Title	Relative efficiencies of reduction machines
Sub Title	
Author	相場, 亮(Aiba, Akira)
Publisher	慶應義塾大学理工学部
Publication year	1986
Jtitle	Keio Science and Technology Reports Vol.39, No.1 (1986. 5) ,p.1- 24
JaLC DOI	
Abstract	Relative efficiency of the reduction of lambda-terms and that of combinatory expressions are compared when they represent recursive programs belonging to the linear branched recursion schema in Strong [7]. The terms and the expressions are extended versions by introducing constants and some primitive operations on them, and all of them are reduced by the left-most reduction strategy. Reduction costs of the lambda-calculus are obtained theoretically, and those of the combinatory logic are obtained by experiments. The conclusions: (1) reduction with subexpression sharing is more efficient than that without sharing, and (2) the reduction in the lambda-calculus is more efficient than that in the combinatory logic if the number of parameters of the program is larger, and the converse is true when its recursion depth is larger.
Notes	
Genre	Departmental Bulletin Paper
URL	https://koara.lib.keio.ac.jp/xoonips/modules/xoonips/detail.php?koara_id=KO50001004-00390001-0001

KEIO SCIENCE AND TECHNOLOGY REPORTS
VOL. 39, NO. 1, pp. 1-24, 1986

RELATIVE EFFICIENCIES OF REDUCTION MACHINES

by

Akira AIBA

Department of Mathematics
Faculty of Science and Technology, KEIO University,
Hiyosi 3-14-1, Kohoku-ku, Yokohama 223, JAPAN

(*Received January 28, 1986*)

ABSTRACT

Relative efficiency of the reduction of lambda-terms and that of combinatory expressions are compared when they represent recursive programs belonging to the linear branched recursion schema in Strong [7]. The terms and the expressions are extended versions by introducing constants and some primitive operations on them, and all of them are reduced by the left-most reduction strategy. Reduction costs of the lambda-calculus are obtained theoretically, and those of the combinatory logic are obtained by experiments. The conclusions: (1) reduction with subexpression sharing is more efficient than that without sharing, and (2) the reduction in the lambda-calculus is more efficient than that in the combinatory logic if the number of parameters of the program is larger, and the converse is true when its recursion depth is larger.

1. Introduction.

Since the presentation of Backus [1] in 1978, functional programming languages have been concerned as a new paradigm. Theoretical base of such languages is the lambda-calculus or the combinatory-logic. In recent years, some attempts to construct machines which execute programs written in such languages were made. Some of these machines are called reduction machines because they execute reductions in the lambda-calculus or those in the combinatory-logic. The relative efficiency of those machines should be compared in various ways before they are constructed, and it has been discussed somehow. In Burton [3], a new method to translate applicative programs to combinatory expressions is proposed. By this method, translated combinatory expressions are small in size. In Jones [5], some actual programs such as factorial function are translated into lambda-term and combinatory expressions, and the efficiencies of reductions are compared. According to Ida [4], the efficiency of the reduction of the lambda-terms is comparable

to that of the combinatory expressions in terms of the order of the computational complexity.

In this paper, the relative efficiency of the reduction of lambda-terms is compared strictly with that of combionatory expressions when they represent recursive programs of the following form:

$$f(x_1, x_2, \ldots, x_m) = \text{if } (p(x_1, x_2, \ldots, x_m),$$
$$q(x_1, x_2, \ldots, x_m),$$
$$h(x_1, x_2, \ldots, x_m,$$
$$f(k_1(x_1), k_2(x_2), \ldots, k_m(x_m))).$$

This schema is selected because

1) it is relatively simple, and

2) many programs are of this schema, since p, q, h, and k_i $(i=1, 2, \ldots, m)$ may be complicated functions made by the function composition. Note that the structure of the combinatory expression depends on the form of the program.

Our comparisons assume the followings.

i) lambda-terms and combinatory expressions are extended by adding constants and primitive-operations on constants. These extended reduction systems satisfy the Church-Rosser property. When the reduction system satisfies this property, each term has the unique result of the reduction (called the normal form) if it exists. This property of such kind of reduction systems is studied in Klop [6].

ii) Lambda-terms and Combinatory-expressions are reduced by the normal order reduction strategy.

iii) A combinatory expression is obtained from a lambda-term by bracket abstraction algorithm in Turner [8]. Note that when the efficiency of the reduction of combinatory expressions is concerned, this translation process is not included in the cost of the reduction.

Reduction costs in lambda-calculus are calculated theoretically as in the Appendix, and those in combinatory logic are given by experiments.

Main results are as follows. (1) Let m denotes the number of parameters of a program, and n denotes the depth of the recursion to obtain its result. The beta-reductions are performed $m(n+1)$ times, and the reductions of combinators are performed $(m^2n+9mn)/2+3m+n$ times. (2) To compare further these expres-

sions, assume that terms are represented as binary-trees or graphs the number of changed pointers and the number of used cells are chosen as the common measures. By these measures the reduction in the lambda-calculus is more efficient than the reduction in the combinatory-logic when the number of parameters of the program is larger, and the converse is true when its recursion depth is larger. The main result is summarized in the Table 1, and Table 3.

In Section 2, reduction systems, lambda-calculus and combinatory logic of their extended version, are defined for comparison. In Section 3, recursive programs to be translated, and the cost of the reduction are defined. In Section 4, the cost of reduction of each reduction system is presented. In Section 5, these costs are compared.

2. Reduction Systems

Reduction systems for comparisons are defined in this section. Definitions of pure lambda-calculus and pure combinatory logic can be referred in Barendregt [2]. Reduction systems in this section are extended ones by adding constants, functions for them, and programs.

Definition 2-1 (Lambda Terms):

1) Lambda-terms are either constants, variables, functions denoted by $\lambda x.M$, where M is a lambda-term and x is a variable, or sequences of lambda-terms $((\ldots(M_1M_2)\ldots)M_n)$. There may be constants true and false in the set of lambda-terms.

2) A lambda-program is an equation $f=M$, where f is a constant and M is a lambda-term.

3) In a lambda-term $\lambda x.M$, occurrences of x in M is called free in M if M does not contain λx.

Definition 2-2 (Reduction Rules for lambda-terms):

Lambda-terms are reduced successively into lambda-terms, in which no reducible expressions called redexes occur, by applying following reduction rules.

1) beta-rule

$((\lambda x.M)N) \rightarrow M[x:=N]$ where $M[x:=N]$ denotes the result of substituting N into every free occurrence of x in M.

2) delta-rule

$c\ M_1M_2\ldots M_n \rightarrow N$ where c is a constant called a delta-constant, M_1, $M_2\ldots$, and M_n are lambda-terms, and N is the result of the reduction of c over

$M_1, M_2 \ldots$, and M_n. The reduction of c is given a priori.

3) f-rule

$f \rightarrow M$ where f is a constant and M is a lambda-term.

Remark:

When a lambda-term N is reduced by applying (1) beta-rule, (2) delta-rule, and (3) f-rule $f \rightarrow M$ then the reduction sequence of N is called *'reduction of N under a program $f=M$'*. If a program '$f=M$' is obvious then '$f=M$' is often omitted.

Notation:

A lambda-term $((\ldots(M_1M_2)\ldots)M_n)$ and $\lambda x_1.\ (\lambda x_2.\ldots.(\lambda x_n.M)\ldots)$ is often abbreviated as $M_1M_2\ldots M_n$ and $\lambda x_1.\ \lambda x_2.\ldots.\lambda x_n.M$, respectively.

Definition 2-3 (Combinatory Expressions):

1) Combinators are lambda-terms with no occurrences of free variables.

$\boldsymbol{S}=\lambda x.\ \lambda y.\ \lambda z.x\ z\ (y\ z)$
$\boldsymbol{K}=\lambda x.\ \lambda y.x$
$\boldsymbol{I}=\lambda x.x$
$\boldsymbol{B}=\lambda x.\ \lambda y.\ \lambda z.x\ (y\ z)$
$\boldsymbol{C}=\lambda x.\ \lambda y.\ \lambda z.\ (x\ z)\ y$
$\boldsymbol{S'}=\lambda x.\ \lambda y.\ \lambda z.\ \lambda u.x\ (y\ u)(z\ u)$
$\boldsymbol{B'}=\lambda x.\ \lambda y.\ \lambda z.\ \lambda u.x\ y\ (z\ u)$
$\boldsymbol{C'}=\lambda x.\ \lambda y.\ \lambda z.\ \lambda u.x\ (y\ u)\ z$

2) Combinatory expressions are either constants, variables, combinators, or sequences of combinatory expressions $((\ldots(M_1M_2)\ldots)M_n)$. There can be constants true and false in the set of combinatory expressions.

3) A combinatory-program is an equation denoted as $f=M$, where f is a constant and M is a combinatory expression.

Notation:

A combinatory expression $((\ldots(M_1M_2)\ldots)M_n)$ is often abbreviated as $M_1M_2\ldots M_n$.

Definition 2-4 (Reduction rules for combinatory expressions):

Combinatory expressions are reduced sucessively into combinatory expressions, in which no reducible expressions called reduction rules.

(1) Combinator-rules

$\boldsymbol{S}\ X\ Y\ Z \rightarrow X\ Z\ (Y\ Z)$
$\boldsymbol{K}\ X\ Y \rightarrow X$
$\boldsymbol{I}\ X \rightarrow X$
$\boldsymbol{B}\ X\ Y\ Z \rightarrow X\ (Y\ Z)$
$\boldsymbol{C}\ X\ Y\ Z \rightarrow (X\ Z)\ Y$
$\boldsymbol{S'}\ X\ Y\ Z\ U \rightarrow X\ (Y\ U)(Z\ U)$
$\boldsymbol{B'}\ X\ Y\ Z\ U \rightarrow X\ Y\ (Z\ U)$
$\boldsymbol{C'}\ X\ Y\ Z\ U \rightarrow X\ (Y\ U)\ Z$

where X, Y, Z, and U are combinatory expressions.

(2) delta-rule

$c\ M_1 M_2 \ldots M_n \rightarrow N$ where c is a constant called a delta-constant, $M_1, M_2, \ldots,$ and M_n are combinatory expressions. N is the result of the reduction of c over $M_1, M_2, \ldots,$ and M_n. The reduction of c is given a priori.

(3) f-rule

$f \rightarrow M$ where f is a constant and M is a combinatory expression.

Remark:

When a combinatory expression N is reduced by applying (1) combinator-rules, (2) delta-rule, and (3) f-rule $f \rightarrow M$, then the reduction sequence of N is called *'reduction of N under a program $f=M$'*. If a program '$f=M$' is obvious then '$f=M$' is often omitted.

A combinatory expression is obtained from a lambda-term by bracket abstraction algorithm in Turner [8]. Let $\lambda x.M$ be a lambda-term to be translated. By the following algorithm, occurrences of x in M are eliminated.

If M is a constant, a variable, or a combinator, then

$\boldsymbol{I}$ if M is x, and
$\boldsymbol{K}\ x$ otherwise.

If M is an expression ($M1$ $M2$), then

$\boldsymbol{S}\ M1'\ M2'$

where $M1'$ and $M2'$ are combinatory expressions which are obtained from $M1$ and $M2$ by eliminating x according to this algorithm.

Then, obtained combinatory expressions are optimized by the following rules.

Definition 2-5 (Optimization rules):

(1) Optimization rules 1

(i) $\boldsymbol{S}\ (\boldsymbol{K}\ X)(\boldsymbol{K}\ Y) \Rightarrow \boldsymbol{K}\ (X\ Y)$

(ii) $\boldsymbol{S}\ (\boldsymbol{K}\ X)\ \boldsymbol{I} \Rightarrow X$

(2) Optimization rules 2

(i) $\boldsymbol{S}\ (\boldsymbol{K}\ X)\ Y \Rightarrow \boldsymbol{B}\ X\ Y$

(ii) $\boldsymbol{S}\ X\ (\boldsymbol{K}\ Y) \Rightarrow \boldsymbol{C}\ X\ Y$

(3) Optimization rules 3

(i) $\boldsymbol{S}\ (\boldsymbol{B}\ X\ Y)\ Z \Rightarrow \boldsymbol{S'}\ X\ Y\ Z$

(ii) $\boldsymbol{B}\ (X\ Y)\ Z \Rightarrow \boldsymbol{B'}\ X\ Y\ Z$

(iii) $\boldsymbol{C}\ (\boldsymbol{B}\ X\ Y)\ Z \Rightarrow \boldsymbol{C'}\ X\ Y\ Z$

In the above rules (1), (2) and (3) X, Y, and Z denote arbitrary combinatory expressions. Optimization rules 2 can be applied if no rules in (1) can be applied, and optimization rules 3 can be applied if no rules is (2) can be applied.

To satisfy the Church-Rosser property, delta-rules for the lambda-calculus and combinatory-logic should be restricted as in Klop [6]. That is, they should be left-linear and non-ambiguous. For instance, cond is introduced into the reduction systems to represent the conditional branching schema if. It should always have three arguments to satisfy that property. When a reduction system satisfies this property, each term in that system has its unique result of reductions if exists.

3. Recursive programs and the cost of the reduction

In this section, recursive program schema and recursive programs considered in this paper are defined first, then assumptions of our study are set up. In the last part of this section, the cost of reduction is defined. The recursive program schema which is defined in this section is linear in the sense of Strong [7]. But we define the desired program schema in a different way, because ours is more restricted than its linear recursion schema.

Definition 3-1 (Term):

Let $\boldsymbol{Bs}=\{b_1,b_2,\ldots,b_n\}$ be a set of base-operation symbols, each element b_i is associated with non-negative integer called the arity of b_i. Elements of $\boldsymbol{Bs}$ with arity 0 are called constants. Let $\boldsymbol{P}=\{p_1,p_2,\ldots,p_m\}$ be a set of predicate symbols, each element p_i is associated with its arity. Let $\boldsymbol{A}=\{a_1,a_2,\ldots,a_l\}$ be a set of argument symbols, and $\boldsymbol{F}=\{f_1,f_2,\ldots,f_k\}$ be a set of function symbols, each element f_i is associated with its arity. The set of terms over $\boldsymbol{Bs}$, $\boldsymbol{P}$, $\boldsymbol{A}$, and $\boldsymbol{F}$ is defined inductively as follows.

(1) An argument symbol is a term,

(2) If $t_1,t_2,\ldots,t_n$ are terms and b is a base operation symbol with arity n,

then $b(t_1,t_2,\ldots,t_n)$ is a term.

(3) If $t_1,t_2,\ldots,t_m$ are terms and p is a base operation symbol with arity m, then $p(t_1,t_2,\ldots,t_m)$ is a term.

(4) If $t_1,t_2,\ldots,t_k$ are terms and f is a base operation symbol with arity k, then $f(t_1,t_2,\ldots,t_k)$ is a term.

Definition 3-2 (Recursive program schema):

A recursive program schema ***Sr*** is a set of equations

$$f_i(x_1,x_2,\ldots,x_n)=t_i,\ i=1,2,\ldots,p$$

where $x_1,x_2,\ldots,x_n$ are elements of ***A***, and t_i is a term over ***Bs***, ***P***, ***A***, and ***F***.

Definition 3-3 (Interpretation):

The interpretation ***It*** for a recursive program schema ***Sr*** is defined as follows.

Let ***D*** be a non-empty set.

(1) Each base-operation symbol b_i with arity n is associated with a mapping $\boldsymbol{D}^n \rightarrow \boldsymbol{D}$. Remark that each constant c is associated with an element d of ***D***.

(2) Each predicate symbol p_i with arity m is associated with a mapping $\boldsymbol{D}^m \rightarrow \{\boldsymbol{t}, \boldsymbol{f}\}$, where ***t*** and ***f*** are elements of ***D***.

Definition 3-4 (Recursive program):

A recursive program is a tuple $\langle \boldsymbol{Sr}, \boldsymbol{It} \rangle$ where ***Sr*** is a recursive program schema and ***It*** is its interpretation.

A recursive program which belongs to the following form is analyzed in this paper.

$$\begin{aligned} f(x_1,\ldots,x_m)=\text{if } (&p(x_1,\ldots,x_m),\\ &q(x_1,\ldots,x_m),\\ &h(x,\ldots,x_m,\\ &f(k_1(x_1),\ldots,k_m(x_m)))) \end{aligned} \tag{1}$$

where p is a predicate symbol, q, h, $k_1\ldots$ and k_m are base-operation symbols. This schema is linear in the sense of Strong [7]. The interpretation of the program

(1) is as follows. The set $\boldsymbol{D}$ is arbitrary. Let $a_1, a_2, \ldots, a_m$ be values of argument symbols $x_1, x_2, \ldots, x_m$ of (1), and $p(k_1^n(a_1), k_2^n(a), \ldots, k_m^n(a_m))$ is simplified to $\boldsymbol{t}$, where $k_i^n(a_i)$ denotes the result of the simplification of a term which is obtained by n times applications of k_i to a_i $(i=1,2,\ldots,m)$.

There are following alternatives of the reduction machine on which recursive programs of the form of (1) are reduced.

i) **Machine language:** There are two kinds of machine languages. One consists of lambda-terms, and the other consists of combinatory expressions. Both reduction systems are extended by introducing constants and primitive operation symbols.

ii) **Sub-expression sharing:** On both lambda-terms and combinatory expressions there is a choice. By one method, sub-expressions are shared by their appropriate occurrences, and by the other they are not shared. In the lambda-calculus, sub-expressions which are obtained by the substitution to occurrences of the bound variable can be shared among the occurrences of such sub-expressions. In the combinatory logic, the plural occurrences of sub-expressions in the contractum of reduction rules, such as the third argument of the combinator S, can be shared by their occurrences as in the reduction machine in Turner [9].

As for reduction machines for combinators, there is a choice on combinators which are used to represent the program. One machine uses combinators $\boldsymbol{S}$, $\boldsymbol{K}$, $\boldsymbol{I}$, $\boldsymbol{B}$, and $\boldsymbol{C}$, and the other machine uses combinators $\boldsymbol{S}$, $\boldsymbol{K}$, $\boldsymbol{I}$, $\boldsymbol{B}$, $\boldsymbol{C}$, $\boldsymbol{S'}$, $\boldsymbol{B'}$, and $\boldsymbol{C'}$. It is clear that the reduction using the latter set is more efficient than that using the former set as in Turner [8].

The relative efficiency is compared among the following three types of reduction machines.

1) **Type I:** Reduction machine for the lambda-calculus in which sub-expressions are not shared.

2) **Type II:** One for lambda-terms in which sub-expressions are shared by their appropriate occurrences.

3) **Type III:** One for combinators $\boldsymbol{S}$, $\boldsymbol{K}$, $\boldsymbol{I}$, $\boldsymbol{B}$, $\boldsymbol{C}$, $\boldsymbol{S'}$, $\boldsymbol{B'}$, and $\boldsymbol{C'}$ in which sub-expressions are shared on reducing combinators $\boldsymbol{S}$ and $\boldsymbol{S'}$.

Consider the program of the form of (1) with two parameters as an example.

$$f(x_1, x_2) = \text{if } (p(x_1, x_2),\ q(x_1, x_2),$$
$$h(x_1, x_2,\ f(k_1(x_1),\ k_2(x_2)))). \tag{2}$$

The program (2) is translated into the following lambda-program.

$$f=\lambda x_1.\ \lambda x_2.\ (\text{cond}\ (p\ x_1\ x_2)\ (q\ x_1\ x_2)$$

$$(h\ x_1\ x_2\ (f(k_1\ x_1)\ (k_2\ x)))). \tag{3}$$

By translating a lambda-term of the right-hand side of the lambda-program (2) to a combinatory expression, the following combinatory-program is obtained by the algorithm in Turner [8].

$$f=\boldsymbol{S'}\ \boldsymbol{S}$$

$$(\boldsymbol{S'}\ \boldsymbol{S}\ (\boldsymbol{B'}\ \boldsymbol{B}\ \text{cond}\ p)\ q)$$

$$(\boldsymbol{S'}\ \boldsymbol{S}\ h\ (\boldsymbol{C'}\ \boldsymbol{B}\ (\boldsymbol{B}\ f\ k_1)\ k_2)). \tag{4}$$

The cost of the reduction is given on the following items.

i) **The number of *f*-reduction:** The total number of reduction of variables which represent function symbols.

ii) **The number of primitive-operations:** The total number of reduction of primitive-operation symbols p, q, h, k_i $(i=1,2,\ldots,m)$ to obtain the normal form.

iii **The number of conditionals:** The total number of reduction of the primitive-operation symbol cond to obtain the normal form.

iv) **The number of proper reductions:** In the reduction machine for lambda-terms, this means the number of applications of the beta-rule to obtain the normal form, and in the reduction machine for combinators, the total number of combinator reductions to obtain the normal form.

The reduction cost is obtained when a lambda-term or a combinatory expression $(f\ a_1\ a_2 \ldots a_m)$ is reduced into a normal form in each type of machine under a program (3) and (4).

4. Costs for reductions

In this section, reduction costs are presented. In Table 1, the number of reductions are shown for each rule, where n denotes the number of applications of $k_1, k_2, \ldots$ and k_m to arguments $a_1, a_2, \ldots, a_m$ of the program before the evaluation of p turns *true* where *true* is a constant in each reduction system. Note that n is also called *'the recursion depth'*, and m denotes the number of parameters of the recursive program.

Table-1: The Cost of the Reduction

Reduction System	λ-calculus		combinatory logic
Type	Type-I (Not-shared)	Type-II (Shared)	Type-III (Shared)
Reduction of f	$n+1$	$n+1$	$n+1$
Primitive Operation	$m \cdot n^2+(m+2)n+2$	$(m+2)n+2$	$(m+2)n+2$
Conditionals	$n+1$	$n+1$	$n+1$
β-reductions	$m(n+1)$	$m(n+1)$	
Combinators			$1/2 \cdot m(mn+9n)$ $+3m+n$
S			$3n+2$
B			$(m+2)n+1$
S'			$3(m-1)n+2(m-1)$
B'			$(m-1)n+(m-1)$
C'			$1/2 \cdot m(m-1)n$

m: The number of parameters of program.
n: The depth of the recursion.

As an example, the reduction sequence of (3) in Type I and Type II reduction machines are listed in the following. Suppose that a_1 and a_2 are arguments for this term, and $(p\ k_1(a_1)\ k_2(a_2))$ is reduced into true, where $k_1(a_1)$ and $k_2(a_2)$ denote normal forms of $(k_1\ a_1)$ and $(k_2\ a_2)$, respectively. Thus, $n=2$ and $m=1$ in this example.

(1) Reduction sequence in Type I.

$$(f\ a_1\ a_2)$$

by f-reduction

$$\rightarrow (\lambda x_1.\ \lambda x_2.\ (\text{cond}\ (p\ x_1\ x_2)\ (q\ x_1\ x_2)$$
$$(h\ x_1\ x_2\ (f\ (k_1\ x_1)\ (k_2\ x_2))))))\ a_1\ a_2,$$

by beta-reduction

$$\rightarrow (\lambda x_2.\ (\text{cond}\ (p\ a_1\ x_2)\ (q\ a_1\ x_2)$$
$$(h\ a_1\ x_2\ (f\ (k_1\ a_1)\ (k_2\ x_2))))))\ a_2,$$

by beta-reduction

$$\rightarrow (\text{cond } (p\ a_1\ a_2)\ (q\ a_1\ a_2)\ (h\ a_1\ a_2\ (f\ (k_1\ a_1)\ (k_2\ a_2)))),$$

by p-reduction

$$\rightarrow (\text{cond false } (q\ a_1\ a_2)\ (h\ a_1\ a_2\ (f\ (k_1\ a_1)\ (k_2\ a_2)))),$$

by cond-reduction

$$\rightarrow (h\ a_1\ a_2\ (f\ (k_1\ a_1)\ (k_2\ a_2))),$$

by f-reduction

$$\rightarrow (h\ a_1\ a_2\ ((\lambda x_1.\ \lambda x_2.\ (\text{cond } (p\ x_1\ x_2)\ (q\ x_1\ x_2)\ (h\ x_1\ x_2\ (f\ (k_1\ x_1)\ (k_2\ x_2))))\ (k_1\ a_1)\ (k_2\ a_2))),$$

by beta-reduction

$$\rightarrow (h\ a_1\ a_2\ ((\lambda x_2.\ (\text{cond } (p\ (k_1\ a_1)\ x_2)\ (q\ (k_1\ a_1)\ x_2)\ (h\ (k_1\ a_1)\ x_2\ (f\ (k_1\ (k_1\ a_1))\ (k_2\ x_2)))))\ (k_2\ a_2))),$$

by beta-reduction

$\rightarrow$ $(h\ a_1\ a_2$

$(\text{cond}\ (p\ (k_1\ a_1)\ (k_2\ a_2))\ (q\ (k_1\ a_1)\ (k_2\ a_2))$

$(h\ (k_1\ a_1)\ (k_2\ a_2)$

$(f\ (k_1\ (k_1\ a_1))\ (k_2\ (k_2\ a_2)))))),$

by k_1-reduction

$\rightarrow$ $(h\ a_1\ a_2$

$(\text{cond}\ (p\ k_1\ (a_1)\ (k_1\ a_2))\ (q\ (k_1\ a_1)\ (k_2\ a_2))$

$(h\ (k_1\ a_1)\ (k_2\ a_2)$

$(f\ (k_1\ (k_1\ a_1))\ (k_2\ (k_2\ a_2)))))),$

where $k_1(a_1)$ denotes the normal form of $(k_1\ a_1)$,

by k_2-reduction

$\rightarrow$ $(h\ a_1\ a_2$

$(\text{cond}\ (p\ k_1(a_1)\ k_2(a_2))\ (q\ (k_1\ a_1)\ (k_2\ a_2))$

$(h\ (k_1\ a_1)\ (k_2\ a_2)$

$(f\ (k_1\ (k_1\ a_1))\ (k_2\ (k_2\ a_2)))))),$

where $k_2(a_2)$ denotes the normal form of $(k_2\ a_2)$,

by p-reduction

$\rightarrow$ $(h\ a_1\ a_2$

$(\text{cond true}\ (q\ (k_1\ a_1)\ (k_2\ a_2))$

$(h\ (k_1\ a_1)\ (k_2\ a_2)$

$(f\ (k_1\ (k_1\ a_1))\ (k_2\ (k_2\ a_2)))))),$

by cond-reduction

$$\rightarrow (h\ a_1\ a_2\ (q\ (k_1\ a_1)\ (k_2\ a_2))),$$

by k_1-reduction

$$\rightarrow (h\ a_1\ a_2\ (q\ k_1(a_1)\ (k_2\ a_2)))$$

where $k_1(a_1)$ denotes the normal form of $(k_1\ a_1)$,

by k_2-reduction

$$\rightarrow (h\ a_1\ a_2\ (q\ k_1(a_1)\ k_2(a_2))),$$

where $k_2(a_2)$ denotes the normal form of $(k_2\ a_2)$,

by q-reduction

$$\rightarrow (h\ a_1\ a_2\ q(k_1(a_1),\ k_2(a_2))),$$

where $q(k_1(a_1),\ k_2(a_2))$ denotes the normal form of $(q\ k_1(a_1),\ k_2(a_2))$,

by h-reduction

$$\rightarrow h(a_1,\ a_2,\ q(k_1(a_1),\ k_2(a_2))),$$

which is the normal form of the given term.

In this reduction sequence, the numbers of each reductions are

f-reduction:	2 times
primitive-operations:	8 times
cond-reductions:	2 times
beta-reduction:	4 times

(2) Reduction sequence in Type II.

$$(f\ a_1\ a_2)$$

by f-reduction

$$\rightarrow (\lambda x_1.\ \lambda x_2.\ (\text{cond}\ (p\ x_1\ x_2)\ (q\ x_1\ x_2)$$

$$(h\ x_1\ x_2\ (f\ (k_1\ x_1)\ (k_2\ x_2)))))\ a_1\ a_2,$$

by beta-reduction

$$\rightarrow (\lambda x_2.\ (\text{cond}\ (p\ a_1\ x_2)\ (q\ a_1\ x_2)$$
$$(h\ a_1\ x_2\ (f\ (k_1\ a_1)\ (k_2\ x_2)))))\ a_2,$$

by beta-reduction

$$\rightarrow (\text{cond}\ (p\ a_1\ a_2)\ (q\ a_1\ a_2)$$
$$(h\ a_1\ a_2\ (f\ (k_1\ a_1)\ (k_2\ a_2)))),$$

by p-reduction

$$\rightarrow (\text{cond false}\ (q\ a_1\ a_2)$$
$$(h\ a_1\ a_2\ (f\ (k_1\ a_2)\ (k_2\ a_2)))),$$

by cond-reduction

$$\rightarrow (h\ a_1\ a_2\ (f\ (k_1\ a_1)\ (k_2\ a_2))),$$

by f-reduction

$$\rightarrow (h\ a_1\ a_2$$
$$((\lambda x_1.\ \lambda x_2.\ (\text{cond}\ (p\ x_1\ x_2)\ (q\ x_1\ x_2)$$
$$(h\ x_1\ x_2\ (f\ (k_1\ x_1)\ (k_2\ x_2))))))$$
$$(k_1\ a_1)\ (k_2\ a_2))),$$

by beta-reduction

$$\rightarrow (h\ a_1\ a_2$$
$$((\lambda x_2.\ (\text{cond}\ (p\ (k_1\ a_1)\ x_2)\ (q\ (k_1\ a_1)\ x_2)$$

$(h\ (k_1\ a_1)\ x_2$

$(f\ (k_1\ (k_1\ a_1))\ (k_2\ x_2)))$

$(k_2\ a_2))),$

(Note that 4 occurrences of the expression $(k_1\ a_1)$ are shared by their occurrences).

by beta-reduction

$\rightarrow (h\ a_1\ a_2$

(cond $(p\ (k_1\ a_1)\ (k_2\ a_2))\ (q\ (k_1\ a_1)\ (k_2\ a_2))$

$(h\ (k_1\ a_1)\ (k_2\ a_2)$

$(f\ (k_1\ (k_1\ a_1))\ (k_2\ (k_2\ a_2)))),$

(Note that 4 occurrences of $(k_2\ a_2)$ are shared by their occurrences).

by k_1-reduction

$\rightarrow (h\ a_1\ a_2$

(cond $(p\ k_1(a_1)\ (k_2\ a_2))\ (q\ k_1(a_1)\ (k_2\ a_2))$

$(h\ k_1(a_1)\ (k_2\ a_2)$

$(f\ (k_1\ k_1(a_1))\ (k_2\ (k_2\ a_2)))),$

by k_2-reduction

$\rightarrow (h\ a_1\ a_2$

(cond $(p\ k_1(a_1)\ k_2(a_2))\ (q\ k_1(a_1)\ k_2(a_2))$

$(h\ k_1(a_1)\ k_2(a_2)$

$(f\ (k_1\ k_1(a_1))\ (k_2\ k_2(a_2)))))),$

by p-reduction

$\rightarrow$ $(h\ a_1\ a_2$

(cond true $(q\ k_1(a_1)\ k_2(a_2))$

$(h\ k_1(a_1)\ k_2(a_2)$

$(f\ (k_1\ k_1(a))\ (k_2\ k_2(a_2))))$,

by cond-reduction

$\rightarrow$ $(h\ a_1\ a_2\ (q\ k_1(a_1)\ k_2(a_2)))$,

by q-reduction

$\rightarrow$ $(h\ a_1\ a_2\ q(k_1(a_1),\ k_2(a_2)))$,

where $q(k_1(a_1),\ k_2(a_2))$ denotes the normal form of $(q\ k_1(a_1)\ k_2(a_2))$,

by h-reduction

$\rightarrow$ $h(a_1,\ a_2,\ q(k_1(a_1),\ k_2(a_2)))$,

which is the normal form of given term.

In this reductionce, the numbers of reductions are

f-reduction:	2 times
primitive-operations:	6 times
cond-reductions:	2 times
beta-reduction:	4 times

Numbers of reductions of Type I and Type II in Table I are investigated in the appendix.

5. Comparisons

By observing Table 1, the followings are clear.

1) The reduction machines which share sub-expressions by their occurrences require the same number of steps of the reduction of f as the reduction machine

without sharing.

2) The number of primitive operations, and that of cond are independent of the reduction systems.

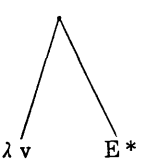
Thus, the difference between the reduction machine for the lambda-calculus and that for the combinatory-logic is in the number of their proper reductions, beta-reductions and reductions of combinators. However, these number can not be compared directly, because one beta-reduction may be different, in the amount of time or space, from one combinator reduction.

Suppose that lambda-terms and combinatory expressions are represented as binary trees in each reduction machine as in Fig. 1, the cost of each reduction can be compared by the following two criteria.

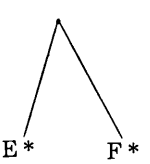
1) The total number of changed pointers in the tree representing a term to perform proper reductions.

2) The total number of used cells to perform proper reductions.

(a) v → v (where v is a variable, a constant, or a base operation symbol)

(b) λ v. E → 

(where E* is the tree representation of E)

(c) (E F) → 

(where E* and F* are tree representations of E and F, respectively)

Fig.-1: Tree representation

By introducing the above two criteria, the beta-reduction can be compared to the combinator reduction. Note that shared sub-expressions are represented by pointers to common nodes.

In Type I reduction machine, there are $(6m+5)$ terminal nodes and $(6m+4)$ non-terminal nodes when the tree represents a lambda-term with m parameters. In the body of the tree, there are $(5m+5)$ terminal nodes and $(5m+4)$ non-terminal nodes. When a beta-reduction is performed on the tree, a sub-tree representing a corresponding argument is substituted to terminal nodes representing

4 occurrences of a bound variable in the body of the tree. Thus, 3 times of cells which construct a sub-tree representing a corresponding argument are used by one beta-reduction. The relation between this number of cells and n is as follows.

n	the number of cells
0	1
1	3
2	5
3	7

Thus, the total number of used cells to perform $m(n+1)$ times beta-reductions are: $3(n+1)^2m=3mn^2+6mn+3m$

On the other hand, 4 pointers are changed to perform substitutions for one beta-reduction, and $2n$ pointers are changed to construct an argument. The relation between this number of changed pointers and n is as follows.

n	the number of pointers
0	0
1	2
2	4
3	6

Note that there are 4 occurrences of each argument. Thus, the total number of changed pointers to perform $m(n+1)$ times beta-reductions are: $3n(n+1)m+4m(n+1)=3mn^2+7mn+4m$

In Type II reduction machine, 4 occurrences of each bound variable are shared by their occurrences, and sub-expression which is substituted to a bound variable are also shared by their occurrences. However, a sub-tree which represent an argument is copied once when a beta-reduction is performed. Thus, the total number of used cells to perform $m(n+1)$ beta-reductions are: $(n+1)^2m=mn^2+2mn+m$

The total number of changed pointers to perform $m(n+1)$ beta-reductions are: $n(n+1)m+4m(n+1)=mn^2+5mn+4m$

In Type III reduction machine, the number of cells and the number of changed pointers to perform a reduction are dependent on the combinator which are used in that reduction. In Table 2, the number of used cells and the number of changed pointers are listed.

By these arguments, the total number of used cells and the total number of changed pointers to obtain the normal form of the term representing a program which is belong to the linear branched recursion schema in each reduction machine can be summarized as in the Table 3. Note that n and m in these tables are same as those for Table 1.

Table-2: Number of used cells and changed pointers of Combinators

	Number of cells	Number of pointers
S	2	6
B	1	4
S'	3	8
B'	2	6
C'	2	5

Table-3: Number of used cells and changed pointers

Reduction System	λ-calculus		combinatory logic
Type	Type-I (Not-shared)	Type-II (Shared)	Type-III (Shared)
Number of cells	$3mn^2+6mn+3m$	$mn^2+2mn+m$	$m^2\cdot n+11mn+9m$ $-5n-2$
Number of pointers	$3mn^2+7mn+4m$	$mn^2+5mn+4m$	$5/2\cdot m^2\cdot n+59/2\cdot mn$ $+26m+8n$

m: The number of parameters of program.
n: The depth of the recursion.

By observing Table 3, the followings are concluded. (1) The reduction with sharing is more efficient than the reduction without sharing. (2) The reduction in the lambda-calculus is more efficient than that in the combinatory logic when they reduce terms representing our recursive programs if m is larger, and converse is true if n is larger.

REFERENCES

[1] Backus, J., 1978, Can programming be liberated from the von Neumann style?: A functional style and its algebra of programs, *Comm. ACM* **21**(8). pp. 613-641.

[2] Barendregt, H.P., 1981, *The Lambda Calculus: Its Syntax and Semantics*, North-Holland, Amsterdam.

[3] Burton, F.W., 1982, A linear space translation of functional programs to Turner combinators, *Inf. Process. Lett.* **14**(**5**). pp. 201-204.

[4] Ida, T., and Konagaya, A., 1984, Comparison of closure reduction and combinatory reduction schemes, *private communication.*

[5] Jones, S.L.P., 1982, An Investigation of the Relative Efficiencies of Combinators and Lambda Expressions, *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pp. 150-158.

[6] Klop, J.W., 1981, *Combinatory Reduction System*, Thesis for the Ph. D., University of Utrecht.

[7] Strong Jr., H.R., 1971, Translating Recursion Equation into Flow Charts, *J. Comput. Syst. Sci.*, **5**, pp. 254-285.

[8] Turner, D.A., 1978, Another Algorithm for Bracket Abstraction, *J. Symbolic Logic*, **44**, pp. 267-270.

[9] Turner, D.A., 1979, A New Implementation Technique for Applicative Languages, *Softw. Pract. Exper.*, **9**, pp. 31-49.

[10] Wadsworth, C.P., 1971, *Semantics and pragmatics of the lambda calculus*, Thesis for Ph. D., University of Oxford.

Appendix: Investigation of the costs of the reductions in Type I and Type II reduction machines

Conditions of the investigation are as follows.

1) Consider the lambda-terms which represent programs of the form (1) in Section 3,

2) a lambda-term $(f\ a_1 a_2 \ldots a_m)$ is given as the initial term of the reduction sequence under a program

$$f=\lambda x_1.\ \lambda x_2.\ldots.\lambda x_m.\ (\text{cond}\ (p\ x_1\ x_2 \ldots x_m)$$
$$(q\ x_1\ x_2 \ldots x_m)$$
$$(h\ x_1\ x_2 \ldots x_m$$
$$(f\ (k_1\ x_1)\ (k_2\ x_2) \ldots (k_m\ x_m))))$$

whose form for $m=2$ is given in (3) in Section 3.

3) the number of parameters is m, and

4) $(p\ x_1\ x_2 \ldots x_m)$ turns true after n times applications of k_1, k_2, ... and k_m to arguments a_1, a_2, ..., and a_m of the term, respectively.

For Type I and Type II reduction machine, each expression in Table 1 is obtained by the following statements.

Statement 1

In Type I reduction machine, $n+1$ reductions of f are required to obtain the normal form.

Proof

Obvious from the condition 2) and 4).

Statement 2

In Type I reduction machine, $m(n+1)$ beta-reductions are required to obtain the normal form.

Proof

For each reduction of f, m beta-reductions are performed by the condition 2). By Statement 1, reduction of f is performed $n+1$ times. Thus, beta-reduction is

performed $m(n+1)$ times.

Statement 3

In Type I reduction machine, $n+1$ reductions of cond are required to obtain the normal form.

Proof

For each reduction of f, the reduction of cond is performed only once to select one branch of it. By Statement 1, reduction of f is performed $n+1$ times. Thus, reduction of cond is performed $n+1$ times. Thus, reduction of cond is performed $n+1$ times.

Lemma 4

In Type I reduction machine, $n+1$ reductions of p is required to obtain the normal form.

Proof

For each reduction of cond, the reduction of p is performed once. By Statement 3, reduction of cond is performed $n+1$ times. Thus, the reduction of p is performed $n+1$ times.

Lemma 5

In Type I reduction machine, only one reduction of q is required to obtain the normal form.

Proof

The reduction of q is performed when $(p\ x_1\ x_2 \ldots x_m)$ turns *true*. Thus, the reduction of q is performed only once.

Lemma 6

In Type I reduction machine, n reductions of h is required to obtain the normal form.

Proof

The reduction of h is performed once when f in arguments of h is reduced. The first reduction of f does not satisfy this condition. Thus, the reduction of h is performed n times.

Lemma 7

In Type I reduction machine, $(n^2+n)m$ reductions of each k_i $(i=1, 2, \ldots, m)$ are required to obtain the normal form.

Proof

Since sub-expressions are not shared in Type I reduction machine, k_i $(i=1,2, \ldots, m)$ which occur in different places should be reduced separately. Once the reduction of f in h is performed, redexes of k_i $(i=1,2, \ldots, m)$ are distributed to arguments for p, q, h, and f. Thus, if such redexes in p are reduced, redexes of k_i $(i=1, 2, \ldots, m)$ which are distributed in other places are not affected. Suppose that k_i $(i=1, 2, \ldots, m)$ is reduced j times when f in h is reduced r times. If f in h is reduced $(r+1)$ times, then k_i $(i=1,2, \ldots, m)$ in p in the lambda-term which is obtained by $(r+1)$-st reduction of f in h should be reduced. Because such k_i-redexes $(i=1,2, \ldots, m)$ have never been touched, there are rm-redexes in such places. But, $(r-1)m$ k_i-reductions $(i=1,2, \ldots, n)$ are not needed to reduce in arguments for q in the lambda-term which are obtained by r-th reduction of f in h because this branch is not selected by the reduction of cond. On the other hand, k_i-redexes $(i=1,2, \ldots, m)$ in arguments for q in the lambda-term which is obtained by $(r+1)$-st reduction of f in h should be reduced. The number of such k_i-redexes $(i=1,2, \ldots, m)$ are rm. After the reduction of q is terminated, that normal form is returned in the argument for h in the lambda-term which is obtained by r-th reduction of f in h. And other arguments for h in such place has k_i-redexes $(i=1,2, \ldots, m)$. The number of such k_i-redexes $(i=1,2, \ldots, m)$ is $(r-1)m$. Thus, k_i $(i=1,2, \ldots, m)$ is reduced $j+2(r+1)m$ times when f in h is reduced $(r+1)$ times. Note that r may varied from 0 to $(n-1)$, and if n is equal to 0 then the reduction of k_i $(i=1,2,\ldots, m)$ is not performed. Therefore, in Type I reduction machine, the reduction of k_i $(i=1,2, \ldots, m)$ is performed $(n^2+n)m$ times.

Statement 8

In Type I reduction machine, $mn^2+(m+2)n+2$ reductions of primitive operations are required to obtain the normal form.

Proof

The number of reductions of primitive operations is the summation of the total number of reductions of p, q, h, and k_i $(i=1,2, \ldots, m)$. The number of reductions of base operations is equal to $(n+1)+1+n+(n^2+n)m$, thus it is equal to $mn^2+(m+2)n+2$.

In Type II reduction machine, sub-expressions are shared. First of all, we investigate that what sub-expressions are shared in the Type II reduction machine to obtain the number of reductions. In the lambda-terms representing our programs of the form (1) in Section 3, there are 4 occurrences of each bound variables x_i.

These variables are shared by their 4 occurrences in Type II reduction machine. Note that only k_i-redexes ($i=1,2,\ldots,m$) have possibilities of putting themselves to such occurrences of bound variables. Thus, the number of reductions of p, q, and h are the same as in Type I reduction machine.

Lemma 9

In Type II reduction machine, mn reductions of k_i ($i=1,2,\ldots,m$) are required to obtain the normal form.

Proof

Suppose that the reduction of k_i ($i=1,2,\ldots,m$) is performed j times if f in h is reduced r times. In Type II reduction machine, once an argument which is substituted to one occurrence of bound variable is reduced, then all occurrences of such an argument in that lambda-term are reduced. Thus, if f in h is reduced $(r+1)$ times, then the number of reductions of k_i ($i=1,2,\ldots,m$) is $j+m$. Note that r may varied from 0 to $(n+1)$, and the number of reductions of k_i ($i=1,2,\ldots,m$) is 0 if n is equal to 0. Therefore, in Type II reduction machine, the reduction of k_i ($i=1,2,\ldots,m$) is performed nm times.

Statement 10

In Type II reduction machine, $(m+2)n+2$ primitive operations are required to obtain the normal form.

Proof

The total number of reductions of primitive operation is the summation of the total number of reductions of p, q, h, and k_i ($i=1,2,\ldots,m$). Thus, the number of base operations is equal to $(n+1)+1+n+mn$. This is equal to $(m+2)n+2$.