UNIVERSITY OF BATH

Citation for published version:
Fitch, JP 1992, Providing REDUCE more easily. in VG Ganzha, VM Rudenko & EV Vorozhtsov (eds), Proceedings of Computer Algebra and Its Applications to Mechanics 1990 (CAAM'90). Nova Science Publishers, New York, pp. Chapter 21, 1-13, Computer Algebra and Its Applications to Mechanics, 1/01/92.

Publication date:
1992

Link to publication

Providing REDUCE More Easily

John Fitch
School of Mathematical Sciences
University of Bath
Bath BA2 7AY
United Kingdom

Abstract

REDUCE is a widely used algebraic system, largely because of its small size, and ready availability. As new versions of REDUCE become available and with it new facilities the advantage of smallness is being lost. At the same time there are new types of computers appearing, and there is a need for rapid implementation of REDUCE to deliver applications to users. This paper considers one approach to both these problems, namely writing entirely in C.

1 Introduction

For many years the algebra system REDUCE [10] has been a major tool for physics and applied mathematics [1, 4, 3]. However there are two problems which have been growing in significance. The first of these is that REDUCE is getting larger, as there is an increase in the network library; that is collections of additional modules which are of importance to some users. This has meant that it is increasingly hard for REDUCE to fit onto a personal machine. Of course technology is developing, and this has led to increased memory and speed on personal machines, but for many people in the world this has not kept pace with the perceived need.

The other problem is created by the fast-moving technology. New computing systems are reaching the market at increasing frequency. This creates a problem for software suppliers as they need to implement their system on a wide variety of systems fast. This is of course a standard problem of all software, but in the case of computer algebra the number of programmers working on the subject is small. Especially in the case of REDUCE the software effort comes mainly from academics and researchers who are principally involved in other things.

The REDUCE system is written (almost entirely) in Standard LISP [15] and so the process of porting REDUCE to a new system has been mainly the task of getting a sufficient subset of Standard LISP to work. For example when porting REDUCE to small Motorola 68000-based machines the actual REDUCE implementation was minimal [2]. The Portable Standard LISP system [8] was designed especially to make this porting easy, but unfortunately the resulting system is rather too large for many of the smaller machines which are available; for example on Intel 80386 processors it requires about 4 megabytes of main memory. Also the work involved in a port is not short.

The work described in this paper is one possible approach to these problems, and the second in particular. We consider a system for the *delivery* of REDUCE as a fixed application, and are willing to dispense with the usual system developer tools and system debugging. I will describe such a delivery system, and indicate to what extent it has been a success. A paper covering similar material in a shorter way is published elsewhere [5].

2 Requirements for an Ideal System

The main requirements for the proposed REDUCE Delivery system are that it should run in under a megabyte, and be such that it can be implemented on a new machine within a few days. In order to do this it is necessary to decide on a programming language to act as the base.

REDUCE is written in LISP, and so there is a pull towards using LISP as the basic language of the implementation. This is what PSL did, and the story has not been a great success. In terms of small and efficient system Cambridge LISP [7] and UOLISP [13] are outstanding. These systems are written in BCPL and assembler respectively. The second of these does not provide a rapid porting strategy, while the first relies on a language which is no longer in common use. However the language C [20] which is a descendent of BCPL is widely used, and is available for a large number of systems. For these reasons this was taken as the implementation language. It does also have a small hidden advantage for me, as I have access to a portable implementation of ANSI C [18]. It is assumed that the C compiler is of sufficient quality that it can compile large program segments.

3 Design

The first and most important decision which has to me made in the design of a delivery system is whether to rewrite all of REDUCE in our base language, or whether to write a LISP system. In making a choice it is important to bare in mind the large amount of code there is in REDUCE. A complete rewrite is unacceptable, and would cause significant troubles when the next version of REDUCE is released. Experiments have been made with treating the RLISP source as a specification which is translated automatically into acceptable C [16], but there are a number of difficulties which have not been solved. The design adopted in the current work is to create a compiler which compiles the REDUCE sources into a C syntax, but with no attempt that this C should be read by humans. Rather it is treated as a universal assembler.

Before describing the details of this compiler it is worth mentioning that Norman has also been working on a full Lisp system written entirely in C [17], which provides the rapid portability we require, but this system is still incomplete.

For ease of implementation a number of assumptions have been had made, not all advisably. It is assumed that the word length is 32 bits, and that 24 bits are sufficient to address all the heap space that REDUCE requires. This allows the use of an 8 bit tag, with the accompanying simplification. It may be noticed that this structure is similar to Cambridge LISP [7] which has allowed me to take and modify some code.

In order to support the assembler-style C there also needs to be some base code providing space administration, arithmetic and some functions. In the design it has been assumed that the REDUCE system is a valid LISP program, and so the type of checking which is important in a real LISP system – the numbers of arguments, taking the CAR of atoms and the like – are not necessary. We will assume that every function exists, that there is no redefinition of the REDUCE system, and we will not provide support of code development nor system debugging. This makes the needs simpler and faster to run.

3.1 Compiler

At the heart of the delivery system is the compiler. The Hearn-Griss compiler [9] which is used in PSL, Cambridge LISP and UOLISP, in differing versions, was again used. This compiler is divided into two major sections; the first translates LISP into an intermediate virtual machine code, and the second part macro-expands these into the target code. We are assuming that we have access to a modern optimising compiler, so there is no need for optimisation in our translation of the macros into C. Some functions are expanded inline for efficiency, but that is the extent of cleverness. In particular we do not have to implement register allocation schemes, as the C compiler will do that for us.

In order to make it possible to arrange garbage collection, it was decided to pass all arguments on a stack, and each function has as its only argument a pointer to the base of an argument frame. As the Hearn-Griss compiler does not preserve values in "registers" over function calls, it saves all values on the stack. Thus for example the small fragment of REDUCE

```
symbolic procedure smemq(u,v);
   %true if id U is a member of V at any level (excluding
   %quoted expressions);
   if atom v then u eq v
    else if car v eq 'quote then nil
    else smemq(u,car v) or smemq(u,cdr v);
```

will get translated by the RLISP system into the LISP

```
(de smemq (u v)
   (cond
      ((atom v) (eq u v))
      ((eq (car v) 'quote) nil)
      (t (or (smemq u (car v)) (smemq u (cdr v)))) ))
```

which in turn gets compiled to the *cmacros*

```
        (!:alloc 2)
        (!:lbl g1)
        (!:jumpnatom g2 2)
        (!:load 1 1)
        (!:jumpn g3 2)
        (!:load 1 (quote t))
        (!:jump g4)
        (!:lbl g2)
        (!:load 1 (car 2))
        (!:jumpn g5 (quote quote))
        (!:lbl g3)
        (!:load 1 (quote nil))
        (!:jump g4)
        (!:lbl g5)
        (!:load 2 (car 2))
        (!:load 1 1)
        (!:link smemq 2)
        (!:jumpt g4)
        (!:load 2 (cdr 2))
        (!:load 1 1)
        (!:store 1 1)
        (!:store 2 2)
        (!:jump g1)
        (!:lbl g4)
        (!:dealloc 2))
```

These macros are expanded one at a time to the following C

```
int _smemq(int *sp)
{
        register int d1,d2,d3,d4;
        int d5,d6,d7,d8,d9,qv,fv;
g166:
        if ((unsigned)nil < (unsigned)(*(sp+1))) goto g167;
        d1 = *(sp+0);
        if (d1 != (*(sp+1))) goto g168;
        d1 = lisptrue;
        goto g169;
g167:
        d1 = CAR(*(sp+1));
        qv = quotes[16];
        if (d1 != qv) goto g170;
g168:
        d1 = nil;
        goto g169;
g170:
        d2 = CAR(*(sp+1));
        d1 = *(sp+0);
        *(sp+0+2)=d1;
        *(sp+1+2)=d2;
        d1 = _smemq(sp+2);
        if (d1 != nil) goto g169;
        d2 = CDR(*(sp+1));
        d1 = *(sp+0);
        *(sp+0) = d1;
        *(sp+1) = d2;
        goto g166;
g169:
        return d1;
}
```

Looking at this code it is apparent that there is lot of loading and storing which hand compilation would not require. Modern compilers tidy up all this code. For example, the corresponding assembler (for an Intergraph Clipper) is given below, so it can be seen that the register allocation of the C compiler has sorted out most of the oddities in the code.

```
__smemq:
        savew11
        subq    $4,sp
        movw    r0,r14
        loada   _nil,r12
        movw    r14,r13
        addq    $8,r13
        loada   _quotes,r11
L2F4:
        loadw   4(r14),r0
        loadw   (r12),r2
        cmpw    r2,r0
        brgtu   L2F7
        loadw   (r14),r1
        cmpw    r1,r0
        brne    L2F10
        loada   _lisptrue,r0
        loadw   (r0),r0
        addq    $4,sp
        restw11
        ret     sp
L2F7:
        andi    $16777215,r0
        loadw   (r0),r0
        movw    r0,r3
        loadw   64(r11),r1
        cmpw    r1,r3
        brne    L2F14
L2F10:
        movw    r2,r0
        addq    $4,sp
        restw11
        ret     sp
L2F14:
        loadw   (r14),r1
        storw   r1,8(r14)
        storw   r0,12(r14)
        movw    r13,r0
        call    sp,__smemq
        loadw   (r12),r1
        cmpw    r1,r0
        brne    L2F20
        loadw   4(r14),r0
        andi    $16777215,r0
        loadw   4(r0),r0
        storw   r0,4(r14)
        b       L2F4
```

This mechanism of passing arguments is less efficient than direct passing. When we know that the called function cannot make any call on the space routines, and the function is in the base, then the arguments are passed in the conventional C fashion. There are still possibilities for further optimisations in this compiler, but the gains are not very large.

3.2 Base Code

While much of Standard LISP can be written in LISP [14], and with compilation this is a large section, there still remains functions which need to be hand written. Included amongst these are the input and output functions, functions for creating identifiers and gensyms, and communication with the operating system. These are not very difficult functions. The only significant part is the design of the object list. The structure used is a hash table, and as part of the limitations of a delivery system there is no provision for what happens on a table overflow.

As well as these necessary functions, it was convenient to write a number of functions in the base, either for speed, or because they were needed in the base. Including in this set are `lengthc`, `nconc`, `memq`, `reversip`, `flagp`, `get` and `equal`.

The remaining parts of the base code are for memory management and for arithmetic. These are described in separate sections.

3.3 Storage Management

There are two types of space requirement. The simplest is cons space, where each item is of equal size. More complex are vectors and strings, which are of arbitrary size. The space administration mechanism chosen is to have separate system for these two types. The cons space is allocated in hunks, and within these hunks there is a free chain. When space is exhausted a new hunk is added. The vectors and strings are also allocated from a hunk scheme, but with a first fit and amalgamation system, following that of the TRIPOS operating system [11].

Garbage collection is invoked whenever there is insufficient space from the free space. A simple mark and sweep algorithm is used, following head pointers from the argument stack and the oblist. There is no attempt at store compaction. This means that store can never be returned, but the amalgamation strategy for vectors does keep some control over fragmentation. As in usual in UNIX system, the use of `malloc` is to be avoided to allocate hunks, and instead a direct use of `sbrk` is used. For pure ANSI C use however there is a variation in the code which uses `malloc`.

As explained above the object list is a hash table. Identifiers are added to it by a linear hashing algorithm. As following a garbage collection an identifier may be removed from the table there has to be a distinction made between slots which are currently empty and those which have never been used. This is of particular significance during initialisation, as described below.

The space system has been seen to work in a satisfactory way. It has been adapted from one which is in use for the Bath implementation of EuLISP, FEEL [6].

3.4 Arithmetic

An algebra system must provide bignums as well as small integers and floating point. Implementing these operations is a significant proportion of the work involved in this project.

As the delivery REDUCE uses the same tagging structure as Cambridge LISP, the BCPL code of the latter can be translated into C to provide arbitrary precision arithmetic. This code is of a certain antiquity, as it was earlier the arithmetic for CAMAL. The algorithms are the simple ones found in Knuth [12].

This part of the system is not yet complete, but some elementary polymorphic arithmetic is available now. Small numbers are 25 bit signed integers. Bignums are represented as binary vectors with radix 10^9, and floating point numbers are simple vectors for double precision. If floating point were heavily used it would be possible to allocate them in their our hunk system.

4 Realisation

The system whose design is considered in the previous section has been under development for about two years. It exists in two distinct versions, based either on PSL on on Cambridge LISP. The examples given above have been taken from the second of these, running on an HLH Orion 1/05, as have all the dimensions given below. It has also been run on various other computing systems, including an Intel 80386 and Acorn ARM.

In order to create the system the first stage is to run the compiler described above to generate the C code. This takes about 40 minutes, and generates 115,569 lines of C, or 1.8 megabytes, as well as 1650 lines of header information. In order to make the C compiler happy this is divided automatically into eight sections. Even so the compilation is lengthy, and some C compilers have refused to accept so large programs. In addition to this code there are the fixed base code sections of a further four thousand lines, and the initialisation code. The problems of initialisation are sufficiently large that a sub section is dedicated to exploring this issue below.

The other two problems which had to be faced are the linkage between compiled code and LISP data structures, and the need for an evaluator. These are also considered below.

4.1 Linking Compiled Code to LISP

Compiled code has on occasion to refer to LISP data structures and identifiers. In our non-dynamic system this does not include names of functions to call which we are assuming are immutable, but there are still quoted lists and identifiers. The problem is to ensure that these do not get lost on garbage collection. As structures never move in the store scheme it would be possible to provide direct pointers to the structures, and have an additional list collecting them against garbage collection. This would in fact be hard to arrange within C, so the quote cell mechanism of Cambridge LISP is used. There is a static C vector which contains the pointers, and compiled code accesses them by indexed indirection.

An example of this can be seen in the example given above for the function `smemq` which needs access to the identifier `quote`.

This system does however leave an additional problem, of initialising the quotes vector.

4.2 Initialisation

There are two aspects of initialisation. As was mentioned at the end of the previous subsection it is necessary to initialise the quotes. The quotes are largely identifiers and short lists. It would be possible to lay the latter down as a separate hunk, but in the present version code is generated to call `cons` to create the structures. This generates much code, about 200Kbytes. The creation of the lists is not as simple as one might hope as most C compilers are limited in how deeply function calls can be nested. The lists have to be unwrapped to some extent.

The other part of the initialisation is all the various calls to `put` and similar in the REDUCE sources. These are collected into a large list, and then compiled at the end of the compilation process with a `progn` wrapped around it. Again the restrictions of C compilers has meant that it is advantageous to divide this into a number of sub-functions, but the amount of code involved is only 27Kbytes.

Not only is the code involved with initialisation large, almost 25% of the system, but it takes time to be obeyed. This leads to a slow start-up of the system. In order to reduce the store requirements and to make the system startup acceptable to the casual it is necessary to provide a check-point, or dump and restart facility. This involve walking over the store and converting each address into a pair of hunk number and offset, and writing the data to a file. The reloading is very similar. This system is still not finished, but there is sufficient evidence to see that this scheme is a winning idea.

4.3 Evaluator

As the system was to be a delivery one, it was initially assumed that an evaluator would not be required, as everything would be compiled. Of course this is not correct, as REDUCE calls `eval` frequently; not least of these calls is the interpretation of the user's input. Thus there is need for both `eval` and `apply`, but they do not need to be elaborate versions. There is no environment manipulation in REDUCE, and there is no need for tracing and debugging, which in Cambridge LISP form the largest part of the evaluator. The implementation of `eval` is fairly simple, needing only 80 lines of LISP which is compiled. On the other hand `apply` is complex, and the current implementation is not in strict ANSI C. As there is no checking for argument number or for the existence of the target binary program, the code is fairly short. Once control reaches compiled LISP then it cannot return to the interpreter except via `eval` or `apply`.

5 Comparisons

the intention in writing this REDUCE delivery system was to provide a small and reasonable efficient system which was easy to move to new architectures. Inevitably the outcome is not entirely as planned. In this section the actual system is looked at in comparison to alternative REDUCE systems.

In the development environment which was used there are three alternative REDUCE systems available; Cambridge LISP, PSL and KCL. This provides a suitable basis for comparison, but there may be biases due to the features of that machine, which is a 32 bit UNIX RISC machine.

Rather than build all of REDUCE a subset was chosen, consisting of the main algebraic routines, integration, factorisation, matrix operations, the solve package and the line editor. This is sufficient to run the test program except for high energy physics. Because of the incomplete nature of the infinite precision arithmetic the test program was slightly modifier to remove the need for them. For these reasons the measurements must be seen as preliminary. In particular while most of the infinite precision routines have been included, they are not debugged, and may grow.

5.1 Sizes

The version of REDUCE which is run as the service on this machine is based on Cambridge LISP. The concept of measuring the size of Cambridge LISP is a little difficult, as it works by loading modules on demand, and unloading them if deemed necessary. The base code is about 300Kbytes, with an initial data area of about the same again. To this should be added the loadable segments which total a further 1.8Mbytes including the LISP compiler; and the system runs happily in 1.5 Mbytes.

The Standard LISP system which is available on this workstation is based on CPSL, the version of PSL with stack groups and the Padget binding model[19], which is a larger that normal PSL. Nevertheless, the main code size is also one megabyte, but the runtime need is a 5 Mbyte partition minimum.

Kyoto Common LISP supports the third REDUCE, and this is considerably larger. The saved core is over 4 Mbytes, most of which is recorded as data. Running it needs about 5 Mbytes.

The system described in this paper, including the initialisation code which will be removed by the checkpoint system, is 1.2Mbytes, and it runs in 1.3Mbytes. This is with the Norcroft ANSI C compiler. With the checkpoint facility it seems as if the target of 1 Mbyte is within reach.

So, the new system is similar in size to Cambridge LISP, and a considerable improvement on the other two. More work is needed in the size aspects of the delivery system.

5.2 Speed

Again working on the Orion 1/05 workstation, and using the test program which is largely the same as the standard REDUCE test without high energy physics or big numbers, the four systems were timed. Cambridge LISP takes 21s plus 4.5s garbage collection and code loading for this test. The larger CPSL takes 24.16s with no garbage collection (as it is running in a large store). The Common LISP implementation shows the costs of the C data structures, taking 46.4s seconds. It is with pleasure that I can report that the delivery system takes only 15.2s plus 2.5s of garbage collection. Thus yet again the attempt to produce a small system has given a fast one.

The lack of runtime checking must be responsible for much of this gain, as well as the direct function calls. The passing of arguments on an auxiliary stack must explain why the gain is not as large as one might wish.

5.3 Porting to Other Machines

The C-based delivery system has been run on some other systems, including SUN3 and SUN4, HP Workstations, Acorn ARM and Intel 80386. The experience of all these ports has been similar. The C code generated is large, 134,000 lines constituting 2.2Mbytes, divided into 10 files. Some C compilers have raised objections to some of the C, and there has been considerable effort spent in generating sufficiently simple code. The size is reflected in long compilation times. Those machines with a true modern ANSI C have been easier.

For example on the Acorn Archimedes A440 the delivery REDUCE has an image of 1.1Mbytes (which is held in a compressed form on disk of under half this), and runs the same test as above in 20.3s with 4.8s garbage collection. This compares with the time of 27.7s plus 1.6s garbage collection for the normal REDUCE, which is another Cambridge LISP based. This is in line with figures above; it should however be noted that both machines are RISC machines.

The system has also run on the i386 under UNIX, but as yet not under MSDOS.

6 Discussion and Conclusions

The system has been described by looking at various detailed parts, but it is of value to look at the object as a whole. It could be thought that this is another LISP system written in C, a small KCL for example. This is not the case, and it is not intended to be so. We are interested in only one application program, and the support needs to be sufficient for it, but no more. In particular this means that the arithmetic support is well developed (at least in the design), providing efficient infinite precision arithmetic for example, but there is no debugging or tracing facilities, and no provision for users to write functions which are compiled. There is no intention to provide symbolic mode support for REDUCE, only a small interactive algebraic system, although at present there is some support for symbolic mode. The current state of development has given a system which is a little too big, but the whole project is aimed at delivery on small machines, not a development environment. There are optimistic signs that we can achieve our original goal.

This paper has presented an experimental system to deliver REDUCE, or any similar LISP application, on a computer which provides only C. It has been shown to give a very fast porting strategy for the machines tried. While there is still some work to do, particularly on initialisation and optimisation, it does give a firm basis from which to develop.

I would like to acknowledge the encouragement and assistance of Tony Hearn and Arthur Norman, as well as the computing facilities provided by Codemist Ltd. I also appreciate the suggestions of Audrey Fitch and Arthur Norman who read earlier drafts of this paper and made many useful suggestions, and to Jules Gilbert for noting the power of the code-name by which this system is known, Automath-2.

References

[1] J. P. Fitch. The application of algebraic manipulation to physics, A case of creeping flow? In W. Ng, editor, *Proceedings of EUROSAM 79*, volume 72 of *LNCS*, pages 30–41. Springer-Verlag, 1979.

[2] J. P. Fitch. Implementing REDUCE on a Microprocessor. In *Proceedings of EUROCAL 1983*, pages 128–136. Springer-Verlag LNCS 162, 1984.

[3] J. P. Fitch. Applying computer algebra. In *International Conference on Computer Algebra and its Application in Theory*, pages 262–275, 1985.

[4] J. P. Fitch. Solving algebraic problems with REDUCE. *J. of Symbolic Computation*, 1(2):211–227, June 1985.

[5] J. P. Fitch. A Delivery System for REDUCE. In *Proceedings of ISSAC90*. ACM, New York, Addison Wesley, 1990.

[6] J. P. Fitch, R. J. Bradford, and K. J. Playford. The Free and Eventually European LISP — FEEL. Technical report, The EuLISP Committee, CEC Brussels, 1989.

[7] J. P. Fitch and A. C. Norman. A High Level Implementation of LISP. *Software — Practice and Experience*, 7:713–725, 1977.

[8] M. L. Griss, E. Bensen, and G. Q. Maguire Jnr. PSL: A Portable LISP System. In *Proc. ACM Sym. Lisp and Functional Programming*, pages 88–97, 1982.

[9] Martin L. Griss and Anthony C. Hearn. Portable LISP Compiler. *Software - Practice and Experience*, 11:541–605, 1979.

[10] Anthony C. Hearn. The REDUCE Program for Computer Algebra. In *Proc. of the Third Colloquium on Advanced Computing Methods in Theoretical Physics, CNRS, Marseilles*, 1973.

[11] T. J. King. TRIPOS Technical Manual. Technical report, University of Bath, 1982.

[12] D. E. Knuth. *The Art of Computer Programming, Volume 2 – Seminumerical Algorithms*. Addison-Wesley, 1981.

[13] J. B. Marti. UOLISP. CALCODE Systems, Venice, CA, 1989.

[14] J. B. Marti and J. P. Fitch. SLISP: A Standard LISP implemented in a high level language. REDUCE Newsletter 2, University of Utah, 1978.

[15] J. B. Marti, A. C. Hearn, M. L. Griss, and C. Griss. Standard Lisp Report. *SIGPLAN Notices, ACM*, 14(10):48–68, 1979.

[16] A. C. Norman. Private communication. Description of Experiments with RLISP to C, 1988.

[17] A. C. Norman. Internal technical note of Codemist Ltd. Cambridge Common LISP, 1990.

[18] A. C. Norman and A. Mycroft. Norcroft C Compiler. Technical report, Codemist Ltd, Bath, England, 1988.

[19] J. A. Padget and J. P. Fitch. Closurize and Concentrate. In *Proceedings of POPL 85, New Orleans, ACM*, pages 255–265, 1985.

[20] Committee X3J13. The Programming Language C. ANSI Draft Standard, 1989.