AN INVESTIGATION OF THE RELATIVE EFFICIENCIES OF COMBINATORS AND LAMBDA EXPRESSIONS

by

Simon L Peyton Jones
Beale Electronic Systems Ltd
Whitehall, Wraysbury, UK.

ABSTRACT

In 'A New Implementation Technique for Applicative Languages' [Tu79a] Turner uses combinators to implement lambda expressions. This paper describes an experimental investigation of the efficiency of Turner's technique compared with more traditional reducers.

OVERVIEW

The basis for comparison of the two systems is discussed in Section 1. This is followed by some implementation considerations in Section 2, while the main results are presented in Section 3. Section 4 presents some discussion of the results and related issues, and conclusions are drawn in Section 5.

1 BASIS FOR COMPARISON

1.1 Background

Functional languages are characterised by the absence of side effects and imperative commands. They are the focus of considerable current interest for two main reasons

(i) Their properties of referential transparency make them easier to reason about than conventional languages, and consequently programming errors are less likely, and programs are more amenable to formal verification.

(ii) The absence of side effects means that expressions can be concurrently evaluated by several cooperating processors. This suggests functional languages as a base for highly parallel computing.

The two main techniques for efficiently implementing functional semantics are data flow and reduction. This paper concentrates exclusively on the implementation of reduction techniques.

The cannonical reduction architecture is the lambda calculus, which has an extensive literature (eg [Ch41], [St77b]). However, some old results derived by Curry and Feys [Cu58] have been used by Turner [Tu79] to implement a reduction machine for the combinator calculus. The combinator calculus has the same semantics as the lambda calculus, but has a rather different implementation. Thus the two calculi can be thought of as two machine codes for a functional .igh-level language. The question of which implementation performs a computation most efficiently then arises naturally, and is the question addressed by this paper.

1.2 Combinators

The combinator calculus is sparsely documented, but the source text is Curry and Feys [Cu58], while a more readable and operational treatment is in [Tu79a,b]. In this paper Turner demonstrates Curry and Feys' technique of abstracting variables from lambda expressions to produce an expression consisting solely of constant operators (combinators) and data. Curry and Feys have shown that two combinators (called S and K) are sufficient to implement all lambda expressions, where

S = LAM f. LAM g. LAM x.(f x (g x))

and

K = LAM x. LAM y.x

However Turner suggests a set of combinators which include several simple optimisations of S and K, together with combinators designed to prevent an explosion in the size of an expression when abstracting several variables at once.

Combinators lend themselves naturally to 'lazy evaluation', a technique whereby the evaluation of arguments of a function is postponed until the value of the argument is required, and the result of any evaluation is made available to other uses of the same argument in the function. This technique allows the construction of entities such as infinite lists.

It has also been suggested that combinators are a useful basis for program transformation and verification [Di81].

Wand [Wa82] describes a technique for rewriting the denotational semantics of a language using combinators, to obtain a compiler for the language and the architecture for a suitable target machine. Despite an abstract development, the resulting operation of the machine turns out to be similar to that of a conventional one.

There is a commercially available extensible combinator reducer, together with a lambda to combinator converter, called CRS/1 [Be82a]. This system is written in C and has so far been installed on a microcomputer and a VAX.

A group based at Cambridge University is at present building the successor to SKIM [Cl80], a machine specifically designed for efficient combinator reduction. Based on bitslice technology, SKIM is a conventional von Neumann machine with an instruction set and memory layout suitable for graph operations.

1.3 Comparison

To avoid dispute over the precise choice of the metric used to compare the two implementations, the view taken here is that the principle costs of the computation should be measured, leaving the exact metric open to choice. These costs can be counted in terms of the number of accesses to data structures whose size is potentially unbounded, in particular the stack and heap used by both reducers. This view, which minimises the importance of fine details of the reducers themselves, is taken on the grounds that

(i) The reducers, being of bounded size, may be stored in fast memory, microcoded into the processor, and provided with special hardware, all at fixed cost.

(ii) Intra-processor acceses will be very much faster than accesses to a very large global memory, which must increase as O(cube root memory size), and which are usually slow for economic reasons.

Accesses to the heap and stack are accounted separately in this investigation since stacks are very amenable to optimisation.

The comparison between the two systems was therefore performed by writing reducers for each, and counting accesses to the heap and stack only, ignoring elapsed runtime. The results therefore do not depend on the efficiency of the reducers themselves, or the machine on which the experiments were performed. This gives a more objective measure for the cost of running a reducer, but there is a set of design decisions to be made when writing the reducers which will affect their performance by the above criteria. These implementation decisions will now be described.

2 IMPLEMENTATION CONSIDERATIONS

Some of the main design decisions made when writing the reducers are described in this section. In addition the main evaluation loop of each reducer is shown in the appendix.

2.1 General

Both reducers operate with a heap consisting solely of two cell nodes. Stacks for the reducers are implemented as separate entities (not as linked lists in the heap).The reducers are written with entirely static variable allocation, and explicitly save necessary state information (including a return address) on their control stack during recursion (for instance when evaluating operands for '+'). Recursion in the functional expressions is handled by using the Y operator (defned by Y f = f (Y f)) rather than by static loops in the expression.

2.2 Reduction order

The Church-Rosser theorem states that normal order evaluation will terminate if any evauation does. However for a lambda reducer, this involves forming a closure for the argument of a function in order to postpone its evaluation. This process is potentially expensive, and in practice most lambda reducers implement applicative order (strict) evaluation (in which the argument is evaluated prior to the application of the function). This impementation will give the same results if it terminates, but will fail to terminate in some cases where normal order evaluation would do so. However, it turns out that normal order is the 'natural' order for a combinator reducer. Three reducers were therefore written: a normal order lambda reducer (NLR), an applicative order lambda reducer (ALR) and a combinator reducer (CR).

2.3 Data types and operations

The reducers support the data types Integer, Character, Pair, and Boolean. Pairs behave like LISP cells, with operations 'cons', 'car', 'cdr'. Strings are supported as lists of characters, while input is handled by regardng an input stream as a string. The function 'open' takes a string identifying the file as its argument, and returns a string wich is the contents of the file.

2.4 Implementing the lambda reducer

The environment is implemented as a linked list in the heap (this makes it easy to 'capture' the environment in a closure). Environment lookup is not precompiled to a fixed offset in the environment (which involves no test for a matching name). The lambda reducer is tail recursive [St77a], which saves Control stack accesses in essentially iterative situations. The applicative order lambda reducer differs from the normal order reducer in several respects in addition to the immediate difference of evaluating the argument before applying the function to it. In particular, a special case has to be made for the 'if' function, and for input streams (otherwise the system would read in a whole file before delivering its first character).

2.5 Implementing the combinator reducer

Turner [Tu79a] contains a good description of the details of implementing a combinator reducer, and the combinator reducer implements the combinators S, K, I, B, C, S', B', C', Y which he describes. Combinator reduction involves reducing graphs, instead of environment manipulation, and requires an extra stack (called the Reduction stack). The stack can also be implemented with a pointer reversal scheme.

3 EXPERIMENTS AND RESULTS

3.1 Test expressions

In order to perform the comparison, it was necessary to have a number of text expressions. These were written in a functional language called Nose, due to Hughes [Hu80]. The Nose compiler transforms the test expressions into lambda expressions, and expands recursive calls into applications of Y. It is difficult to meet the criticism that the choice of test expressions might be biassed. However an attempt was made to pick a set of fairly 'typical' expressions, which are described individually below. They all share one factor in common, in that they are all small expressions. It might perhaps be argued that very different results would be obtained for larger expressions. In particular, none of them will have very many names in scope at any point, so the nvironment will be small for the lambda reducer. Unfortunately, simulations are extremely costly in computing resources, so investigation of 'real' sized problems was precluded. The test expressions used were as follows:

Factorial. Computes the first 10 factorials recursively.

Fibonacci. Uses the doubly recursive algorithm to compute the first 7 Fibonacci numbers.

List reverse. Reverses a list of 15 elements.

Primes (1). Computes the first 15 primes by test division.

Primes (2). Computes the first 10 primes by a functional Seive of Eratosthenes.

Permutations. Permutes 3 elements.

Towers of Hanoi. Uses 5 discs on 3 pillars.

Ackermann. A good source of deep recursion, called with arguments 2 and 3.

Sort. Sorts a list of 7 elements by linear insertion.

Bracket abstraction. Performs the lambda to combinator transformation on the expression LAM x.(LAM y.(a x b y c)), using S, K, I only.

Twice. Twice f x = f (f x) in the expression Twice Twice Twice succ 1

Numbers. Defines and manipulates 'functional numbers'. The number n is represented as a function of two argments, such that: n a b -> a (a (a ....(a b) ...)) here there are n applications of a to b. In this symbolic arithmetic system, the expression calculates 3 squared and the first 5 factorials.

3.2 Results

Each reducer was used to reduce each test expression, and results were accumulated for the following parameters:

Size of expression (in tokens)
Number of store accesses
Amount of store claimed
Number of stack accesses
Maximum stack depth

The number of store accesses includes all heap accesses, including those made during cons operations, but excluding stack accesses. The number of stack accesses includes both control and reduction stacks for the combinator reducer, and the maximum stack depth is calculated as the sum of the maximum depths of both stacks. In some cases no figure is supplied for the applicative lambda reducer. This is because these expressions involved infinite lists which would cause it to fail to terminate.

Table 1 shows various performance indicators for the ALR and NLR, normalised to those for the CR. Thus a figure of 2 in the 'store claimed' column means that the lambda reducer claimed twice as much store as the combinator reducer. Figures for the NLR are in brackets.

Fxpression	Size of expression	Store accesses	Store claimed	Stack accesses	Stack depth
Factorial	0.85	2.9(4.8)	2.2(4.6)	1.6(2.9)	2.7(5.0)
Fibonacci	0.8	2.7(4.5)	2.2(4.5)	1.6(2.8)	3.2(5.8)
Primes (1)	0.8	(4.2)	(3.4)	(2.4)	(13.8)
Primes (2)	0.85	(5.2)	(4.6)	(2.9)	(7.0)
List reversing	1.0	3.3(4.5)	1.4(3.8)	1.6(2.8)	3.2(10.3)
Permutations	0.9	2.7(3.8)	1.3(3.1)	1.3(2.0)	1.7(6.2)
Towers of Hanoi	0.7	1.9(3.5)	1.4(2.8)	0.8(2.4)	4.3(2.1)
Ackermann	0.75	1.9(3.2)	1.4(2.8)	1.3(2.2)	3.3(12.2)
Sort	0.75	1.6(2.7)	0.8(2.1)	1.0(1.7)	3.3(4.8)
Bracket abstr.	0.75	2.6(4.7)	1.6(4.2)	1.5(2.4)	4.0(2.8)
Twice	1.85	2.0(3.0)	2.1(3.0)	1.3(2.0)	2.5(4.9)
Numbers	1.3	(4.4)	(3.4)	(3.3)	(22.3)
'Average'	0.9	2.4(4.0)	1.6(3.4)	1.3(2.5)	3.2(8)

Table 1

3.3 Other results

Though the full tables are not presented here, the data analysed suggested the following further general conclusions:

(a) The CR does approximately 1.3 times as many store operations as stack operations, while both the ALR and NLR do about 2.2 times as many store operations as stack operations. This means that using fast hardware to support the stack(s) would have a significant effect in both cases, but the CR would benefit more than the NLR or ALR.

b) Both the lambda reducers spend roughly half their store accesses in environment lookup. This suggests that a really substantial improvement in this area could make the ALR comparable with the CR, but the NLR would still be slower even with free environment lookup.

c) Nearly a quarter of all reductions performed by the CR are I-node reductions. This suggests that using the garbage collector to collect redundant I-nodes would make a significant improvement to the CR.

d) The Primes (1) expression was run to produce larger lists of primes (up to 100 primes).The factor by which the CR outperformed the NLR (in terms of store accesses) reduced from 4.0 to 3.5 and then appeared to stabilise. This suggests that the results hold for larger computations as well.

4 DISCUSSION

This section discusses some further issues involved in the choice of implementation of a functional language.

4.1 Optimisations

It is possible that the results in this paper are swamped ty the effects of the various possible optimisations that may be performed on the reducers. However the range of possible optimisations does not seem to strongly favour one reducer or the other, and the results presented in this paper may form a basis for further work. A brief summary of the major optimisations known to the author will now be given.

Stack optimisations

Both reducers use a stack fairly heavily, and there are several well-known optimisation techniques for stacks. The simplest is to provide a fast hardware cache for the stack, while other ideas include racks [St79b] (a hardware assisted on-the-fly peephole optimiser for stack access) and 'phantom' stacks [St79a] (which speeds up access to stacks held as linked lists in the heap).

Combinator optimisations

The most obvious fine tuning that can be done to a combinator reducer is to choose the best set of combinators (the 'instruction set' of the machine). Very little is known at the moment about the effect of using different combinator sets, but it is unlikely that the set chosen for this investigation is optimum. Hughes [Hu80] suggests that a significant speed improvement can be achieved by choosing a combinator set specifically for the expression to be reduced (he calls these expression-specific combinators super-combinators). The compilation step to generate the super-combinators can be done in linear time.

It is possible to use the garbage collector to 'short out' I-nodes in a combinator expression. These serve only as indirection nodes and waste both space and time. In view of the result that almost a quarter of all reductions are I-reductions, it seems likely that this will give some speed increase at very modest cost. This technique is implemented in CRS/1 and SKIM.

Lambda optimisations

A possible semi-compilation which would have considerable benefits is the compile-time determination of environment search paths. The state of the art in this and other optimisations is probably represented by the MIT Scheme chip [Su81].

4.2 Compilation

It seems at first that the most significant optimisation to both reducers would be to compile the expressions to the machine code of the target machine. However, there are really two phases to compilation. The first preprocesses the expression, doing any work that only needs to be done once (predetermining offsets in the environment is a classic example). The second phase generates code that the target machine can execute directly.

An alternative to the code generation phase is to microcode a reducer into the target machine, so that it can execute the output of the preprocessor directly (most machines' instruction sets are in any case interpreted by microcode). This view suggests that the benefits obtained from code generation could equally well be gained from re-microcoding the target machine. The more fundamental benefits of compilation come from the preprocessing phase.

For conventional machines, of course, compilation is very beneficial. There are many LISP compilers available, and Johnsson [Jo81] descibes a design for a combinator compiler. However no performance figures are available for the latter as yet.

It is possible to regard a lambda to combinator converter as a preprocessor which performs the operation of replacing references to variable names with pointers to the relevant piece of code. This operation is performed by all compilers, from assemblers upward, and we might expect it to give the combinator reducer an advantage over the lambda reducer.

4.3 Scale effects

An important question is whether the results of Section 3 will continue to hold for large expressions, and deep recursion. For larger expressions the CR will have to use many reductions to work the library functions into the middle of the expression where they are needed, while the LR will have to search a longer environment. However, the CR will only have to perform these overheads once, since the results of a reduction (eg working in a library function towards the middle) are never forgotten, whereas the LR has to look up the name of the function every time it is used. Thus the CR may perform even better on larger programs. Preliminary results (3.3 (d) above) support this claim.

4.4 Other considerations

The raw efficiency of the two reduction systems is not the only factor that will be taken into account when choosing an implementation of a functional language.

Functional programs may be harder to debug than imperative ones, because of the absence of 'flow of control' and 'state' information. This is particularly so in the case of combinator expressions, since they have no variables. On the other hand, the lambda model corresponds fairly well with the 'way programmers think', and rather sophisticated debugging systems exist for some LISP implementations. A considerable amount of work will need to be done before combinator implementations can rival such systems, but there is no reason to suppose that it is impossible. This matter is further discussed in [Be82b].

Both lambda and combinator expressions may be concurrently evaluated by several processors. However, in the case of the lambda reducer, concurrent evaluation of parts of an expression requires concurrent access to the environment in which it is to be evaluated. For a sufficiently parallel expression, such accesses could saturate the bandwidth of the memory unit(s) in which the environment is held. By contrast, a combinator expression is complete in itself, and arguments are distributed within a function by pointer replication and graph rearrangement. This is an inherently more decentralised strategy, which would be an advantage on a highly parallel system.

5 CONCLUSIONS

An experimental investigation of the relative efficiencies of a combinator and a lambda reducer has been described. The costs of a computation are counted in terms of accesses to data structures whose size is potentially unbounded, since the cost of such accesses is the only factor that must increase as larger computations are undertaken. Three reducers were written, a combinator reducer (CR), a normal order lambda reducer (NLR) and an applicative order lambda reducer (ALR), and their performance was measured. Some discussion of possible optimisations to each reducer, the effects of compilation, and the suitability of the reducers for parallel processors, has been presented.

The CR outperforms the NLR by a factor between 3 and 5 (taking store accesses as the main indicator), and the ALR by a factor between 1.5 and 3. Furthermore, the easiest hardware support to provide for either system ould be a fast stack cache, and this would not only be cheaper for the CR (smaller stack) but also would give the CR an even greater advantage (since it ˍpends a greater proportion of its time in stack access).

However, it is possible that some seemingly unimportant design decision has substantially affected the results, and a factor of 3 to 5 might be swamped by such effects. For instance there seems some reason to believe that the exact extent and method of optimisation in the combinator code (eg introduction of S', B', C') causes major effects on the runtime of programs. This subject would benefit greatly from a totally independent investigation, preferably by someone determined to prove that lambda expressions were 'better', and with experience of writing optimal lambda interpreters.

Perhaps the most important conclusion is that a combinatorial implementation of a functional language is at least competitive with the better known alternatives, and thus deserves serious consideration by those implementing reduction architectures.

APPENDIX

This appendix shows code for the main evaluation loop of each reducer. It is presented so that the major design decisions made in constructing these loops may be seen. The reducers were both written in BCPL.

Combinator reducer

```
Reduce( Exp ) = VALOF
$(
  LET Result = 0
  Push( CurrentExp )  || Save
  Push( 0 )          || Dummy return address
  NewFrame()         || New stack frame
                     || Uses one push to save
                     || old stack depth.

|| Main loop
  EnStack() REPEATWHILE Perform()
  Result := UnStack()

|| Tidy up
  UnFrame()                || Uses one pop.
  Pop()                    || Dummy return address
  CurrentExp := Pop()      || Restore
  RESULTIS Result
$)

AND Perform() = VALOF
|| This is just a switch on the operator.
$(
  IF CurrentExp=SOp THEN RESULTIS DoS()
  IF CurrentExp=KOp THEN RESULTIS DoK()
    :
    :
  IF CurrentExp=YOp THEN RESULTIS DoY()
  RESULTIS CurrentExp  || Default
$)

AND EnStack() BE
$(
  WHILE IsApplication( CurrentExp ) DO
  $(  Push( CurrentExp )
    CurrentExp := Head( CurrentExp )
$)$)

AND UnStack() = VALOF
$(
  WHILE StackDepth() > 0 DO
    CurrentExp := Pop()
  RESULTIS CurrentExp
$)
```

Lambda reducer

```
LET Reduce( Exp, Env ) = VALOF
$(
  LET Result = 0
  Push( CurrentExp )    || Save
  Push( CurrentEnv )
  Push( 0 )             || Dummy return address

|| Main reduction
  Result := CheapReduce( Exp, Env )

  Pop()                 || Dummy return address
  CurrentEnv := Pop()   || Restore
  CurrentExp := Pop()
  RESULTIS Result
$)

AND CheapReduce( Exp, Env ) = VALOF
$(
  CurrentExp = Exp
  CurrentEnv = Env

  TEST Atom( CurrentExp ) THEN
    TEST IsIdentifier( CurrentExp ) THEN
       RESULTIS LookUp( CurrentExp, CurrentEnv )
    ELSE
      RESULTIS CurrentExp
  ELSE
  $(
    LET Fun = Head( CurrentExp )
    SWITCHON Fun INTO
    $(
      CASE SuspendOp:        || Suspension
      $(
        LET Arg = Tail( CurrentExp )
        RESULTIS Reduce( Head( CurrentExp ),
                         Tail( CurrentExp ) )
      $)

      CASE FunArgOp: RESULTIS CurrentExp

      CASE LamOp: RESULTIS    || Lambda expression
        Cons( FunArgOp,
              Cons( CurrentEnv,
                    Tail( CurrentExp ) ) )

      DEFAULT: RESULTIS       || Application
        Apply( Reduce( Fun, CurrentEnv),
               Cons( SuspendOp,  || Form suspension
                     Cons( Tail( CurrentExp ),
                           CurrentEnv ) ) )
    $)
  $)
$)

AND Apply( Fun, Arg ) = VALOF
$(
  TEST Atom( Fun ) THEN    || Must be built in op.
    RESULTIS Operate( Fun, Arg )
  ELSE
    TEST Head( Fun ) = FunArgOp THEN
    $(
      LET Closure = Tail( Fun )
      LET Env = Head( Closure )
      LET Value = Tail( Closure )
      LET Param = Head( Value )
      LET Body = Tail( Value )
      LET NewEnv = Bind( Env, Param, Arg )
      RESULTIS CheapReduce( Body, NewEnv )
    $)
    ELSE
      Error()
$)

AND Operate( Fun, Arg ) = VALOF
$(
  IF Fun = IfOp THEN RESULTIS DoIf( Arg )
    :
    :
  IF Fun = DivideOp THEN RESULTIS DoDivide( Arg )

|| Default
  RESULTIS Cons( Fun, Reduce( Arg, GlobalEnv )
$)

AND Bind( Env, Param, Arg ) =
    Cons( Env, Cons( Param, Arg ) )

AND LookUp( Ide, Env ) = VALOF
$(
  LET Temp = Env
  UNTIL Temp = Nil DO
  $(
    LET EnvItem = Tail( Temp )
    IF Head( EnvItem ) = Ide THEN
    $(
      LET Value = Tail( EnvItem )
      Push( Temp )
      Value := Reduce( Value, GlobalEnv )
      Temp := Pop()
      SetTail( Tail( Temp ), Value )
      RESULTIS Value
    $)
    RESULTIS Ide         || Default
$)
```

ACKNOWLEDGEMENTS

Part of this work was carried out while in receipt of a SERC grant, and part with the support of Beale Electronic Systems Ltd. The initial motivation for the investigation was provided by Dr Arthur Norman, while many of the ideas clarified in discussion with John Hughes and Gerry Sussman, and discussions with Nicholas Beale were very useful during preparation of the paper itself.

BIBLIOGRAPHY

[Be82a] CRS/1 specification. Available from Beale Electronic Systems Ltd, Wraysbury, UK.

[Be82b] Beale NCL. Compiling ADL/1 to combinators. Beale Electronic Systems Ltd Tech. Rep. TR8202.

[Ch41] Church A. The calculi of lambda conversion. Annals of Mathematical Studies. Princeton University Press, 1941.

[Cu58] Curry & Feys. Combinatory Logic Vol I. North Holland, 1958.

[Cl80] Clarke T, Gladstone P, Maclean C and Norman A. SKIM, the SKI reduction machine. Proc. 1980 LISP conference, pp128-135.

[Di81] Discepolo AM, and Keaton-Williams JP. A Practical Verification System. ACM SEN, July 1981, Vol 6 No 3 pp50-54.

[Hu80] Hughes RJM. The design and implementation of an applicative language. Cambridge University Computer Science Diploma Project, 1980.

[Hu82] Hughes RJM. Super-combinators. Proc. 1982 Symposium on LISP and Functional Programming.

[Jo81] Jonhsson T. Code generation for lazy evaluation. Chalmers University of Sweden, Nov 1981.

[Pe80] Peyton Jones SL. A comparison of the relative efficiencies of the combinator and lambda calculus. Cambridge University Computer Science Diploma Project, 1980. (Revised as Beale Electronic Systems Technical Report TR8201)

[St79a] Stallman R. Phantom stacks. MIT AI Memo 556, 1979.

[St79b] Steele GL, and Sussman GJ. Dream of a lifetime. MIT AI Memo 527, 1979.

[St77a] Steele GL. Lambda, the ultimate goto. MIT AI Memo 443, 1977.

[St77b] Stoy JE. Denotational semantics. MIT Press, 1977.

[Su81] Sussman GJ, Holloway J, Steele GL and Bell A. Scheme-79 - Lisp on a chip. IEEE Computer July 1981, Vol 14 No 7, pp10-21.

[Tu79a] Turner D. A new implementation technique for applicative languages. Software practice and experience 1979, Vol 9, pp31-44.

[Tu79b] Turner D. Another algorithm for bracket abstraction. Journal of Symbolic Logic, Jun 1979, Vol 44, No 2.

[Wa82] Wand M. Semantics directed machine architecture. Proc. 1982 ACM POPL, pp234-241.