a].) Under the current assumption, many mathematical operations are specified by the traditional function-call syntax common to many programming languages. However, Maple's syntax is enriched with mathematical constructs such as *equations,* and *ranges* (e.g. 1..3).

2.1. Sample Maple Statements

The following sample statements serve to illustrate some of Maple's syntax. (Note that the double-quote operator " is used as a "ditto" symbol to specify the latest expression.)

```
taylor( exp(3*x**2 + x), x=0, 4 );
```

$$1 + x + 7/2\,x^2 + 19/6\,x^3 + \frac{145}{24}x^4 + O(x^5)$$

```
sum( (5*i-3)*(2*i+ 9), i = 1..n );
```

$$10/3\,(n+1)^3 + 29/2\,(n+1)^2 - 269/6\,n - 107/6$$

```
expand(");
```

$$10/3\,n^3 + 49/2\,n^2 - 35/6\,n$$

```
eqn1 := 3*x + 5*y = 13;  eqn2 := 4*x - 7*y = 30;  solve( {eqn.(1..2)}, {x, y} );
```

$$eqn1 := 3\,x + 5\,y = 13$$

$$eqn2 := 4\,x - 7\,y = 30$$

$$\{y = -\frac{38}{41},\ x = \frac{241}{41}\}$$

```
limit( (tan(x)-x)/x**3, x=0 );
```

```
fibonacci := proc (n)
        option remember;
        if not type(n,integer) or n<0 then
                ERROR(`invalid argument to procedure fibonacci`)
        else
                if n<2 then n else fibonacci(n-1) + fibonacci(n-2) fi
        fi
end;

fibonacci(101);
```

573147844013817084101

2.2. Control Structures

Many of the control structures in the Maple language have been borrowed from other languages. Specifically, from Algol 68 we borrowed the repetition statement:

```
for <name> from <expr> by <expr> to <expr> while <expr>
    do <statement sequence> od
```

and the selection statement:

```
if <expr> then <statement sequence>
elif <expr> then <statement sequence>
 . . .
else <statement sequence>
fi
```

From C we borrowed the break statement for breaking out of a loop, and RETURN(expr) for returning from a procedure. The ERROR(string) construct, similar to a feature in MACSYMA, is a special function which causes an immediate return to the top level of Maple with "ERROR: string" printed out as a message. However, a procedure may be given the "errortrap" option to allow it to "catch" an ERROR condition in it or in one of the subprocedures it calls -- this is useful for error-checking in library functions, for example.

2.3. Some Semantic Features

An important semantic feature is that Maple applies *full, recursive* evaluation of expressions as the standard evaluation rule. For example, the sequence of statements

```
a := x;
x := 3;
a;
```

yields the value 3, not x. The quoting facility for preventing the evaluation of an expression is to surround the expression with single-quotes, as in 'a+ b'.

Another semantic feature in Maple is the general rule that all parameters to all functions (system-supplied or user-defined) are fully evaluated from left to right before being passed. (Again, the quoting facility can be used to explicitly prevent evaluation). We have allowed precisely four exceptions to this general rule, for four specific system functions: *assigned* (which returns true or false depending on whether the name passed as its argument is assigned or not), *evaln* (which evaluates its argument to a name), *evalb* (which evaluates its argument as a Boolean expression), and *remember* (which is a function used to place the result of a computation in an internal table for later retrieval). Another important feature of Maple is the set of powerful primitive functions that are available when writing procedures in the user-level Maple language. Some examples of such primitive functions are *degree, coeff, lcoeff* (to extract the leading coefficient), *op* (to pick operands from an expression), and *map* (to apply a procedure onto each of the operands of an expression, separately).

2.4. Types in Maple

Maple provides a *type* function for run-time type-checking. For example, if a procedure f has a parameter x then a common construct in the procedure body is a statement such as:

```
if not type(x, algebraic) then
        ERROR(`invalid argument to procedure f`) fi
```

The Maple language has been designed to avoid obligatory type declarations, a principle that we think is important if we are to have a convenient interactive system. Furthermore, we think that the syntax and semantics which applies when writing Maple procedures should be identical with the syntax and semantics of Maple's interactive mode. Consequently, no type declarations are required in Maple and writing type-independent Maple code comes naturally.

On the other hand the Maple language is not type-less. Every object has a precise type and the type information is coded in the data structure. Our concept of "objects" and "types" applies not only to the conventional objects such as integers and lists, but also to mathematical objects such as sums and products, and to objects such as procedures and tables (arrays). As an illustration of the concept of a procedure as an object, a definition of the function *abs* in Maple could take the form:

```
abs := proc (x)
        if not type(x,rational) and not type(x,real) then
                ERROR(`invalid argument to procedure abs`)
        else
                if x<0 then -x else x fi
        fi
       end;
```

This is an ordinary assignment statement, where the procedure definition (the proc...end construct) on the right-hand-side is a valid Maple expression (i.e., an *object* with its own data structure of type *procedure*). The name *abs* could later

be re-assigned any other value (of any type). It is also possible to have a procedure definition which has not been assigned to any name, as in the following expression to reverse the left and right hand sides of a list of equations:

```
map( proc (x) op(2,x)=op(1,x) end,    [ a=b, c=d, e=f ] ) .
```

3. Data Structures

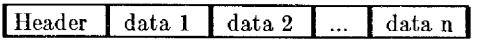
Maple has a rich set of data structures designed into it, currently about 36 different structures. Approximately one-quarter of these data structures correspond to programming language statements: assignment, if, read, etc. The remaining data structures correspond to the various types of expressions, including expressions formed using standard arithmetic and logical operators, and structures for numbers, lists, sets, tables, (unevaluated) functions, procedure definitions, equations, ranges, and series. All of these structures are represented internally as dynamic arrays (vectors), similar to the approach taken by[Nor82a].

3.1. Advantages of Dynamic Vectors

This approach using dynamic vectors at the machine level and a rich set of data structures at the abstract level has significant advantages in improved compactness and efficiency of the resulting system code. Firstly, in Maple there is only one level of abstraction above the system-level objects. It is clear that in symbolic mathematics there are many data types. The fewer and more direct the mappings between the abstract objects and the system-level objects, the simpler and more efficient will be the code that manipulates these objects. Secondly, we believe that the design of data structures should be related, if possible, to the language that describes the data objects. In our case we have a simple BNF language with the LALR(1) property, and it is natural to relate the data structures to the productions in the language. This immediately suggests the need for many data structures since there are many productions in the language. Thirdly, dynamic vectors allow us, in many cases, to have direct access to each of the components of the structure at about the same cost. This is highly desirable in some circumstances over the sequential access required when all objects are represented as lists. Fourthly, dynamic vectors are more compact than structures linked by pointers. In summary, an important part of the compactness and efficiency of Maple is due to the use of proper data structures.

3.2. Examples of Maple's Data Representation

All of the internal data structures in Maple have the same general format:

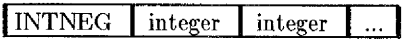


The *header* field encodes the length $(n+1)$ of the structure, the type, one bit to indicate simplification status, and two bits to indicate garbage collection status. Every data structure is created with its own length and this length will not change during its entire existence. Data structures are typically not changed after creation since it is not predictable how many other data structures are

pointing to a given structure. The normal procedure to modify structures is to create a copy and modify the copy, hence returning a new data structure.

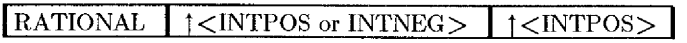
The following are some specific examples of data structures in Maple. The notation ↑<xxx> will be used to indicate a pointer to a structure of type xxx.

Negative integer



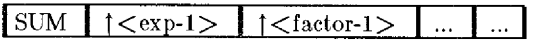
Here the INTNEG header includes both the tag for INTNEG and the length of the data structure, which depends upon the size of the negative number being represented. Each integer field of an INTNEG contains one base BASE digit. BASE=10000 for 32-bit machines and BASE=100000 for 36-bit machines; that is, BASE is the largest power of 10 that will fit into a half word on the host machine.

Rational number



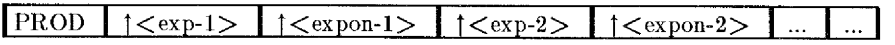
The second integer is always positive and different from 0 or 1. The two integers are relatively prime.

Sum of several terms



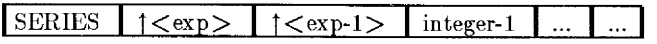
This structure should be interpreted as pairs of expressions and their constant factors. The simplifier lifts all explicit constant factors from each expression and places them in the <factor> entries. A term consisting only of a rational constant is represented with factor 1.

Product/quotient/power



This structure should be interpreted as a product of $<\text{exp-i}>^{<\text{expon-i}>}$. Rational number or integer expressions to an integer power are expanded. If there is a rational constant in the product, this constant will be moved to the first entry by the simplifier.

Series



The first expression is the "taylor" variable of the series, the variable used to do the series expansion. The remaining entries have to be interpreted as pairs of coefficient and exponent. The exponents are integers (not pointers to integers) and appear in increasing order. A coefficient O(1) (function call to the function "O" with parameter 1) is interpreted specially by Maple as an "order" term.

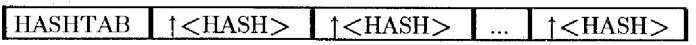
4. The Use of Hashing in Maple

Maple handles all table searching in a uniform way. All of the searching is done by an algorithm which is a slight modification of direct-chaining hashing. Although it is not obvious, the internal tables play a crucial role; they are used for: locating variable names, keeping track of simplified expressions, keeping track of partial computations, mapping expression trees into sequential files for internal input/output, and for storing arrays and tables. It is immediately obvious that the searching in these tables has to be fast enough to guarantee overall efficiency.

The algorithm used for these tables can be understood as an implementation of direct-chaining where instead of storing a linked list for each table entry, we store a variable-length array. This requires a versatile and efficient storage manager, but without one, symbolic computation would not be feasible regardless.

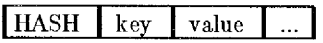
The two data structures used to implement tables are:

Table entry



Each entry points to a HASH entry or it is 0 if no entry was created. The size of HASHTAB is constant for the implementation. For best efficiency, the number of entries should be prime.

Hash-chain entry



Each entry in the table consists of a consecutive pair, the first one being the hashing key and the second the stored value. A key cannot have the value 0 as this is the indicator for the end of a chain. For efficiency reasons, the HASH entries are incremented by 5 entries at a time and consequently some entries may not be filled. Keys may be any integer or pointer which is representable in one word. In many cases the key is itself a hashing value (two step hashing).

4.1. The Simplification Table

All simplified expressions and subexpressions are stored in the simplification table. The main purpose of this table is to ensure that expressions appear internally only once. Every expression which is entered into Maple or which is internally generated is checked against this table, and if found, the new expression is discarded and the old one is used. This task is done by the simplifier which recursively simplifies (applies all the basic simplification rules) and checks against the table.

The task of checking for equivalent expressions within thousands of subexpressions would not be possible if it was not done with the aid of a "hashing" concept. Every expression is entered in the simplification table using its *signature* as a key. The signature of an expression is a hashing function itself, with

one very important attribute: it is order independent. For example, the signatures of the expressions a+ b+ c and c+ a+ b are identical; the signatures of a**b and b**a are also identical. Searching for an expression in the simplification table is done by:

- Simplifying recursively all of its components;
- Applying the basic simplification rules.
- Computing its signature and searching this signature in the table. If the signature is found then we perform a full comparison (taking into account that additions and products are commutative, etc.) to verify that it is the same expression. If the expression is found, the one in the table is used and the searched one is discarded.

The number of times that we have to do a full comparison on expressions is minimal; it is only when we have a "collision" of signatures. Some experiments have indicated that signatures coincide once every 50000 comparisons for 32-bit signatures. (Notice that the signatures are still far from uniform random numbers). The resulting expected time spent doing full comparisons is negligible. Of course, if the signatures disagree then the expressions cannot be equal at the basic level of simplification.

4.2. The Partial Computation Table

The partial computation table is responsible for handling the option remember in function definitions in its explicit and implicit forms. Basically, the table stores function calls as keys and their results as values. Since both these objects are data structures already created, the only cost (in terms of storage) to place them in the table is a pair of entries (pointers). Searching these hashing tables is extremely efficient and even for simple functions it is orders of magnitude faster than the actual computation of the function.

The change in efficiency due to the use of the remembering facility may be dramatic. For example, the Fibonacci numbers computed with

```
f := proc(n)
        if n<2 then n else f(n-1)+ f(n-2) fi end;
```

take exponential time to compute, while

```
f := proc(n) option remember;
        if n<2 then n else f(n-1)+ f(n-2) fi end;
```

requires linear time.

Besides the facility provided to users, the internal system uses the partial computation table for diff, taylor, expand, and evalr. The internal handling of expand is straightforward. There are some exceptions with the others, namely:

- diff will store not only its result but also its inverse; in other words, if you integrate the result of a differentiation the result will be "table-looked up" rather than computed. In this sense, integration "learns" from differentiation.
- taylor and evalr need to store some additional, environment, information (Degree for taylor and Digits for evalr). Consequently the entries in these cases are extended with the precision information. If a result is requested with less

precision than what is stored in the table, it is retrieved anyway and "rounded". If a result is produced with more precision than what is stored, the table entry is replaced by the new result.

- evalr only remembers function calls; it does not remember the results of arithmetic operations.

Arrays are implemented using internal tables, with the address of the (simplified) expression sequence of indices used as the hashing key. (Note that since simplified expressions appear only once, we can use their addresses as keys.) Since arrays are treated just like tables at the internal level, dense and sparse arrays are handled equally efficiently.

5. Compact Size as a Design Goal

The kernel of the Maple system (i.e., the part of the system which is written in the systems implementation language) is kept intentionally small -- for example, it occupies about 100K bytes on a VAX. The kernel system includes only the most basic facilities: the user programming language interpreter, numerical, polynomial and series arithmetic, basic simplification, facilities for handling tables and arrays, print routines, and some fundamental functions such as *coeff, degree, subs* (substitute), *map, igcd* (integer gcd computation), *lcoeff* (leading coefficient of an expression), *op, divide, imodp/imods* (integer modular operations using positive/symmetric representation), and a few others. Some of the fundamental functions have a small "core" coded in the kernel and an interface to the Maple library for extensions. The interface is general enough so that additional power, such as the ability to deal with new mathematical functions of interest to a particular user, can be obtained by user-defined Maple code. Some examples of functions which have such a "core" and a user interface are *diff, expand, taylor, type,* and *evalr* (for evaluation to a real number). Other functions supplied with the system are entirely in the Maple library, including *gcd, factor, normal* (for normalization of rational expressions), *int,* and *solve.*

The compactness of a system is affected by many different design decisions. The following points outline some of the design decisions which have contributed to the compactness of the Maple system.

1. *The use of appropriate data structures.* As we have pointed out in section 3, an important factor in compactness is the design of a rich set of data structures appropriate to the mathematical objects being manipulated, with a direct mapping between these abstract structures and the machine-level "dynamic arrays". This data structure design avoids the introduction of an intermediate "artificial" level of structure such as lists. One level of compactness is thus achieved because the number of pointers is reduced compared with a linked-list representation. Significantly, another level of compactness is achieved because the code required to manipulate these data structures is generally shorter than the code which must deal with a list representation.

2. *The use of a viable file system.* By having an efficient interpreter and by placing much of the code for system functions into the user-level library, Maple has the property that "you only pay for what you use". Writing functions in the user-level Maple language has the additional advantages of readability, maintainability, and portability. This necessarily depends upon having a file system that (at least through efficient simulation) has some desirable properties such as a tree-structured directory system and variable-length records. It may have been unreasonable a decade ago to make such assumptions about the file system, but these assumptions are (or will be) satisfied by many current and future mainframe and micro computer systems.

3. *Avoiding a large run-time support system.* Providing an "integrated programming environment" or a large run-time support system can lead to non-trivial memory requirements. For example, Franz Lisp on Berkeley Unix starts off at almost 500K bytes. We view Maple as just one of many software tools that a user may employ to solve problems, regardless of which system it may be used on. We see no need to provide all of these tools within Maple itself, not only because they greatly increase the problems of porting without providing any greater algebraic computation power, but also because many computing environments will allow their native software tools to be easily connected to Maple (say, as communicating processes) once Maple has been ported to that environment. For example, Unix EMACS[Gos81a] can invoke Maple as a subprocess on Berkeley Unix, providing some screen managing and editing facilities for Maple. Thus we do not view the basic Maple system, which provides minimal programming support (e.g., only a simple trace package and no editor), as lacking a programming environment. Rather, we see Maple as being easy to integrate into an environment chosen by the user. We certainly think that having a good user/programming interface to Maple is important. Indeed, we look forward toward developing a "personal algebra machine" in the near future. However, we envision this kind of work as building upon the basic Maple system rather than building more into it.

4. *A policy of treating main memory as a scarce resource.* We believe that this point of view is important if we are to achieve the goal of providing a symbolic computation system to "the masses". Because we have adopted such a point of view, we are constantly concerned about which functions belong in the Maple kernel and which functions can be supplied as user-level code in the Maple library. Since we have an efficient mechanism to retrieve Maple functions from the library, and an efficient interpreter, we are not forced to abandon computational power for the sake of compactness.

5. *The choice of the BCPL family of systems implementation languages.* Implementing Maple in systems languages from the BCPL family has helped us to achieve the compactness goals outlined in the above points. These languages typically produce relatively compact and efficient object code, thus contributing directly to the goal of treating main memory as a scarce resource. The support of "dynamic arrays" in the implementation language allows the

creation of compact data structures for the higher-level objects. Furthermore, an implementation language in the BCPL family typically has a run-time library that is small, selectively included, and yet provides the desired functionality.

Although the availability of inexpensive memory and hardware support for large address spaces makes it possible to design a programming system which has all of its routines contained within a large (virtual) main memory, we consider such a design to be inefficient both on mainframe timesharing systems and on the arriving generation of inexpensive but powerful microprocessor systems. It will continue to be true, in our view, that a more efficient design can be achieved by treating main memory as a scarce resource. Maple's design with a relatively small kernel interfacing to an external library takes the latter point of view.

6. Computational Power through Libraries of Functions

Another goal of the Maple system is to provide a powerful set of facilities for symbolic mathematical computation. In other words, we are not willing to achieve compactness by sacrificing the computational power of the system. Thus while the number of functions provided in the kernel system is kept to a minimum, many more functions for symbolic mathematics are provided in the system library, to be loaded as required. The functions in the system library are written in the high-level Maple programming language and are therefore readily accessible to all users of the Maple system. A load module for each library procedure is stored in "Maple internal format" which is a quick-loading expression-tree representation of the procedure definition. When a library function is invoked, its load module is read into the Maple environment (if not already loaded) and the expression tree is interpreted by the Maple interpreter.

Since run-time loading of compiled code is not (yet) a portable feature for BCPL-family languages on most systems, the execution speed of the system is seen to depend on the interpreter for the Maple language. Maple's interpreter is relatively efficient; for example, an experiment performed by running the tak function[Gri82a] shows Maple's interpreter to be about four times faster than Vaxima's interpreter on that particular benchmark. Consequently, the tradeoff between "user-level" and "system-level" code is not as great in Maple as in other systems. When a critical function has been identified as causing a serious degradation in execution time, it has been moved into the compiled kernel system*. Undoubtedly, there would be some gain in execution speed if all of the Maple functions were coded entirely in the compiled kernel but the resulting loss of compactness, and hence of user accessibility, outweighs such gains in execution speed.

* This was done, for example, with the function for polynomial division which was first placed in the system library and then later moved into the kernel. On the other hand, some functions such as solve and int have been moved from the kernel out to the system library without causing a significant degradation in performance.

7. Portability

As part of the general goal of "user accessibility", the Maple system is not tied to one operating system, nor to one programming language. Maple is intended to be portable across several languages, descendants of BCPL. To achieve this level of portability and to have a *single* source code (multiple copies are viewed as a disastrous scenario) we use a general purpose macro-processor called Margay. Our current Margay macros define a language very similar to B or C except for the places where the languages differ, where we do one of the following:

(i) Write a new macro which can be easily mapped onto every language. (Most of the time the macro will have some additional information which may be redundant for some languages but used by others). This is possible since the whole internal maple is relatively small (5500 lines) and we are willing to modify the code to improve portability.

(ii) Avoid using a particular feature if it is too peculiar to a single language.

(iii) Avoid, whenever possible, constructs that may be ambiguous across different languages.

The macro-processor is used not only as a way of providing a higher level of readability of the source code, as M6 was used with Altran [Hal71a], but also as a way to make Maple portable across several languages.

Maple is currently running under the GCOS operating system on a Honeywell 66/80 (110K words maximum address space) and under Berkeley Unix on VAX 11/780's. We have begun experiments porting to C on various operating systems on MC68000-based microcomputers, such as Xenix, Unisoft Unix, and the WICAT operating system. We have plans to port Maple into other BCPL-derivative languages in the near future, such as the locally-developed languages WSL[Bos80a] and PORT[Mal82a].

8. Notes on Software Development

Maple development started on a Honeywell system in B when the project began in 1980. When Waterloo acquired a VAX in 1981, we ported Maple to C. At that time, we were forced to demonstrate portability between languages and operating systems out of necessity, since Maple had to continue to work on the Honeywell for student use.

8.1. Choice of BCPL-derivatives as implementation language

While Maple's behaviour is based as much on our coding of algorithms and data structures as on our choice of implementation language, it seems clear to us that a general-purpose system based on a BCPL-family language can be compact, yet have reasonable performance on interesting problems. The software tools available (parser-generators, execution profilers, etc.) have made the implementation process proceed in a timely fashion with a small staff. While we don't think any final conclusions should or can yet be drawn about the relative merits of Lisp or BCPL-family languages as vehicles for symbolic systems, we do suggest that the choice of system implementation languages now seems less limited than in the

early '70s when the last generation of algebraic systems were being designed. Our approach towards portability is of course tied to the health and propagation of BCPL-family languages, but we feel this is assured, at least for the next few years, given the interest of the larger computer science community in such languages. We feel that our approach frees us to concentrate on providing algebraic computation power, as opposed to worrying about machine code generators, portable subsets, or porting programming environments.

8.2. Breadboarding

Our mode of operation up to this point is akin to "breadboarding" an electrical design, in that we can observe real, not merely theoretical, performance over a long period of time, and yet be in a position to make possibly incompatible changes in a timely fashion. The compactness of the Maple kernel is what makes breadboarding feasible for us, in that someone modifying the kernel must deal with only 5500 lines of code. As a consequence of this approach, version 1 of the Maple system is almost unrecognizable as a predecessor of version 2, although version 3 (currently under development) is characterized mainly by added facilities rather than by fundamental design changes compared with version 2.

We do not claim to have worked out all of the design and implementation issues facing us. Maple has changed since the inception of the project, not only through the introduction of additional features, but also through incompatible changes made because we changed our minds. Nevertheless, we have willingly subjected the system to significant usage at every stage of its development. The first version of the Maple system was running within a week of the first discussions on its design, with significant "real-world" problems solved using it within a month. Hundreds of students at Waterloo have already used Maple in undergraduate and graduate classes *. We are continuing to operate in a mode where there is a short time period between ideas and their implementation, with the result that the practical, real, applications of "great ideas" are soon found, and the "great ideas that are not-so-great" are modified or discarded. In this, we are grateful for the flexibility of our academic environment (and students!), and for the vigour of workers in algebraic manipulation of the past decade who have provided us with a wealth of implemented algorithms and applications problems that are obvious tests for Maple.

To some extent, the breadboarding approach means that we have had to proceed slowly on the design of "large features" such as user-directed simplification, but we think that by tying the design of Maple closely to its implementation and usage we have gained invaluable experience and feedback. Furthermore, we think that doing so has kept us from designing beyond our immediate capacity to remain faithful to maintaining efficiency and portability.

* Maple is used in an undergraduate data base class (its support of sets and tuples was used for a relational data base package), as well as courses in algebraic manipulation. It has also been used for "real" formula manipulation by some of our departmental colleagues, and as an algebraic calculator by students on a casual basis.

9. Conclusions

We expect several more cycles of building, using, and learning for Maple. Nevertheless, we believe that our accomplishments so far affirm the validity of our approach towards data representation and manipulation, towards portability, and towards making algebraic manipulation generally available. David Stoutemyer once said that one way to make computer symbolic math economically feasible for the masses would be to encourage the University of Waterloo to develop a compact "WATALG" system [Sto79a]. With the Maple system, we have taken up the spirit of that challenge.

Acknowledgments

We wish to acknowledge the assistance of: Robert Bell, Greg Fee, Brian Finch, Marta Gonnet, Barry Joe, Howard Johnson, Patrick McGeer, Michael Monagan, Mark Mutrie, Sophie Quigley, Carolyn Smith, and Stephen Watt for their various contributions to the Maple project.

References

a.

Bos80a. F. David Boswell, *A Secure Implementation of the Programming Language Pascal,* Dept. of Computer Science, University of Waterloo (1980). (M.Math thesis)

Fod81a. John Foderaro and Richard Fateman, "Characterization of VAX Macsyma," *Proceedings of the 1981 ACM Symposium on Symbolic and Algebraic Computation,* pp. 14-19 Association for Computing Machinery, (1981).

Gos81a. James Gosling, *Unix Emacs Reference Manual.* 1981.

Gri82a. Martin Griss, Eric Benson, and Gerald Maguire, Jr, "PSL: A Portable LISP System," *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming,* pp. 88-97 (1982).

Hal71a. Andrew D. Hall, Jr., "The ALTRAN System for Rational Function Manipulation - A Survey," in *Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation,* ed. S.R. Petrick, Special Interest Group on Symbolic and Algebraic Manipulation, Association for Computing Machinery (1971).

Hof79a. Carl Hoffman and Richard Zippel, "An Interactive Display Editor for MACSYMA," *Proceedings of the 1979 MACSYMA User's Conference,* p. 344 (1979).

Mal82a. Michael Malcolm, Bert Bonkowski, Gary Stafford, and Phyllis Didur, *The Waterloo Port Programming System,* Dept. of Computer Science, University of Waterloo (1982).

Mos74a. Joel Moses, "MACSYMA - The Fifth Year," in *Proceedings of the Eurosam 74 Conference,* , Stockholm (August 1974).

Nor82a. Arthur Norman, "The Development of a Vector-based Algebra System," *Proceedings EUROCAM '82,* pp. 237-248 Springer-Verlag, (1982). Lecture Notes in Computer Science #144.

Ric79a. Art Rich and David Stoutemyer, "Capabilities of the muMATH-79 Computer Algebra System for the INTEL-8080 Microprocessor," *EUROSAM 1979,* pp. 241-248 Springer-Verlag, (1979).

Sto79a. David Stoutemyer, "Computer Symbolic Math and Education: a Radical Proposal," *Proceedings of the 1979 Macsyma User's Conference,* pp. 142-158 MIT Laboratory for Computer Science, (1979).

Sto80a. David Stoutemyer, "PICOMATH-80, an Even Smaller Computer Algebra Package," *SIGSAM Bulletin* **14**(3) pp. 5-7 (1980).