Some Practical Methods for Rapid Combinator Reduction

W. R. Stoye, T. J. W. Clarke and A. C. Norman

University of Cambridge Computer Laboratory

Abstract

The SKIM II processor is a microcoded hardware machine for the rapid evaluation of functional languages. This paper gives details of some of the more novel methods employed by SKIM II, and resulting performance measurements. The authors conclude that combinator reduction can still form the basis for the efficient implementation of a functional language.

Introduction

The SKIM I processor was designed and built here at Cambridge several years ago [Clarke 80]. It is a microcoded machine for the reduction of combinators [Turner 79], and it has been a great success as such.

A successor to it, SKIM II, has now been constructed. At first SKIM II was microcoded to reduce combinators in much the same way as SKIM I, but since then its speed of computation has been improved by about fifty percent by careful remicrocoding. This involved several new methods and algorithms, which are the subject of this paper.

The authors' conclusion is that there is still considerable room for improvement in the implementation of functional languages, because they are sufficiently far removed from a hardware-dominated model of computation that the implementor has great freedom in the choice of evaluation methods.

The following subjects will be covered:

the elimination of recursion in the evaluator
one bit reference counts on the heap
equating identical structures transparently
compilation of complex operations into microcode

Little will be said about SKIM II's hardware except that it is a simple, microcoded machine fairly similar to SKIM I. A description can be found in [Stoye 83]. The main improvements are a larger address space, the allocation of more tag bits to each word, and microcode in RAM. SKIM I was an experimental hardware device, SKIM II is in contrast a stable basis for software experimentation.

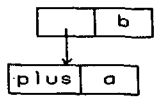
The Elimination of Recursion

A new scheme is presented for the elimination of recursion from a combinator evaluator when dealing with strict functions.

Turner's original scheme for combinator reduction is that expressions should be represented as graphs, with function application denoted by head linked lists of binary cells. For example, the expression

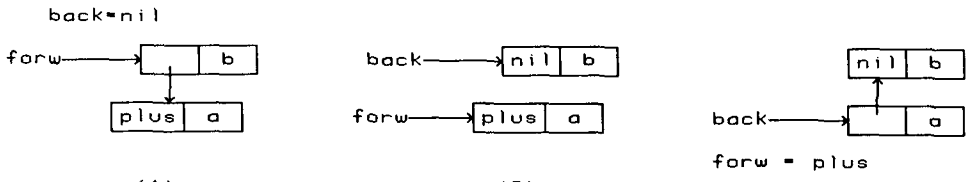
plus a b

would be represented as shown in diagram 1. Note that the 'application' pointers shown in this diagram are distinguishable from data pointers, so that a cell containing a and b represents 'b applied to a' rather than 'a pair with a on the left and b on the right'. Fundamental functions like 'plus' are represented by objects known as combinators. Turner's algorithm for evaluating this object was to follow the chain of application pointers in the head sides of cells, pushing onto a stack the addresses of cells past. When a combinator object (like plus) is reached at the extreme left hand end, a jump is performed in the reducer to code specific to that combinator.



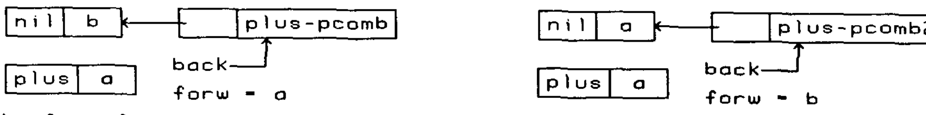
plus a b

diagram 1: graph representation of an expression

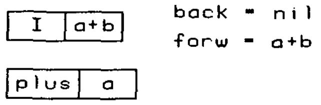


(1) (2) (3)

diagram 2: Pointer reversal during the evaluation of an expression



(4) after first transformation (5) after second transformation



(6) final result

diagram 3: Reduction of the expression plus a b using postfix combinators

The code for plus, to continue this example, would check on the stack that it had two arguments, call the evaluator recursively on each of its arguments (they should both evaluate to be integer objects in this case), add their values together and overwrite the top cell in the diagram with the new value. This overwriting has to occur because there might be other application pointers to this fragment of the graph, and it is an important principle of lazy evaluation that no computation should be performed twice.

Note that plus is a 'strict' combinator in that it has to evaluate its arguments. Other combinators (like S and K) do not have to do this, but in practice many of the combinators evaluated are strict ones.

There are two uses for a stack in this process, one to record the sequence of cells passed when traversing the left hand sides and another associated with the need to evaluate the arguments of some functions before performing the function itself. Both version of SKIM use pointer reversal to cope with the first of these. Two registers are used when evaluating called '**forw**' and '**back**', to hold the expression being evaluated and the list of reversed pointers. To evaluate an expression **forw** is initialised to point to the graph representing the expression and a null value is placed in **back**. Evaluation proceeds as shown in diagram 2.

At this point evaluation continues at the code specific to the plus combinator. The combinator has to re-reverse the pointers in order to find its arguments, and so that the tree is left in a tidy state after the reduction. On SKIM II pointer reversing performance is enhanced through the use of read-modify-write memory access cycles so that fetching the forward pointer and writing the backward pointer occur in one memory cycle.

SKIM I uses a stack for the implementation of strict combinators, when the evaluator is called recursively to evaluate the arguments. This stack is implemented as a list of cells on the heap, mainly because there is no hardware support for anything else.

SKIM II extends the idea of pointer reversal to deal with strict arguments. This is reminiscent of the pointer reversal in Deutch-Schorr-Waite garbage collection [Schorr 67], where both left and right subtrees of a cell must be marked without using any extra storage. We continue the pointer reversal process when evaluating the arguments a and b, leaving a special marker in the back chain saying where to continue computation of the suspended 'plus' operation. The first part of the code for plus reforms the graph to look like state (4), shown in diagram 3. The (plus a) cell has to be left unchanged, in case there are other pointers to it. A new cell has been generated, which sits on the back chain and contains a special marker which is designated plus-pcomb in the diagram.

This marker is called a postfix-combinator or pcomb and it contains an entrypoint to the reducer, just like a normal combinator. Evaluation now continues, and the argument a becomes fully evaluated.

When the **forw** register contains an object that is fully evaluated, the reducer looks at the backward pointer. If it contains the null value, evaluation is complete. If not, the chain of pointers that it represents will be re-reversed until a cell is found with a postfix-combinator in its tail. A jump will occur in the reducer to the code represented by this pcomb.

In this way the first argument has been 'recursively' evaluated with a minimum of state-saving in the evaluator. There is now no data structure involved in evaluation except the expression graph itself, which fits in well with the reduction paradigm that SKIM is designed to support. The further evaluation of plus in fact involves a second pcomb, because the evaluator has to be 'reentered' again for the evaluation of b, but this is merely a repetition of the above process.

There have been two main benefits from the implementation of this algorithm, the first being a performance improvement of about ten percent overall due to reduced state-saving in evaluating strict combinators. The second is that the reducer has a very small amount of internal state, and is in fact simpler than the first version of the microcode, which used a stack. This proved a great advantage when adapting SKIM's microcode to provide a multi-tasking model of parallel reduction [Stoye 84]. The combinator expression graph in SKIM's main memory is reduced at many places 'simultaneously', to simulate the operation of a machine with many processors, and it is very important that this should not cause incorrect interactions when several processes try to reduce the same expression graph at the same time. Keeping all state information as part of a single graph structure has helped us concentrate on the interlocks that are required. The effect of having multiple physical processors involved in this operation has not yet been considered by us in detail.

It might be argued that better performance would be obtained from SKIM by the addition of a separate hardware stack, which could be used to remember the arguments of any suspended operations. But this would add substantially to the complexity of SKIM's hardware, and would have made multi-tasking more complicated, while better performance improvements could probably be gained by a simple associative cache on the main memory.

One Bit Reference Counts

One problem with combinator reduction is that the use of heap storage seems profligate. The process of reduction involves the continuous use of heap cells, and although SKIM's memory structure is carefully optimised to allow rapid marking during garbage collection it was found to be spending more than half of its time in the garbage collector when memory was only one third full. Most cells are only in use for a very short time, and then discarded.

Reference counting is often suggested as a replacement for or an addition to garbage collection, but the bookkeeping overheads concerned with doing it for very small scale objects are considerable. Every time a pointer to an object is copied the object has to be accessed and its reference count incremented and written back. As it is necessary to cope with increment overflow this can be an absurdly complex operation, so that although most storage may be reclaimed computation would proceed overall at a slower rate.

SKIM uses reference counts in a way that speeds up computation even for calculations that do not provoke garbage collection, as well as reducing garbage collection. This is done by maintaining one bit reference counts in pointers rather than in objects.

The method is that any application pointer is identifiable as being either a 'multiple application pointer' or a 'unique application pointer' (**mappl** and **uappl**). Use of a **mappl** is synonymous with the conventional meaning of an application pointer, but use of a **uappl** has the additional meaning that 'there are no other pointers to this cell'. Thus, if a **uappl** pointer is being discarded then the cell that it designates is known to be free and can be given back to the memory allocator (ie placed at the head of the freechain).

If an object is being copied, there is no need to access it under this scheme. If the copied object is a **uappl** then it must be changed to a **mappl**, but this is a very simple operation and does not take very long. It is not useful to have more than two states, because the increment operation would have to change all other pointers to the object and that is clearly not possible.

This can speed up computation when reducing combinators, because new cells are frequently needed just as old ones become available, so that overheads associated with freechain maintenance are reduced. As an example, consider the S combinator which performs the following action:

S a b c = a c (b c)

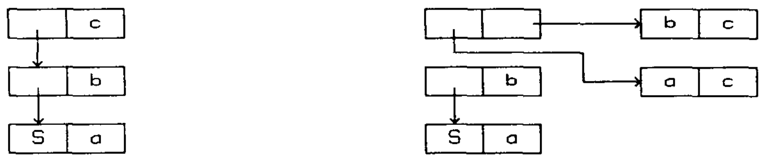


diagram 4: Conventional reduction of the S combinator
S a b c is transformed into a c (b c)
(pointer reversal has been omitted for clarity)
two cells are generated

u——→ denotes a unique application (uappl) pointer
m——→ denotes a multiple application (mappl) pointer

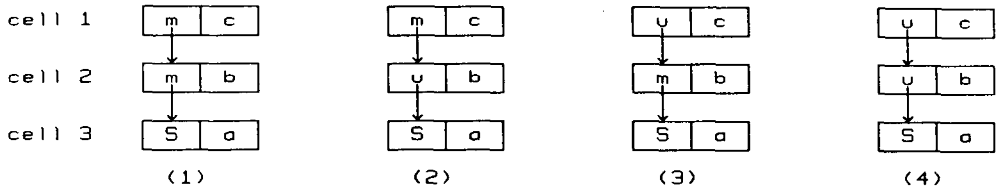
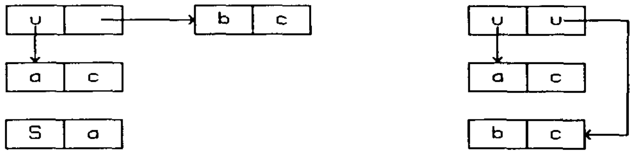


diagram 5: the four different possibilities for S a b c
with respect to reference counts



case (3) after reduction
one cell generated

case (4) after reduction
no cells generated

diagram 6: Reduction of S a b c using reference counts

In terms of graph structures this is represented by the transformation shown in diagram 4. Note that the original cells have to be left, in case there are other pointers to the middle of this structure. Two new cells have been generated, and the value 'c' copied without being evaluated.

In terms of **uappls** and **mappls**, there are four forms in which this expression may appear, depending on the two pointers in it. These are shown in diagram 5.

When cases one and two are encountered, the reduction method is unchanged. In case 3, cell 2 will not be accessible after the reduction and so can be reused. In case 4, cells 2 and 3 can both be reused, which is very convenient as we were just about to generate two new cells. By not giving the inaccessible cells back to the space allocator, but reusing them straight away, the reduction of this combinator will proceed faster. After the reduction, cases (3) and (4) will appear as shown in diagram 6.

Deciding which of these cases to use happens while picking up the arguments, and works very neatly. Although the amount of microcode for each combinator is increased because of the number of different cases to prepare for, in practice most reductions salvage some space in this way. The S combinator has an additional complication in that it duplicates its third argument, so if 'c' is a **uappl** it must be changed to a **mappl**. Some other combinators (like plus) have free cells that they cannot immediately use, these are placed on the freechain.

The results of applying this technique have been spectacular - on average, about seventy percent of wasted cells are immediately reclaimed. Detailed timings were made for two small computations. The first is defined by

```
tak x y z =  If     y >= x
             Then   z
             Else   tak ( tak ( x - 1, y, z ),
                          tak ( y - 1, z, x ),
                          tak ( z - 1, x, y ) );
tak 18 12 6
  (the answer should be 7)
```

This test is called Takeuchi's function and has been used on a number of Lisp systems as a crude benchmark. The second computation is the sorting of 2000 numbers, using a very simple minded sort program that divides its input into two lists of

test	free memory	microcode version 1		microcode version 2		microcode version 3	
name	(K nodes)	secs	GC's	secs	GC's	secs	GC's
tak	125	16	12	15	5	13	4
	105	16	13	15	6	14	6
	75	18	14	17	9	14	7
	45	29	32	20	15	19	12
	15	82	96	38	41	36	38
sort	125	23	18	21	6	17	5
	105	25	22	21	7	18	6
	75	32	32	23	9	19	8
	45	48	58	27	17	23	15

table 1: Speed improvements for reference counts and recursion removal

equal length, recursively sorts the two sublists and then does a binary merge. The time spent is almost independent of what the numbers are. The results are shown in table 1.

The table shows measurements for three different versions of microcode, running with varying amounts of free memory. The machine has a total of about 130,000 cells of heap. For each run the total time t in seconds and the number g of garbage collections was recorded. Version 1 microcode did no reference counting and used recursion to evaluate strict functions, version 2 microcode did reference counting, version 3 also used pointer reversal instead of recursion for strict functions. As the figures show, performance degredation with memory full is dramatically reduced when using reference counts.

It is important to note that the wasted cells saved by this technique form a different set to those saved by eliminating redundant laziness [Mycroft 81]. Compiled code using this technique is slightly more complex (but not unmanageably so) and would benefit considerably.

At the moment this technique has only been implemented on application pairs, but there is no reason why it should not be used on data pairs too. Indeed, it could be useful in any heap storage system.

Equating Equal Structures

One frequently used combinator compares two data structures recursively for equality and returns a boolean value, like the Lisp function EQUAL. The implementation on SKIM has a secret side effect, that if the objects are equal their space is commoned up. This not only saves space, it speeds up future comparisons of the same objects.

This would not be possible in a Lisp system, because future use of RPLACA and RPLACD would cause chaos. There is also the more subtle possibility that future use of EQ would generate a misleading reply. Optimisations like this can be performed safely on a functional machine because there are no side effects to consider, and because there are no functions whose semantics are based on the conventional view of computer memory. It would be perfectly legitimate to common up equal objects or expressions wherever they are found, and this has a number of possibilities for further optimisation.

In idle time there is no reason why the machine should not scan over the heap and common up any equal structures that it finds. Although no measurements have been done, we estimate that combinator expressions could be reduced to about half their previous size by commoning up equal parts. This is becuase of the very small number of distinct combinators. It would be very interesting to implement Hashcons [Goto 74] on SKIM, as many of the conventional arguments against it for Lisp systems do not apply, but this has not yet been done. The saving of space would provide a considerable performance boost, and structure comparison would always be instantaneous.

Another operation that could be performed in any idle time is the reduction of any combinators that have enough arguments to be reduced, and whose reduction would not increase the size of the resulting expression. This limitation, or perhaps a less severe one, is needed to prevent store being filled with unwanted expansions of expressions.

combinators		
length	7000 per second	
reverse	7000 per second	
append	7000 per second	
equal	1650 per second	
nfib benchmark	8000 per second	
hand compiled		
length	220000 per second	(*30 speedup)
reverse	120000 per second	(*18 speedup)
append	30000 per second	(* 6 speedup)
equal	20000 per second	(*13 speedup)
tak	tak 18 12 6 in 1.5 seconds	
nfib	90000 per second	

table 2: Speed improvements for microcode compilation

Compilation into Microcode

The lack of fast implementations for functional languages is frequently bemoaned, as being a serious stumbling block in the examination of large software systems written in a functional style. We have chosen to deal with this problem by constructing special hardware to implement existing techniques of evaluation (ie combinator reduction), but many other groups are attempting to develop new techniques for the evaluation of functional languages.

In particular, a great deal of work is being done on the design of sophisticated compilers targeted towards conventional machines, and theoretical and practical knowledge in this field is increasing all the time. Work in Cambridge [Fairbairn 82] parallelling the SKIM development has produced a compiler from a pure functional language into code for a 68000 microprocessor [Motorola 82]. This compiler's code is orders of magnitude faster than interpreted combinator reduction on a 68000, and it would be highly beneficial if SKIM could make use of advances of this kind.

Microcode is usually viewed as a small control program sitting in fast, expensive RAM: it is part of the main processor, as an engineer's implementation mechanism for complex hardware. Even on machines with microcode in RAM such as the DEC VAX and the Xerox Dorado, most microcode is hand written by experts as part of systems implementation.

SKIM is better viewed as a machine with separate data and program memory, with the structure of each optimised for its intended use. The program memory is only microcode in as much as it is viewed as being static. It is implemented using cheap 150 nanosecond static RAMs organised into 40 bit words, with a 250 nanosecond cycle time. There is a 16 bit microcode address bus, so that the processor could support up to 64K*40 bit words of microcode. Using 8K*8 static RAMs, which should be cheap and common within the next year, this occupies one Extended Double Eurocard.

Thus there is no reason why SKIM microcode should not be generated by compiler, from a functional language. As the same language is used to generate combinators, compilation into microcode would not require functional programs to be rewritten, merely submitted to a different version of the compiler. This would allow the programmer to boost the performance of important or frequently used functions while retaining the flexibility of combinators for less critical ones.

Some small functions have been 'compiled by hand' for SKIM, which are nevertheless more complex operations than what is generally viewed as being a 'combinator'. The microcode instruction set was found to be no more difficult to work with than the machine code of a conventional machine. The functions compiled so far have included the recursive structure comparison EQUAL, and reverse and append for lists. The results are shown in table 2, and demonstrate a speedup by a considerable factor over combinator evaluation. The measurements are crudely expressed in terms of cells per second of action on a linear list. Reverse and equal are completely strict and compile into microcode loops, while append does not. Garbage collection time is included. A compiler's code would not be quite as fast as this, but should be close.

The best strategy for compilation appears to be to remove laziness wherever possible, as laziness means the generation and reduction of more graph structures. It is clear that some functions are more susceptible to this than others, and so while some functions (e.g. simple recursive numerical computations like factorial) would speed up far more than this, others (complex manipulation of high level functions) will hardly speed up at all. We do not have any evidence yet about the overall effect of compilation on large programs.

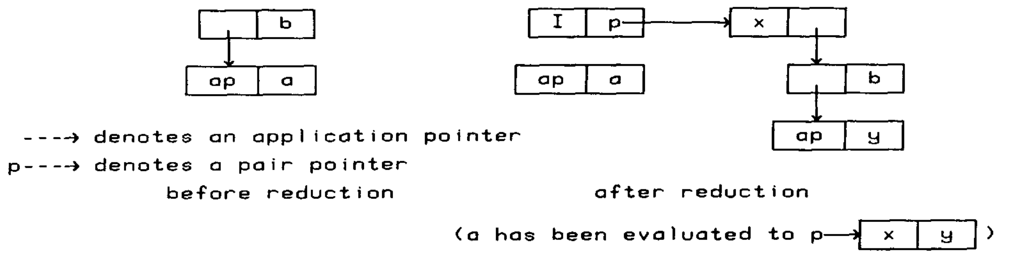

b
ap
a
---→ denotes an application pointer
p---→ denotes a pair pointer
before reduction
I
p
x
ap
a
b
ap
y
after reduction
(a has been evaluated to p→ x y)

diagram 7: The append combinator.

The best 'grain' at which to work appears to be that of 'supercombinators' [Hughes 82]. For instance, the append combinator performs the following operation shown in diagram 7. The argument **a** must be evaluated and examined. If it evaluates to a data pair then the head and tail halves are fetched and a new structure built as in (2). Otherwise, the result is **b**. Three new cells are needed, although in practice the reference counting mechanism will collect up the two superfluous ones.

We do not yet have any practical experience of how large programs written in a functional language will perform when compiled, but (as an example) current compilation techniques [Mycroft 81] will not reduce the number of new cells used in the append example above. Even when compilers become common for functional languages, SKIM should still provide competitive performance.

Results and Conclusions

Details have been given of various optimisations to Turner's combinator reduction method. Many of these methods are not at all obvious, and there may be many more similar ideas waiting to be discovered. They have resulted in considerable performance improvements for SKIM.

It is important to note that all of these improvements are transparent to the high level, functional programmer. They have been possible becuase the combinator model of computation is not tied to any one hardware scheme. By resisting the temptation to bend the high level programming model towards what is efficient to implement, as I believe has happened to many Lisp and Prolog systems, the implementor keeps future options open as well as ensuring software compatability. Although current implementations of functional languages may not rival more serious implementations of Lisp, future improvements in implementation expertise are likely to narrow this gap over the next few years.

Are all of these optimisations dependent on special hardware? Implementors of functional languages on conventional machines may conclude that the methods described here are of no interest, becuase they rely on the particular strengths of the SKIM processor. To address this issue it is worth examining the motivation for the SKIM project. There are two main reasons why we felt that it was worth building specialised hardware to execute functional languages.

By having a moderately powerful test implementation more substantial test programs will get tried. Although we do not provide detailed measurements here, considerably larger programs than simple sorting routines (e.g. a screen editor, a small compiler) have already been run on SKIM, and we are confident that over the next year more system software will get written. A simulator or an interpreter running on already available hardware may appear to provide greater flexibility in trying new implementation techniques, but will not lead to such meaningful tests of the true usefulness of the result.

We argue that 'conventional' machines are specialised to the running of stack-based languages such as Pascal, and that in fact SKIM is a <u>less</u> specialised hardware device. Efforts to implement functional languages efficiently on conventional machines are dominated by the benefits gained from transforming the language to conform with the conventional stack-based model of computation. We consider it more interesting to study how to do graph reduction better, rather than how to remove graph reduction from the program. The result is a very different perspective on what strategies are productive, and (we hope) better insight into how future implementations should be constructed.

Note that the use of multiple processors to increase performance is orthogonal to our work, although it appears that our techniques map onto multiple processors more easily than other approaches that take advantage of more conventional architectures.

The developments discussed here are also orthogonal to improvements in techniques for generating combinators (e.g. work by M. S. Joy at the Universiy of East Anglia, and S. L. Peyton-Jones at University College London). Current methods for generating combinators are still far from satisfactory, but when improvements happen SKIM should be able to make use of them.

Although such things are difficult to measure absolutely, SKIM appears to deliver a performance in solving common problems that is about one-quarter of that of a conventional compiled language on hardware of similar cost. The machine used for this comparison had a 68000 CPU running at 8-MHz, running BCPL and Algol68C as representative high level languages. Programming is far simpler, being in a functional language, and our future research is in the organisation of large programs and system software on such a machine.

References

[Clarke 80]: T. J. W. Clarke, P. J. S. Gladstone, C. D. MacLean and A. C. Norman, SKIM - the S, K, I Reduction Machine, Proceedings of the 1980 ACM Lisp Conference, pp 128 - 135, August 1980.

[Motorola 82]: MC68000 16-bit Microprocessor User's Manual, Prentice-Hall, 1982.

[ACM 82]: Conference record of the 1982 ACM Symposium on Lisp and Functional Programming, August 15-18, 1982, ACM order no 552820

[Hughes 82]: R. J. M. Hughes, Oxford University, Super Combinators: A New Implementation Method for Applicative Languages, [ACM 82].

[Fairbairn 82]: J. Fairbairn, Ponder and its Type System, Cambridge University Computer Laboratory Technical Note, 1982.

[Turner 79]: D. A. Turner, A New Implementation Technique for Applicative Languages, Software Practice and Experience, Volume 19, pp31-34, 1979.

[Mycroft 81]: A. Mycroft, Abstract Interpretation and Optimising Transformations for Applicative Programs, University of Edinburgh Dept. of Comp. Science, 1981.

[Goto 74]: E. Goto, Monocopy and Associative Algorithms in an Extended Lisp, University of Tokyo, Japan, May 1974.

[Schorr 67]: H. Schorr and W. Waite, An efficient Machine-independent Procedure for Garbage Collection in Various List Structures, Comm ACM 10, 8, pp 501-506, August 1967.

[Stoye 83]: W. R. Stoye, The SKIM II Microprogrammer's Guide, Cambridge University Computer Laboratory Technical Note, 1983.

[Stoye 84]: W. R. Stoye, An Operating System written in a Purely Functional Language, document in preparation.