An Alternative Scheme for Evaluating Combinator Expressions

MASATO TAKEICHI*

Turner shows how combinators can be used for implementing applicative languages. In his method, a combinator expression is represented by a graph with the nodes comprising functions and their arguments. Application of a function to an argument causes graph reduction which corresponds to the beta-reduction of lambda calculus. Graph reduction is performed in a way such that the node representing a functional application is overwritten by its result. Another scheme for combinator expression evaluation is proposed by Jones and Muchnick. Although their evaluator is a fixed-program and would have some advantages over Turner's graph reduction scheme, it seems unusual in dealing with higher order functions.

In this paper we describe an alternative scheme for evaluating combinator expressions. The evaluator is almost a fixed-program and is easily extended to include new combinators. It deals with higher order functions consistently as Turner's evaluator does. That is, the proposed scheme shares both advantages of Turner's graph reduction and of a fixed-program. And it is most attractive in implementing the evaluator on conventional hardware. An experimental evaluator is also presented.

1. Introduction

In [9] Turner describes an implementation technique for applicative languages which is based on combinator calculus [3]. Function definition in an applicative language is translated into a lambda expression which is then transformed into a combinator expression with no bound variables. Function application in the form of combinator expression is reduced to a simpler one by an interpreter for combinator reduction. In his method, the combinator expression is stored as a graph with the nodes representing functional applications and the interpreter performs graph reduction on this structure. He has used this technique to implement the applicative language SASL [8]. A similar evaluator for combinator expressions has been microprogrammed to gain in execution speed of combinator reduction [2].

The method presented by Turner has become of major interest in normal order evaluation of lambda expressions [7]. In particular, it is most attractive for dealing with higher order functions which convey generalized forms of similar algorithms [1].

Another approach to evaluating combinator expressions is presented in [6]. Unlike the graph reduction scheme, the proposed fixed-program evaluator does not change the code itself. Although the advantages of the fixed-program over the graph reduction scheme are stated there, it is not appropriate for dealing with higher order functions which are to be partially evaluated with an insufficient number of arguments to yield another functions.

The purpose of this paper is to show an alternative scheme, called "*Graph Copying Scheme*", for evaluating combinator expressions. The new scheme would be most appreciated for its simplicity in implementing evaluators on conventional hardware. Algorithms of translating lambda expressions into combinators are not of our concern. Combinators appeared in combinator expressions may be freely chosen; we can apply the scheme to *super-combinators* [4] as well as standard combinators considered in [6, 9].

We consider the evaluation order of functions such as normal and applicative ones as a mathematical issue, and the parameter mechanism such as call-by-name and call-by-value as language issue. When expressions are to be evaluated is an implementation issue. Evaluation of combinator expressions is usually implemented in a lazy, actually fully lazy way (See Section 3), and so is our evaluator. Such evaluation scheme realizes normal order evaluation with call-by-name semantics. In order to avoid confusion, we do not use the terms "normal order evaluation", and "call-by-name mechanism" when we discuss combinator expression evaluation.

2. Combinator Expression

It is well known in the theory of lambda calculus that variables in lambda expressions are unnecessary if a few functions called combinators which embody certain common patterns of application are introduced. Combinators **S** and **K** defined as

*Department of Computer Science, The University of Electro-Communications.

S $f\ g\ x = f\ x(g\ x)$.

K $x\ y = x$

are adequate for eliminating variables from any lambda expression. By convention we denote functional application by juxtaposition and assume that it associates to the left. Turner uses additional combinators for practical reasons [9]:

I $x = x$

B $f\ g\ x = f\ (g\ x)$

C $f\ g\ x = f\ x\ g$

Moreover, combinators for conditional expressions, recursive applications, and arithmetic operations are also introduced:

IF $e\ t\ f = t$ if e is true, f otherwise

Y $f = f\ (\mathbf{Y}\ f)$

$+: x\ y = x+y$, etc.,

$=: x\ y =$ true if x equals y, false otherwise, etc.

A combinator expression is simply an applicative expression, i.e., a constant or an applicative form of a function and an argument, with the restriction that its constituents must be applicative expressions. We assume that globally defined functions and combinators are treated as constants in an expression.

Although the above combinators are considered standard, any closed function, i.e., function without free variables can be taken as a combinator. Hughes [4] proposes a new compilation algorithm for translating any lambda expression into a combinator expression which comprises constants and specifically chosen combinators called super-combinators for the original lambda expression.

Example: Factorial function

Function definition

fac n=**if** n=0 **then** 1 **else** n**fac* (n−1)

Lambda expression

fac=λn. **IF** (=: n 0) 1 (*: n (*fac* (−: n 1)))

Combinator expression using the standard combinators

fac=**S(C(B IF** (=: 0))1) (**S***: (**B** *fac* (**C** −: 1)))

Combinator expression using a super-combinator *fac*:1

fac=*fac*:1

where

fac:1 n=**IF** (=: n 0) 1 (*: n(*fac* (−: n 1)))

The last definition shows that super-combinator *fac*:1 is identical to function *fac* to be defined because *fac* has no free variables. The super-combinator is most useful in compiling functions with many variables, which will be demonstrated in Section 6.

3. Evaluation by Graph Reduction

Turner shows an implementation method for combinator expression evaluation based on graph reduction. As mentioned earlier, the combinator expression is represented by a tree-like structure. The interpreter uses the left ancestor stack (LAS) in order to keep the pointers to nodes involved in combinator reduction. The stack initially contains the pointer to the expression to be evaluated, and the function field of the node pointed to by the top of the stack is pushed down onto the stack in turn until a combinator appears at the top of the stack. Then the appropriate reduction rule for the combinator is applied as shown in Fig. 1.

After the combinator reduction step is performed, stacking operation is resumed. In graph reduction, the graph is transformed according to the combinator reduction rules. Any node representing an expression might be shared by pointers from several nodes for achieving fully lazy evaluation of the first expression. Fully lazy evaluation means that every expression is evaluated at most once [4]. For the sake of shared nodes in the graph, the result of combinator reduction must be left in the root node of the original expression. Hence the node to which the reduction rule is applied is necessarily overwritten with the result, and the indirection node is required when reducing such combinators as **K** and arithmetic operators.

4. Evaluation by Graph Copying

The key to the new "graph copying" scheme relies on the fact that the expression graph can be simplified by copying its arcs and that the node pointed to from multiple nodes in the graph should appear only at reduction of specific combinators. For example, the reduction of

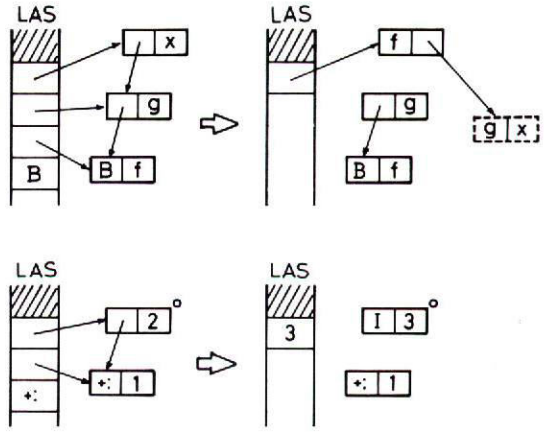

Fig. 1 Reduction of combinators (Graph Reduction).
The box depicted by dotted lines show a newly generated node, and the box marked ∘ becomes an indirection node.

combinator **S** with f, g, and x yields a graph representing $(f\,x\,(g\,x))$ in which both occurrences of x refer to the same expression. This corresponds to the shared node described in the last section.

The evaluator by graph copying uses an argument stack (ARGS) to keep the arguments themselves, not the pointers to them, for combinator reduction. The mechanism of pushing arguments onto the stack and dispatching combinators for reduction follows that of the graph reduction scheme. Since the argument stack does not keep pointers to nodes, rewriting the node is impossible. Instead of rewriting, we make a copy of the expression graph according to the need. Fig. 2 shows the case of reducing combinator **B** by graph copying.

A problem arises along with the reduction of **S**. Although a straightforward application of the above method could implement the copy rule of call-by-name parameters, each occurrence of the same expression must be evaluated independently in the evaluator. As described above, for fully lazy evaluation two occurrences of the expression x should be identical after the reduction in that if either one of these happens to be evaluated to yield a (possibly functional) value, the other should become the value. In the practical point of view, the property of full laziness is most important. Full laziness in graph copying evaluation can be implemented using a special combinator THUNK after the analogy of an established technique for compiling call-by-name parameters in procedural languages [5]. Fig. 3 shows the effect of reducing **S** in the graph reduction and the graph copying schemes. Note that the node with THUNK as its function field is shared by two pointers.

The reduction rule for THUNK is:

THUNK (**eval* e)=e'
: with rewriting the node to yield (THUNK (**deliver* e')) where e' is the result of evaluating e.

THUNK (**deliver* e)=e

where **eval* and **deliver* are distinguished constants.

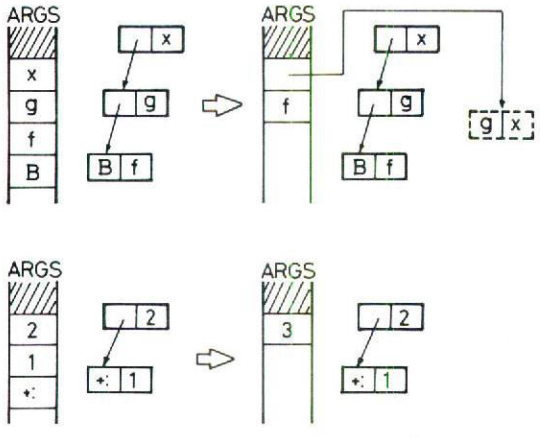

Fig. 2 Reduction of combinators (Graph Copying).
The box depicted by dotted lines shows a newly generated node. Three nodes of the original graph become useless, because f, g, and x are copied onto the stack and the created node.

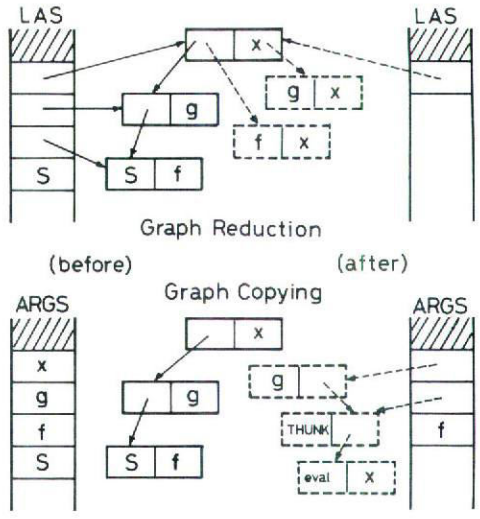

Fig. 3 The reduction of combinator S.
The dotted lines show the state after the reduction.

The effect of applying the reduction rule for THUNK is shown in Fig. 4. A similar mechanism is suggested in [6] with reference to call-by-need evaluation.

The extension of the above method to other combinators presents no major problems. Assume that the reduction rule for combinator **Q** is given as

$$\mathbf{Q}\ x1\ x2\ \ldots\ xn \Rightarrow \mathrm{F}(x1, x2, \ldots, xn),$$

where $\mathrm{F}(x1, x2, \ldots, xn)$ is an applicative expression comprising variables $x1, x2, \ldots, xn$, and constants. Recall that we take free variables, functions and combinators as constants. When reducing **Q**, THUNKs are necessary only for arguments which correspond to variables with multiple occurrences in F.

In general, graph copying reduction for combinator **Q** proceeds as follows:

Let Xt be the set of variables which appear more than once in F.

1) Make thunks with **eval* indicator each for x in Xt.
2) Fill the thunks for xi in Xt with i-th argument on the stack.

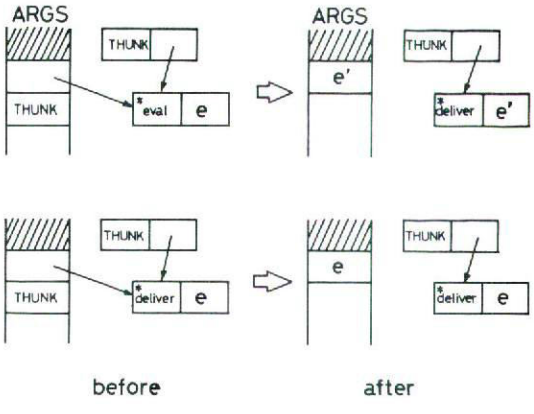

Fig. 4 The reduction of combinator THUNK.
e' is the result obtained by evaluation of e.

3) Construct a graph representing applicative form for F comprising function and argument parts, with replacing *xj* not *Xt* by *j*-th argument on the stack and *xi* in *Xt* by corresponding thunk.

Now that we have had the algorithm for combinator reduction, the evaluation algorithm should be specified:

```
[Graph Copying Evaluator]
 1:
  while (top of ARGS points to a cell of application)
  do
      Discard the top of ARGS;
      Push argument and function parts of the cell in
    this order;
  od;
  if (top of ARGS is a combinator) then
    if(number of arguments on ARGS are sufficient to
      reduce the combinator) then
      Reduce it using as many arguments on ARGS;
      Put the result on ARGS in place of them;
      goto 1;
    else
      Construct a graph representing the applicative
      form for partially evaluated function;
      return the result as value of the expression;
    fi;
  fi;
  if (top of ARGS is data) then return it fi;
  if (top of ARGS is THUNK) then
    if  (it has *eval indicator) then
      Evaluate the expression in the THUNK;
      Rewrite the THUNK with *deliver indicator and
      the result;
      Put the result on ARGS in place of the THUNK;
      goto 1;
    else
      Get the value from the THUNK;
      Put it on ARGS in place of the THUNK;
      goto 1;
    fi;
  fi;
```

It should be noted that the evaluator is called recursively in evaluating THUNKs and special combinators such as **IF**, +:, etc.

We make a few remarks on different evaluation schemes proposed so far before going into the actual implementation of the graph copying scheme.

5. Discussion

Rewriting the node to which the reduction rule is applied ensures the property of full laziness in the graph reduction scheme. If we consider combinators as machine instructions, and super-combinators as microprogrammed instructions, graph reduction evaluation looks like self-reorganizing execution of machine-coded programs. From a methodological point of view, such programs should be avoided at least on conventional hardware. This holds also for a similar scheme used in SKIM [2], where the technique of reversing pointers is applied in place of the left ancestor stack. The fixed-program scheme presented in [6] relies on compiling combinator expressions to yield stack machine code, and an algorithm should be required for extending this method to dealing with super-combinators. Although generated code remains as it is in this evaluator, the stack used in executing the code contains both atomic values and locations in the code. This would not be convenient for saving partially evaluated functions for later use. Such treatment of the code turns out to be an obstacle in using higher order functions. Higher order functions, especially in their curried form, are used to describe abstract algorithms which can be instantiated to obtain concrete ones by supplying some of their arguments. Famous '*map*' function exemplifies such a usage of higher order functions:

map f x: yields a list obtained by applying *f* to the components of list *x*.

If we have function '*square*', we can make a function (*map square*) which yields a list of squares of the list given as the argument. Such instantiation can be possible even for functions over non-integers. The method in [6] does not keep (*map square*) as the same form as '*map*', and special treatment will be required for using it as a function.

The graph copying scheme and the graph reduction scheme share a desirable property for higher order functions that the result of evaluation is either an atomic value or a graph representing a combinator expression. Although both schemes are similar at first sight, the graph copying scheme has a practical advantage over the graph reduction in that it can be implemented efficiently on conventional hardware.

As to storage allocation strategies in graph copying, there is much room for choice. We might use *chunks*, i.e., contiguous memory words, of arbitrary sizes for storing applicative form with many arguments in place of pairs of function and argument fields (Fig. 5).

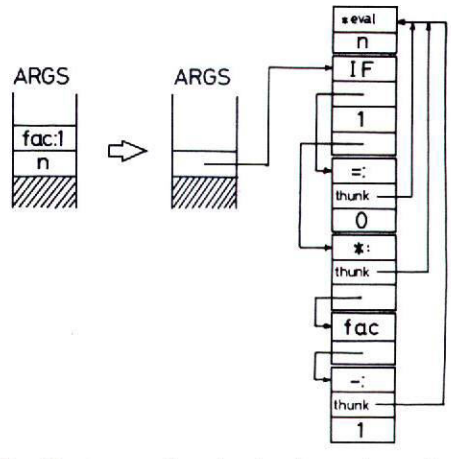

Fig. 5 Possible storage allocation for the graph copying scheme. ARGS grows upward.

It should be noted, of course, that the size of the chunk is kept either in itself or with the pointer to it. An informal algorithm for the evaluator using chunks follows:

```
[Graph Copying Evaluator Using Chunks]
 1:
  while (top of ARGS points to a chunk of application) do
    Discard the top of ARGS;
  (*) Move the chunk for the applicative form onto ARGS;
  od;
  if (top of ARGS is a combinator) then
    if (number of arguments on ARGS are sufficient to reduce the combinator) then
    (*) Reduce it using as many arguments on ARGS;
        Put the result on ARGS in place of them;
        goto 1;
    else
    (*) Dump the arguments on ARGS as a chunk of applicative form representing a function;
    return the result as the value of the expression;
    fi;
  fi;
  --- following cases are the same as in Section 4.
```

The lines marked (*) offer the advantage of using data transfer instructions such as '*load multiple words into registers*', '*store registers into multiple words*', or '*move multiple words from memory to memory*'. It is impossible in the graph reduction scheme. The second of these, i. e., the reduction step, would require explanation. As shown in Fig. 5, the result of reduction can be represented by chunks allocated in contiguous memory words; in the case of Fig. 5, 6 chunks are allocated in 17 contiguous words. In this way, we can allocate as many words as required for reduction all at once, and then copy the template of the resulting graph into these words just allocated using data transfer instructions. After copying, we have only to adjust pointers to the chunks of the graph being created, and fill argument positions with arguments on the stack.

In order to make comparison of two schemes, we now return to our old representation of the graph for expressions.

It should be noted that the number of nodes required seems to be approximately the same in graph reduction and in graph copying. Table 1 shows the number of new nodes required for reducing the standard combinators. In fact, it turns out that the numbers of **S** reductions and **C** reductions taken in evaluating functions are approximately the same. In case of the factorial function in Section 2, both reductions occur $2*n+1$ times each when evaluating (*fac n*).

Table 1 The number of new nodes required for reduction.

	Graph Reduction	Graph Copying
S	2	3
K	0	0
I	0	0
B	1	1
C	1	0

Since the evaluation speed depends heavily on the technique and on the hardware used in actual implementations, precise figures seem meaningless. Our experiments using Lisp functions show that both evaluators run at a comparable speed. The graph copying evaluator using chunks gains much in efficiency and runs fast. However, we have no idea to compare it to any implementation of the graph reduction evaluator. We believe that it is difficult to compare implementation techniques with the full use of specific strategies for memory allocation and specific instructions.

To summarize, the graph copying scheme is most attractive because of its simplicity and its versatility in implementing the evaluator on conventional hardware.

6. Implementation

In order to save the trouble of machine dependency in illustrating the graph copying evaluator, we show an experimental evaluator written in Franz Lisp, a dialect of Lisp on the VAX UNIX 4.1 bsd. Although various optimization techniques could be applied to gain speed, we show a simple version in Fig. 6. The evaluator comprises a few Lisp functions:

eval:	dispatches a reducible expression to the combinator reducer
make-thunk:	makes a thunk
THUNK	reduces THUNK combinator
IF	implements the rule for **IF**
=: ...	evaluates predicate=, ...
+: ...	performs arithmetic operation +, ...

The key idea behind this implementation lies in using the *fexpr* (or *nlambda*) Lisp function for passing the argument stack '*args*:' unevaluated. Since the argument stack is realized by a list, new cells are required not only for graph copying, but for stack elements. In this implementation, for ease of treating graphs as Lisp expressions the combinator expression is represented as an S-expression which differs from the description in Section 4. For example, while an applicative form (**B** *f g x*) is so far represented a (((**B**. *f*).*g*).*x*) in Lisp's dot notation, it is now represented by a list (*B f g x*) which is to be treated as function application of *B* with arguments *f*, *g*, and *x* by the Lisp evaluator. This leads to another storage allocation strategy mentioned in Section 5.

Functions for reducing combinators such as **S**, **K**, **I**, **B**, and **C** can be defined by a code generator '*define-combinator*' which translates reduction rules into Lisp functions (Fig. 7).

Two compilers '*sc-compile*' and '*skibc-compile*' are also provided in order to compile lambda expressions into combinator expressions using super-combinators and using the standard combinators, respectively. They pass the reduction rule to '*define-combinator*' for defin-

```
(def gceval
  (nlambda (args:)
    (eval: (car args:))))

(def eval:
  (lambda (args:)
    (prog (top-args:)
     L    (setq top-args: (car args:))
          (cond ((numberp top-args:)
                 (return top-args:))
                ((hunkp top-args:) (return top-args:))
                ((atom top-args:)
                 (return (apply top-args: args:)))
                (t (setq args:
                         (append top-args:
                                 (cdr args:)))
                   (go L))))))

(def make-thunk:
  (lambda (e)
    (cond ((numberp e) e)
          (t
           (list 'THUNK
                 (cons '*eval e))))))

(def THUNK
  (nlambda (args:)
    (let ((e (cadr args:)))
         (cond ((= (car e) '*eval)
                (rplacd e (eval: (ncons (cdr e))))
                (rplaca e '*deliver)
                (rplaca (cdr args:) (cdr e))
                (eval: (cdr args:)))
               (t (rplaca (cdr args:) (cdr e))
                  (eval: (cdr args:))))))))

(def IF
  (nlambda (args:)
    (cond ((< (length args:) 4) args:)
          ((eval: (ncons (cadr args:)))
           (eval: (cons (caddr args:) (cddddr args:))))
          (t (eval: (cdddr args:))))))

(def =:
  (nlambda (args:)
    (cond ((< (length args:) 3) args:)
          (t
           (= (eval: (ncons (cadr args:)))
              (eval: (ncons (caddr args:))))))))

(def +:
  (nlambda (args:)
    (cond ((< (length args:) 3) args:)
          (t
           (+ (eval: (ncons (cadr args:)))
              (eval: (ncons (caddr args:))))))))
```

Fig. 6 An experimental evaluator (Graph Copying Scheme).

ing Lisp functions which perform combinator reduction. Generated Lisp functions can be compiled by the Lisp compiler *liszt* to yield faster combinator reducers.

Fig. 8 shows how the higher order function is dealt with by this evaluator. Although the higher order function would be most useful in generating functions for handling structured data such as lists, there is no space here to discuss it.

The effect of THUNK is demonstrated in Fig. 9, where the trace function of Franz Lisp is used.

We now turn to discuss about execution time for evaluation. Although comparison between the supercombinator and the standard combinators is not our main concern, we show the time required for evaluating (*fac* 10) in Table 2. This agrees with the result by Hughes [4].

The usefulness of the thunk can be realized by evaluating a highly recursive function:

$f\ x\ y\ z =$

if $z > y$ **then**

$f\ (f\ y\ z\ (x\text{-}1))\ (f\ z\ x\ (y\text{-}1))\ (f\ x\ y\ (z\text{-}1))$

else y

Fig. 10 shows the result of translating f given as a lambda expression into a combinator expression with supercombinators f, $f{:}1$, $f{:}2$, and $f{:}3$ generated. Lisp func-

```
-> (define-combinator (S f g x) (f x (g x))) ;defines S
S
-> (pp S) ;prints combinator reducer S

(def S
  (nlambda (args:)
    (cond ((< (length args:) 4) args:)
          (t
           (let ((f (cadr args:))
                 (g (caddr args:))
                 (x (make-thunk: (cadddr args:)))
                 (args:: (cddddr args:)))
                (eval:
                 (cons f
                       (cons x
                             (cons (cons g (ncons x))
                                   args::)))))))))

t
-> (define-combinator (K x y) x) ;defines K
K
-> (pp K)

(def K
  (nlambda (args:)
    (cond ((< (length args:) 3) args:)
          (t
           (let ((x (cadr args:)) (args:: (cdddr args:)))
                (eval: (cons x args::)))))))
```

Fig. 7 Definition of combinators.

```
-> (define-combinator (W f x) (f x x)) ;define W
W
-> (pp W) ;prints combinator reducer W

(def W
  (nlambda (args:)
    (cond ((< (length args:) 3) args:)
          (t
           (let ((f (cadr args:))
                 (x (make-thunk: (caddr args:)))
                 (args:: (cdddr args:)))
                (eval:
                 (cons f (cons x (cons x args::)))))))))

t
-> (define square: (W *:)) ;defines square function
(square: = (W *:))
-> (pp square:)

(def square:
  (nlambda (args:)
    (let ((args:: (cdr args:)))
         (eval:
          (cons 'W (cons '*: args::))))))
```

Fig. 8 An example of higher order function.

```
-> (trace W *: THUNK) ;traces how THUNK works
(W *: THUNK)
-> (gceval(square: (+: 3 4))) ;evaluates square of (3+4)
1 <Enter> W (W *: (+: 3 4))
|1 <Enter> *: (*: (THUNK (*eval +: 3 4)) (THUNK (*eval +: 3 4)))
| 1 <Enter> THUNK (THUNK (*eval +: 3 4))
| 1 <EXIT>  THUNK  7
| 1 <Enter> THUNK (THUNK (*deliver . 7))
| 1 <EXIT>  THUNK  7
|1 <EXIT>  *:  49
1 <EXIT>  W  49
49
```

Fig. 9 The effect of the thunk.

Table 2 Comparison of the super-combinator and **SKIBC** combinators (unit is 1/60 sec).

	Super-combinators	SKIBC combinators
Using compiled reducer	8	15
Using Lisp interpreter	49	93

Table 3 The effect of the thunk in evaluating a highly recursive function (unit is 1/60 sec).

	Using compiled code		Using Lisp interpreter	
	Graph Copying	Lisp	Graph Copying	Lisp
(*f* 0 2 4)	8	1	76	3
(*f* 0 3 6)	29	14	179	36
(*f* 0 4 8)	59	249	338	676
(*f* 0 5 10)	86	-	526	-
(*f* 0 6 12)	123	-	-	-

```
sc-source:
(f
 (lambda (x y z)
         (IF (>: z y)
             (f (f y z (-: x 1))
                (f z x (-: y 1))
                (f x y (-: z 1)))
             y))
 )

level:3
reduction-rule:
((f:3 p00007 p00006 p00010 p00008 p00005 p00004 z)
 =>
 (IF (>: z p00004)
     (f (p00005 z p00006)
        (f z p00007 p00008)
        (p00010 (-: z 1)))
     p00004))

level:2
reduction-rule:
((f:2 p00011 p00009 y) => (p00011 (p00009 y) (-: y 1) (f y) y))

level:1
reduction-rule:
((f:1 x) => (f:2 (f:3 x (-: x 1)) (f x)))

level:0
reduction-rule:
((f) => f:1)
```

Fig. 10 Translating a lambda expression into a combinator expression.

tions *f*, *f*:1, *f*:2, and *f*:3, can be compiled by the Lisp compiler.

Table 3 summarizes execution time required to evaluate

$$(f\ 0\ n\ 2{*}n) \text{ for several values of } n$$

by our graph copying evaluator using super-combinators and by a Lisp function which is a straight translation of the above definition. We conclude from this experiment that it would be intolerable without thunk.

7. Conclusion

It has been shown that the graph copying scheme for evaluating combinator expressions is superior to others as described in Section 5. If we apply this scheme to applicative languages with user-definable data structures such as record with arbitrary number of fields, a new strategy for storage allocation using chunks for applicative forms as well as data structures deserves attention. We must leave a detailed discussion about this for a future study.

References

1. Burge, W. H. *Recursive Programming Techniques*, Addison-Wesley, 1975.
2. Clarke, T. J. W., Gladstone, P. J. S., MacLean, C. D. and Norman. A. C. SKIM—The S, K, I Reduction Machine, *Conf. Rec. of the 1980 LISP Conf.*, 128–135.
3. Hindley, J. R., Lercher, B. and Seldin, J. P. *Introduction to Combinatory Logic*, Cambridge Univ. Press, 1972.
4. Hughes, R. J. M. Super-combinators, *Conf. Rec. of the 1982 ACM Symp. on LISP and Functional Programming*, 1–10.
5. Ingerman, P. Z. Thunks, *Comm. ACM* **4** (1961), 55–58.
6. Jones, Neil D. and Muchnick, Steven S. A. Fixed-Program Machine for Combinator Expression Evaluation, *Conf. Rec. of the 1982 ACM Symp. on LISP and Functional Programming*, 11-20.
7. Peyton Jones, S. L. An Investigation of the Relative Efficiencies of Combinators and Lambda Expressions, *ibid.*, 150–158.
8. Turner, D. A. *SASL Language Manual*, Univ. of St. Andrews, 1976.
9. Turner, D. A. A New Implementation Technique for Applicative Languages, *Softw. Pract. Exper.* **9** (1979), 39–49.

(Received November 17, 1983; revised May 7, 1984)