SKIM - The S, K, I Reduction Machine

T.J.W. Clarke, P.J.S. Gladstone, C.D. MacLean, A.C. Norman

Trinity College, Cambridge

Abstract

SKIM is a computer built to explore pure functional programming, combinators as a machine language and the use of hardware to provide direct support for a high level language. Its design stresses simplicity and aims at providing minicomputer performance (in its particular application areas) for microcomputer costs. This paper discusses the high level reduction language that SKIM supports, the way in which this language is compiled into combinators and the hardware and microcode that then evaluate programs.

1. Introduction

In [1] Turner shows how combinators can be used as an intermediate representation for applicative programs. He compares (software) interpretation of combinator forms with more traditional schemes based on lambda calculus, and demonstrates that his new method is both elegant and efficient, at least when normal order evaluation is required. SKIM is an investigation of how Turner's ideas translate into hardware. It views his combinators as machine code, and the fixed program that obeys them as microcode. In section 2 we will present the particular applicative language we use, and comment on the need for special computers to support such languages. Section 3 reviews Turner's observations about combinators, leading in to section 4 where SKIM's hardware is described. The SKIM microcode is covered in section 5, and performance in section 6. Our conclusions, which are given at length in section 7, are that languages with applicative semantics are fun, and that very modest amounts of hardware can provide good support for them.

2. Small: an applicative language

SKIM achieves much of its simplicity by being specialised to support just one style of high level language. The language we use is called Small, and was initially designed as the user interface to an algebra system [2]. This origin has had two effects on Small. The language is applicative (i.e. no constructs in it can ever have side-effects) since this leads to a programming style which fits in very smoothly with the mathematical flavour of symbolic algebra. Also, since in an algebra system even small amounts of arithmetic may involve calling fairly expensive subroutines, the initial design for Small did not feel obliged to allow for compilation into efficient machine code. As a user-level language for driving large packages it can afford an interpretive implementation. This results in a language which demands proper treatment of functional objects (the Funarg facility, so often missing or restricted in full sized LISP systems), call-by-need (otherwise known as lazy evaluation) and an error-handling scheme compatible with the semantics of the rest of the language.

Figure 1 gives a few simple examples of Small functions and so illustrates how it compares with the direct use of lambda calculus or LISP. It is easy to demonstrate the positive features of a language such as Small, such as its pattern-matching test for decomposing structures, its capability for recursive definitions of data as well as program and its lazy evaluation. When these points have been covered there remain various real worries as to how practical Small could be for the development of large programs. Here we will ignore most of these - for instance those concerning the relationship between pure language and file stores - and just discuss the two concerns that we have considered most pressing. We pose each in the form of direct questions:

(a) Is it possible to write substantial programs without using (updatable) global variables and without the ability to alter existing data structures?

(b) Is there a time penalty associated with the use of applicative languages?

In (a) the word "possible" has to be read as implying convenience and reasonably clean programs, for after all the lambda calculus is computationally universal. Concern over time in (b) will have to consider both the effects of purity on algorithms and the practical implications of call-by-need, retention and so on.

Experience with the initial interpretive implementation of Small rapidly convinced us that any loss in expressive power that may result from removing imperative constructs from a language is more than balanced by the convenience of having normal order evaluation and higher order functions. Our consideration of gradually larger programming tasks soon however ran into problem (b) - not only was Small implemented interpretively, but its interpreter was not a particularly fast one. The speed penalty we were paying was one which has been observed before when lazy evaluation schemes have been implemented, and we have seen several suggestions for reducing its effects. In [3] Mycroft shows that many programs can be executed using an efficient call-by-value implementation, and he proposes compilation techniques that mean that the costs associated with call-by-need are only paid when they are provably necessary. This line of work views programs in pure languages as specifications of tasks to be performed and seeks to map the processes onto ones which make efficient use of existing computers. At the other extreme Mago [4] and Berkling [5] have been involved with the design of large-scale hardware that will execute applicative languages directly. The next section presents the basis of our attack on the efficiency problem.

3. Combinators and their use as a machine code

Our response to the cost of running an applicative language has been to accept that we will need an interpreter to support it, and to view the (fixed) program that is the interpreter as being microcode for the computer that our end-user sees. By doing this we hope to show that the observed performance problem is just an effect of the bias of today's computers towards efficient support of procedural language, and that it is not intrinsic to applicative ones. The idea of hardware support for side-effect-free languages immediately raises the possibility of parallel processing. This is an issue which we have not addressed: we wish to demonstrate how limited amounts of special hardware can have a large effect on performance. Given that we wish to have a machine with a substantial amount of Small-specific microcode, we have to decide what data structures this microcode will work with. It would perhaps be possible to interpret Small direct from the character form of source code, but we are prepared to use a simple compiler if doing so will speed up the interpreter.

There are two conventional ways of balancing compilation effort against microcode complexity. The first is to store programs as trees representing lambda-expressions, the second is to flatten the trees into something that resembles the object code of an ordinary machine but which provides good support for LISP-like variable access and function linkage. The scheme we adopt is closer to the first of these, but following Turner it uses combinators rather than lambda expressions as its representation of programs.

To start the description of our interpreter we will review some basic facts about combinators. Full details of the properties of combinators and the way in which they can form a model of computation can be found in any text on symbolic logic (e.g. [6]), for present purposes the following sketch will be sufficient. In its most primitive form combinatory logic is built up using just the two symbols S and K, which represent functions satisfying

```
      K x y  =  x
and   S f g x  = f x (g x)                    (A)
```

By convention functional composition associates to the left so that S f g x is ((S f) g) x. Define a new object I = S K K then observe that for any x we have I x = S K K x = K x (K x) (using the definition of S) = x. Thus I acts as the identity function. Any lambda expression can be represented in terms of the basic combinators by using the abstraction rules

```
 λ a.a     ->  I     (a is atomic),
 λ a.b     ->  K b   (b atomic and a ~= b),   (B)
 λ a.p q   ->  S (λ a.b) (λ a.q)
                (p q is any function application)
```

If a language such as Small can be parsed to give a tree equivalent to some lambda expression, the rules (B) define a codegenerator that unwinds the tree into a combinator machine code. For programs which model all their data using lambda expressions the entire order set of the machine then needed is (A).

In practice of course we will want to use an applied lambda calculus or combinator model, with lists and integers (for instance) as the data types supported. Doing this does not make any significant difference to the compilation process (B) but does mean that the machine specification (A) has to be expanded to include rules like

```
    plus m n  = (the sum of m and n)
and cons a b =(a list node with refs. to a and b).
```

How does this idea compare with the use of a lambda calculus interpreter? The main thing is that the reduction rules used are extraordinarily simple and do not involve the manipulation of anything besides the program text. In contrast lambda calculus reducers either have to keep association-list like environments or use some form of systematic renaming of user variables. The combinator language also avoids almost all temptation to make hardware registers and stacks in any way visible to the end user, whereas in lambda calculus there are opportunities to abandon retention and use shallow binding of variables with quick address modes for the first few variables in each stack frame.

Provided the compilation process is slightly extended to use a few extra combinators it also turns out that the amount of work that has to be done in performing a normal order reduction of a combinator expression is very close to that

involved in doing an applicative order reduction on an equivalent lambda expression, and is substantially less than that invovled in a 'lazy' normal order lambda reduction. Figure 2 lists the complete set of combinators we use: it can be seen that none of them are substantially more complicated than S or K.

It was the simplicity and elegance of the combinator model which made us consider a hardware implementation worthwhile.

4. The Architecture of SKIM

SKIM was designed specifically to provide hardware support for a combinator interpreter. It is a microprogrammed design with a microinstruction set that has been optimised for tree manipulation. This optimisation has succeeded in that SKIM's performance is limited by main memory bandwidth rather than by the details of the design.

We will describe the design at microinstruction level. Data is held in 16 internal registers and in 32K words of main memory, the word length being 16 bits throughout. There are only three microinstruction types: memory read, memory write and ALU. Each specifies two internal registers, Ra and Rb (two four bit fields) and performs the following data transfer:

```
Memory read    Rb <- (Ra)
Memory write   (Ra) <- Rb
ALU            Rb <- f Ra Rb
 where f is the ALU function selected.
```

This set is sufficient because of the following features:

a) All I/O is handled over a 16 bit port, addressed as an internal register (R15).

b) The microprogram sequence register is available as another internal register (R13).

c) All microinstructions include a conditional branch. Four bits in the microinstruction are decoded to give one of sixteen branch conditions and twelve bits are used to give a full 4K branch address.

The branch address, suitably extended to 16 bits, can be addressed as a read-only internal register (R14), and provides for insertion of immediate data when the branch facility is not used.

Rather than list the ALU functions we remark that the chips we use are 74S281's, and that bits in the instruction control the shifting of the data and the carry bit. Support is provided for multiple length operations.

So far the design has seemed quite general purpose. The hardware optimisation involves the organisation of main memory. This is divided into two banks (Head and Tail), so that a memory location is given by one bit in the instruction, selecting the bank, together with a 14 bit address. This addressing mode is the only one needed, since all programs, data, stacks etc. are held as trees. The top two bits of a data word give information about the type of pointer and branch conditions are provided to test these. Associated with each word in memory is a flag bit, used by the garbage collector, and memory write instructions may set or reset this bit. During a memory read a conditional branch may be made on the value of this flag bit.

This microinstruction set determines the hardware design, shown schematically in figure 3. All microinstructions are the same length, 600 ns, which is determined by the cycle time of the main memory (4116 16K dynamic RAM). Conveniently, this means that slow MOS EPROMs can be used to store the microcode, provided that one level of pipelining is used. The long cycle time allows all data transfers to be multiplexed through one main bus, there being at most three such transfers per cycle, greatly simplifying the design. ALU instructions, which represent wasted memory cycles, provide an economical way of refreshing the RAM.

The completed design, including 64K bytes of RAM and 4K words of EPROM, comes to just 100 packages, which fit comfortably on two double sized Eurocard boards. This is no larger than many microprocessor systems with the same amount of memory, vindicating our decision to use msi TTL rather than bit-slice technology.

5. SKIM microcode

The SKIM microcode falls into four main sections which are, in decreasing order of size, the combinator evaluator, the expression reader, the printer and the storage manager/garbage collector. The first and last of these are interesting enough to warrant special attention. We will cover the storage manager first.

Apart from a very few words used by various routines to contain useful constants and a save area for the garbage collector, all of SKIM's memory is organised as pairs and is subject to the control of the garbage collector. To avoid having to set aside dedicated memory and because SKIM is not very good at running real stacks this has a non-recursive mark phase, the code for which is displayed in figure 4. Without a full key to the mnemonics used in SKIM's assembler the details of this code will remain obscure. It is however possible to see from it how a large proportion of the instructions are able to perform tests and conditional branches in parallel with data movement. Timings of this code will be quoted in the next section. Close inspection of the code

will reveal more details of the hardware. For instance the cycle after a conditional branch has to be inhibited while a pipeline register is drained; after unconditional branches this is not done. The effect of this is that unconditional branches have to be written one line "early" in the source code, but to compensate for this they do not slow the machine down at all.

The core of the combinator reducer is a piece of code which searches down the leftmost branch of a tree until it finds an operator as a leaf, and then it goes off to decode and execute the operator. To make the final part of this process fast we make the internal representation of combinators just the absolute address in microcode store of the program that does the required reduction. This also provides an easy way to gain non-standard entry to diagnostic routines hidden in the microcode. The reducer, like the garbage collector, uses pointer reversal to keep track of its tree-walk. Figure 5 shows how pointers get reversed in the process of finding and processing an S combinator. Note how when S needs to find its arguments the back-pointer chain gives convenient and cheap access to them. If, as the result of a programmer error, the pointer reversing reducer were given a cyclic structure representing a self-dependent computation then it would eventually reach a portion of the structure which it had already traversed. In this situation it traces the wrong way up the tree, finding its back-pointer terminator as a combinator. This is easy to detect and provides a neat check for this form of illegal program. When working on the operands of an arithmetic operator such as + the reducer does need a real stack. We use a linked list, which with SKIM's hardware is almost as efficient as a real stack on a more normal machine. It also has the advantage that the stack does not have to occupy large contiguous areas of free store, but can use any free pairs.

6. Performance

In evaluating a design such as SKIM it is necessary to measure both the low level efficiency of the microcode and the effect that the programming model used has on gross performance. The results to be presented here are still fairly rough, since SKIM has only been running for a month or so. We can however report results of paper studies, simulations of the SKIM hardware, other software combinator-reducing programs and programs run directly on SKIM, and hope that a good composite picture of the potential of combinators will emerge.

The first piece of code to be considered is the garbage collector of Figure 4. We have used the pointer reversing tree-walk algorithm that it uses as a benchmark to compare SKIM against a number of other processors. The programs used in this comparison were written in assembly code by a number of different people, and on some processors (notably the Z80) it was clear that individual coding skill could make a large difference to the times reported. We nevertheless find the final results, table 1, consistent enough to be used as a rough measure of machine speed. For the simpler machines (i.e. those without elaborate instruction look-ahead and cache stores) the times quoted were computed from published instruction timings. Where possible empirical verification was then obtained. For the 370, times depend critically on the exact sequencing of orders, and so the result given is based solely on observation.

Two things show up in these timings. The first is that the scattered memory references and dense control structure of a tree-walk program can be bad news for a mainframe such as the IBM 370 - its performance is limited by the speed of its main memory which is not very much faster than that of a mini. The second is that SKIM can get a great deal of list processing done per cycle. We attribute most of this to the fact that it is a fixed program machine, and so instruction fetches are done in parallel with store manipulation.

We now consider the SKIM code in more detail to get some idea of its absolute performance. A first way of measuring this is to consider its use of memory cycles, counting each ALU cycle in the processor as a wasted opportunity to do something with main store. Here we find that about 60% of all cycles are memory accesses. Of those that are left about 10% are needed for refreshing the dynamic RAMs. Thus if we stay with an architecture where only one word is touched at a time, elaborations to the SKIM processor can at the very best less than double the speed of its garbage collector. Since the main combinator reducing algorithm has a similar structure to that of the list marker, and since the most important combinators involve pointer adjustments, we expect a similar result to hold for the entire SKIM system.

Now how does SKIM perform when programmed in its combinator language? The simple answer is: like a system that runs interpretively on a fairly fast processor. We consider it fair to compare SKIM against interpretive LISP on the local IBM mainframe and against a fast integer BASIC running on a 6502 micro. The first test we ran involved the generation of lists of prime numbers. The algorithm used in all cases involved test division by all primes up to the square root of the next number to be tested for primality. On SKIM this can be done very neatly using a recursively defined list structure. By running SKIM for a few hours we have made it tabulate the primes up to a million or so. For present purposes we quote figures for computations lasting about a minute. In 1 minute a program running in interpreted 32-bit integer BASIC on 1 2MHz 6502 [7] was able to find all primes up to 4621 (there are slightly over 600 of them). For the same calculation SKIM takes 35 seconds. A compiled version of the same BASIC program produced between 1100 and 1200 primes in a minute - to get to the same state SKIM takes 80 seconds. We note that the 2MHz version of

the 6502 used is still one of the faster 8-bit micros available. For comparison with a mainframe we coded an arbitrary precision integer arithmetic package in both Small and LISP, in each case avoiding machine dependance by representing numbers base 1000. Interpreted LISP [8] on the 370/165 took 12 seconds to compute 2**1000, while SKIM took 25 seconds. Compiled LISP code obtained the same result in about 3 seconds. A separate test, involving the computation and display of the first 12 Legendre polynomials again showed a tenfold speed advantage to the 370 when running compiled code. Given the fairly high level of SKIM's combinator machine code and the fact that our test programs were all produced by compilation from Small (i.e. without any hand-optimisation) we find these results most encouraging.

7. Conclusions

When the SKIM project started we felt that the clean logical structure of combinators ought to lead to a simple yet fast hardware design. As the hardware and microcode has crystallised, and as we have gained experience with combinators and applicative programming these initial hopes have been largely fulfilled. SKIM demonstrates yet again that special purpose hardware can be a great help when a slightly unconventional language is being implemented, and that at least for list-processing applications even a simple computer can deliver a quite respectable performance. There have been a few ways in which SKIM has surprised us. The amount of microcode needed to implement a combinator reduction scheme is much less than we had feared, and although we allowed for 4K words we at present need only 2K. It has also become clear that our machine, even though it was intended just for the support of reduction languages can be viewed as a medium speed minicomputer where all programs happen to reside in PROM. This is encouraging us to experiment with it, not just by modifying our combinator reducer so that it works on lambda-expressions and so LISP programs, but by considering SKIM as the basis for real-time music generation and the like.

Suppose we were starting the project from fresh, what would we do in the light of the experience we have gained so far? Almost certainly we would succumb to the temptation to make the processor slightly more elaborate but slightly faster. The main test programs we have used so far have all been badly limited by the rather slow multiplication and division that SKIM supports. A modest add-on extended arithmetic unit might speed up our primes and long arithmetic code substantially. However, beyond that rather straightforward change it seems clear that major speed enhancements can only come out of better use of memory. We suggest that the main ways of doing this would be:

Change the microcode model so that all store accesses use three registers, rather than two, with a typical operation having the form

R1 -> (R2) -> R3,

i.e. using R2 as a memory address, reading from that word into R3 and writing R1 as the new memory contents. This combines a register transfer with a read-modify-write memory cycle and would drastically reduce our need for non-memory cpu cycles as well as reducing the total number of memory cycles required.

Allow instructions to access both left and right pointers in a cell at once, using a 32 bit (rather than 16 bit) memory data bus.

Either Provide a separate special memory for use as a stack, or Slave the top few links of backpointer chains in a special FIFO register that could provide rapid access to the arguments of combinators.

More elaborate timing generation and microcode to keep all cycles as short as possible.

It seems plausible that a combination of these techniques would improve performance by a factor of about 4 (at the cost of perhaps doubling the size of the processor), and would move us into a class of machine where we would want to increase our wordlength and provide much more than 64k bytes of memory. The other direction in which we could move would be towards a VLSI implemention of a combinator reducing processor. Our estimates for the number of transistors in SKIM as we have it at present suggest that (providing we do not suddenly find a need for a lot more microcode) it would fit fairly tidily onto a single chip.

Perhaps the real conclusion of this paper is that one cycle of hardware design is just coming to completion for us, but that we still have a lot of software experience to gather, and that there is plenty of scope for that to lead us to further processor designs. The support that SKIM provides will now make it possible for us to treat applicative programming as a viable option rather than just as an interesting but impractical idea.

8. Acknowledgements

The SKIM project was financed by the Cambridge University Processor Group and by Research Machines Ltd of Oxford. Additional assistance came from Acorn Computers/CPU of Cambridge. Many people around Cambridge provided encouragement and pressure to make the processor work: of these we note particularly Ian Kitching who wrote the final version of the SKIM emulator and Bill Worzel who was much involved in the decision to go ahead and actually build something.

Figure 1.

(a) reverse a list

```
LISP:  DEFINE ((
       (REVERSE (LAMBDA (L) (REV1 L NIL)))
       (REV1 (LAMBDA (A B) (COND
          ((NULL A) B)
          (T (REV1 (CDR A) (CONS (CAR A) B))))))
       ))

Small: Let REVERSE L = REV1 L Nil
       [ REV1 A B =
         If A Is NEXT . A' Then REV1 A' (NEXT . B)
         Else B Fi ]
```

Combinators:

```
          = REVERSE(C(Y(B C(B(S' U' K)(C(
              B'(B' C))(C P)))))NIL).
```

Notes: Square brackets in Small introduce qualifying clauses or local definitions. The If...Is...Then construction matches the structure of A against a template and provides names for the components so found. The symbol 'Fi' is used to terminate conditionals, following the style of Algol68. The combinators can only be understood by experts.

(b) a fixed point operator

Lambda-calculus:

$$Y = \lambda f.\ ((\lambda h.\ h\ h)\ (\lambda g.\ f(g\ g)))$$

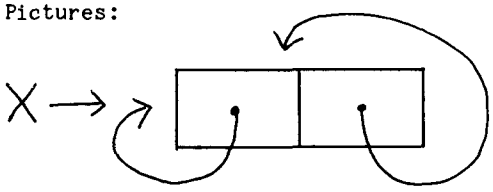
```
Small: Let Y F = H H [ H G = F (G G) ]
   or more directly: Let Y F = F (Y F)
```

Combinators (using S, K and I only):

```
    = Y S(K(S I I))(S(S(K S)K)(K(S I I))).
```

(c) a re-entrant list

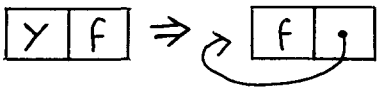
Pictures:



Small: X [X = X . X]

Figure 2. The SKIM instruction set

```
S f g x      -> f x (g x)
K x y        -> x
I x          -> x
B f g x      -> f (g x)
C f x y      -> f y x
S' k f g x   -> k (f x) (g x)
B' k f g x   -> k f (g x)
C' k f x y   -> k (f x) y
Y f          -> f (Y f)
(but this is implemented as follows:)
```




```
If bool a b  -> a if bool is true
             -> b if bool is false
+ p q        -> p + q
Similarly for -, *, /, ...

P a b        ->  a.b  (a list)
U' f g a     -> f a  if a is not a list
U' f g (p.q) -> g p q  where p.q is a list
A b1 b2      -> b1 and b2   (boolean)
O b1 b2      -> b1 or b2    (boolean)
U f x        -> f (car x) (cdr x)
car, cdr
```

and various specialised functions to get at internal representations for debugging purposes.

Figure 3. Machine block diagram

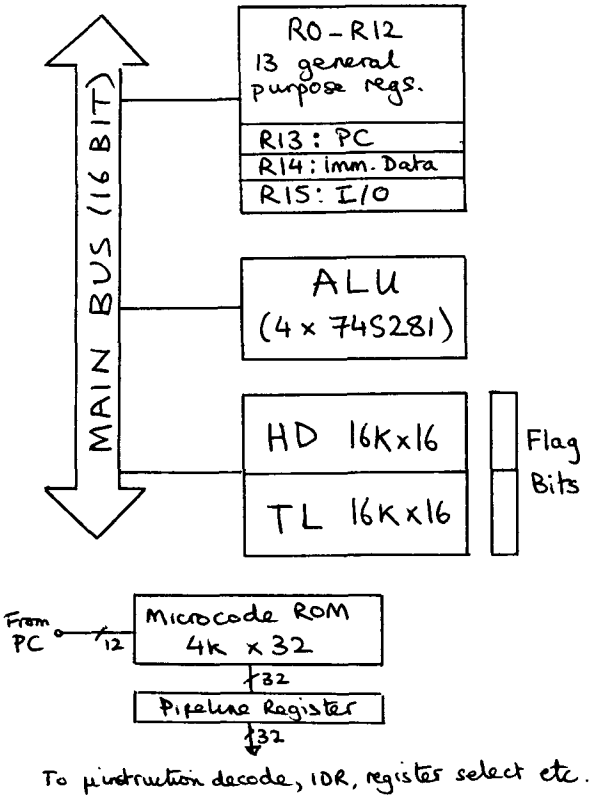


Figure 4. Garbage collector microcode

```
        *  MOV  RA,RB means Move RA to RB
        *  SUB  RA,RB means Subtract RA from RB
        *  WRTL RA,RB means Write RA to tail of RB
        *  RDHD RB,RA means Read head of RB to RA
        *  F=0 means reset flag bit
        *  B=(cond,loc) means jump to loc on cond
        *
|       MARK     MOV   R3,FPTR,B=(JGE,NUMBERP) Is pointer number or immediate data
|       MARK3    RDHD  FPTR,R3,B=(JOF,ISMARKED) Has the cell been marked
|                WRHD  BACK,FPTR,F=1,B=(JMP,MARK) Mark this cell as in use
|                MOV   FPTR,BACK          Move forward pointer to back
|       * That completes the HEAD side mark.
|       NUMBERP  MOV   R3,NULL,B=(JLA,ISNUM) Is pointer numeric
|                SUB   R5,R3,B=(JEQ,ITSINODE)
|       ISMARKED MOV   BACK,NULL,B=(JEQ,DONEMARK) BACK equals NIL ?
|       ISNOTTOP RDTL  BACK,R3,B=(JOF,DONETAIL) are we coming up 2nd time
|                RDHD  BACK,R4            read out the back pointer
|                WRTL  R4,BACK,F=1,B=(JMP,MARK) Mark doing tail and save back ptr
|                WRHD  FPTR,BACK,F=1      restore head side pointer
|       ISNUM    RDHD  FPTR,R3,B=(JMP,ISMARKED) Read head side so can be marked
|                WRHD  R3,FPTR,F=1        Mark cell (numeric) as in use
|       * That concludes the TAIL side mark. Now we have the upward traverse
|       DONETAIL WRTL  FPTR,BACK,F=0      restore tail side pointer & unset tag
|                MOV   BACK,FPTR,B=(JMP,ISNOTTOP) climb up a level and
|                MOV   R3,BACK,B=(JEQ,DONEMARK) pretend cell was marked
* Vertical bars indicate the minimal garbage collector. The rest is optional
        * We remove indirection nodes for efficiency
        ITSINODE MOV   BACK,R3,B=(JNP,ISMARKED) Are we in an application cell
                 RDTL  BACK,NULL,B=(JOF,ISMARKED) Was I in the head side
                 RDTL  R3,FPTR,B=(JGE,IMMOPND) we get an immediate pointer
                 RDHD  BACK,BACK          Move BACK back a bit so cell lost
                 WRHD  R5,R3,F=0          Restore the head cell and unmark it
                 MOV   FPTR,NULL,B=(JSM,MARK3) Pair pointer
                 RDHD  FPTR,R4,B=(JMP,MARK3) get head of next cell
                 SUB   R5,R4,B=(JEQ,*+1)  Goto mark3 if not an I-node
        * It is an I-node (possibly the same one) [ Y I ]
        * We now mark this node and trace till we find either
        * a not I-node or a marked I-node
                 MOV   R3,FPTR            Head of list of I-nodes
        ANOTHERI WRHD  R5,FPTR,F=1        Flag this cell
                 RDTL  FPTR,FPTR,B=(JNP,NOTINODE) move to next cell
                 RDHD  FPTR,R4,B=(JOF,MKDINODE) get head side
                 SUB   R5,R4,B=(JEQ,ANOTHERI) We have another I-node
                 MOV   PC,NULL,B=(JNE,NOTINODE) Always branch to NOTINODE
        MKDINODE SUB  R5,R4,B=(JNE,NOTINODE)
        * This implies that we have an infinite list of I's. We take the
        * arbitrary decision to make this list finite by changing it to
        * I Fixed-point-of-I-combinator   which is a value held as a literal
                 WRTL  IDR,FPTR,I=FIXEDPTI
        NOTINODE MOV   FPTR,R4
                 SUB   R3,R4,B=(JEQ,MARK) When we reach the last cell
                 WRHD  R5,R3,F=0          Unmark this cell
                 MOV   R3,R8
                 RDTL  R3,R3,B=(JMP,NOTINODE) Go down list of I's
                 WRTL  FPTR,R8            Helps next time
        IMMOPND  RDHD  BACK,BACK          Repeat some earlier code
                 WRHD  R5,R3,F=0,B=(JMP,NUMBERP) restore and unmark head cell
                 MOV   FPTR,R3,B=(JLA,ISNUM)
```

Figure 5.

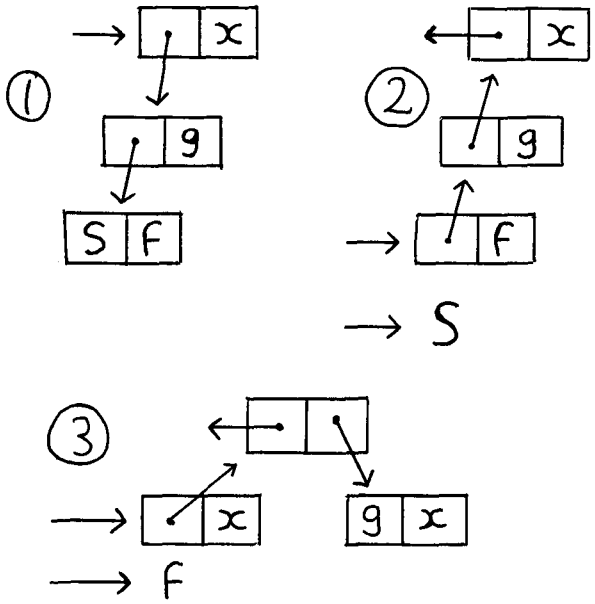


Table 1. Timings for the garbage collector

Machine	SKIM	Z80	M68000	IBM 370/165	
Node Size	34	32	64	64	(bits)
Time/Node	10	64	27	7	(μs)

References.

[1] Turner, D. A. "A new implementation technique for applicative languages" Software Practise & Experience, 1979

[2] Norman, A. C. and Moore, P. M. A. "The design of a vector-based algebra system" Proc. EUROSAM 79, 1979 (Springer Lecture Notes in Computer Science 71, ed: E. Ng)

[3] Mycroft, A. "The theory and practise of transforming call-by-need into call-by-value" 4th Int. Colloq. on Programming, Paris 1980

[4] Mago, G. "A network of microcomputers to execute reduction languages" Int. Jrnl. of Comp. & Inf. Sciences, Oct 79

[5] Berkling, K.

[6] Curry and Feys "Combinatory Logic" North Holland

[7] BASIC Users manual, Acorn Computers, 4a Market Hill, Cambridge. 1980.

[8] Fitch, J.P. and Norman, A. C. "Implementing LISP in a high-level language" Software Practise and Experience, 1977.