数理解析研究所講究録
第547巻 1985年 230-267

Comparison of closure reduction and combinatory reduction schemes

Tetsuo Ida † and Akihiko Konagaya ‡

† Riken: Institute of Physical and Chemical Research

‡ C & C Systems Research Laboratories, NEC Corporation

This work is partly based on the activities of WG. 5 of Fifth Generation Computer Project of ICOT.
The work is also supported by the Grant in Aid of Ministry of Education and Culture, No. 59580027.
Authors' addresses: T. Ida, Institute of Physical and Chemical Research, 2-1, Hirosawa, Wako-shi, 351-01, Japan,
Akihiko Konagaya, C & C Systems Research Laboratories, NEC Corporation 1-1, Miyazaki, 4-chome, Miyamae-ku, kawasaki, Kanagawa, 213, Japan

Abstract

We analyze the efficiencies of closure reduction and combinatory reduction schemes by introducing a labelled tree representing a λ-term. Translation of a λ-term into combinatory terms, i.e. bracket abstraction, can be viewed as attaching labels **S**, **B**, **C**, **K**, **I** to each node. Similarly, a node of a tree representing a λ-term can be labelled depending upon the presence of free variables in the subtrees. Resulting labelled trees which represent a λ-term and the translated combinatory term are made similar, i.e. whose underlying trees are the same. We can then make performance comparisons in terms of the cost involved in traversing the labelled trees by machine models reflecting the essential behaviors of Turner's combinatory reducer and a closure reducer. Our work is an elaboration of Turner's and Peyton Jones'es experiments of combinatory reductions. However, our approach is not to resort to actual runs of programs, but is more theoretical based on abstract machine models working on labelled trees. Our conclusion of the performance comparisons is that a closure reducer is in most cases more efficient than combinatory reducers in terms of storage consumption which is a dominant factor in determining the overall performance of the reducers. Furthermore, we show that the two reducers which seem quite different at first sight is in fact very similar and with small modifications the two schemes become essentially the same.

Keywords and phrases: λ-calculus, combinatory logic, functional programming, reduction machine

CR Categories: C.1.3, D.1.1, F.1.1, F.4.1, F.2

1. Introduction

This paper is an attempt to attain unified understanding of the behaviors of the two known machine-implemented reduction schemes of Λ-calculus; closure reduction and combinatory reduction.

In [10], Turner presented a new technique of implementing functional programs using combinators. Turner's scheme consists in compiling functional programs into a sequence composed of combinators and constants that are represented internally as a graph, and reducing the graph into a normal form by the leftmost reduction strategy. Efficiency comparisons are described in [10] in terms of the number of consumed cells and the number of reduction steps in the graph reduction system and SECD machine [8]. Later Peyton Jones gathered the statistics of timings of various program runs on both the combinatory graph reduction system (to be called *combinatory reducer*) and the λ-calculus reduction system (to be called *λ-reducer*) [9]. In this paper, we investigate the efficiency of the schemes with the view to elaborating the results of the efficiency comparisons made by Turner and Peyton Jones, and further show that both reducers can be made to become the same reducer (which we will call a *labelled tree reducer*) by successive improvements.

Given a functional program, we have three alternatives for its execution following the previous works as above;

(1) to translate it into a λ-term and directly interpret
the λ-term by the λ-reducer,
(This scheme is to be called *closure reduction* for the reasons
to be described in section 2.)

(2) to translate it into combinators, and interpret the combinatory expression
by the combinatory reducer,

(3) to translate it into SECD-like machine code and interpret the code
by that machine.

Measuring the timings of runs on the three abstract machines (simulated by the same real machine) is one way of comparing the efficiencies. However, care has to be

taken lest that any peculiarity of the real machine might squeeze into the performances of the simulated machines. We first ask ourselves whether one machine subsumes the other in some fundamental way.

Therefore we start our investigation by presenting a model of computation at a λ-term level, and elaborate the model into a machine model which is similar to the SECD machine. In this process we attempt to form a clear view of underlying machines for λ-calculus and to give intuitive and convincing arguments to the efficiency comparisons. In this paper we limit our considerations to machine models that are essentially of von Neumann type; it consists of a single processor and random access memory. The processor is to execute the various reduction strategies such as λ-calculus leftmost reduction. The memory is organized as heap and stacks; the heap for storing the tree representation of the terms, and stacks for storing temporary operands and control information. Finally, some examples of comparisons are given.

2. Λ-calculus and machine models

We list preliminary notions below. For full exposition of λ-calculus and combinatory systems, readers are referred to [1] .

- Programs are a sequence of terms†, e.g. $((..(M_1M_2)\ ..\ )M_n)$
- Λ-terms are either constants, variables, or functions denoted as $\lambda x.M$, where M is a term and x is a variable.
- A variable occurrence immediately after λ is called a λ-variable.
- A pair of terms is called an application.
- Function with n arguments $x_1,..,\ x_n$ is represented † as $\lambda x_1..x_n.M$. Assume M does not contain λ, then $x_1,..,x_n$ which appear in M is called "bound" by λ, otherwise "free".

foot note

† We use following short hand notation: $M_1M_2..M_n$ for $((..(M_1M_2)..)M_n)$ and $\lambda x_1..x_n.M$ for $\lambda x_1.(\lambda x_2.(..(\lambda x_n.M)..))$.
We omit 'λ' whenever context permits us to infer.

Machine L0

Λ-terms are reduced successively to a no longer reducible form called a *normal form* by applying following reduction rules:

(1) substitution rule (β-rule):

$(\lambda x.M)a \rightarrow M[x:=a]$ where $M[x:=a]$ is the result of substituting a for every free occurrence of x in M.

(2) δ-rules:

$kM_1M_2..M_n \rightarrow L$ where k is a constant called δ-constant. L is the result of the reduction (δ-reduction) specified by k on M_1, M_2 ,.., M_n. The reduction associated with k is given a priori. δ-rules are built into the system to make certain operations on terms as primitive, e.g. + in + m n → $m+n$. We assume that the binary relation δ induced by δ-rule satisfies Church-Rosser property.

Observations I

- In the realization of β-rule, the literal substitution on von Neumann type computers is expensive, since it entails copying of the term to be substituted.
- Finding every occurrence of variables at the substitution time is also time-consuming.
- The literal substitution may cause variable name clashes.

Therefore, we use a pointer to term a instead of the term itself, and simply establish the correspondence between the value (pointer) and the variable. We delay the actual substitution until the variables are referenced. To enable the delayed substitution, we use an *environment list* consisting of the (dotted) pairs of a variable name and the corresponding value, i.e. substituted term. Therefore, $M[x:=a]$ is actually represented as a pair of the term and the environment, i.e. $【M,\mathcal{E}】$. Here, 【 , 】 is called *closure*, and $\mathcal{E}$ in this case is $((x.a).\mathcal{E}')$ where $\mathcal{E}'$ is the environment before the substitution. The variable name clash can be avoided if a search for a variable in the environment list is made consistently from the head to the tail of the list.

Given term $M_1M_2..M_n$, Machine LO reduces the term by finding the leftmost reducible application (reducible application is called *redex*). It can be shown that the

leftmost reduction strategy is normalizing (i.e. the normal form, if exists, is reached by repeated application of the reduction rules to the leftmost redex) [P. 321, 1] . This normalizing property is one of the reasons why recent functional programming systems [e.g. 5, 1] adopt the leftmost reduction strategy despite the added complexity (and possible degradation of efficiency) compared with the languages with the applicative order reduction strategy. We assume that term $((..(M_1M_2)..)M_n)$ is represented as a *binary tree* (or *tree* for short) (see Fig. 1).

The machine is equipped with a reduction stack and heap storage. The leftmost reduction strategy is implemented on machine L0 by the following algorithm ρ_λ: (Implicit in this algorithm is the presence of control stack which is necessary to implement the recursive algorithm below).

Leftmost λ reduction algorithm ρ_λ

Initial input to ρ_λ is *term*, ε_0 and an empty reduction stack, where *term* is a program to be reduced, and ε_0 is an initial environment.

$\rho_\lambda(M, \varepsilon, stack) \equiv$

let $stack=\langle s_1,..,s_n\rangle$ or $\langle\rangle$ i.e. empty
(1) M is atomic[†1]
  if M is a terminal object[†2]
  then resultis[†3]

$$\begin{cases} M \text{ when } stack \text{ is empty} \\ M\rho_\lambda(s_1,(),newstack()^{\dagger 4})..\rho_\lambda(s_n,(),newstack()) \text{ otherwise} \end{cases}$$

  else if M is δ[†5]
  then resultis

$$\begin{cases} Ms_1s_2..s_n \text{ when } n \text{ is less than the number of the terms} \\ \quad \text{required in this } \delta\text{-reduction} \\ \rho_\lambda(w, \varepsilon, stack') \text{ otherwise} \end{cases}$$

  where w is the result of the δ-reduction, i.e.
  $w = M(s_1,...s_i)$[†6].
  and $stack'$ $(=\langle s_{i+1},..,s_n\rangle)$ is the *stack* after the reduction.
  else let $w = \rho_\lambda(lookup^{\dagger 7}(\varepsilon,M),(),newstack())$
  update[†8] environment ε with $(M.w)$
  resultis $\rho_\lambda(w,\varepsilon,stack)$
(2) M is of the form $\lambda x.body$
  resultis

$$\begin{cases} M \text{ when } stack \text{ is empty} \\ \rho_\lambda(body,((x.s_1).\varepsilon),\langle s_2,s_3,..,s_n\rangle) \text{ otherwise} \end{cases}$$

(3) M is closure
  let $M = \lfloor M',\varepsilon' \rfloor$
  resultis $\rho_\lambda(M',\varepsilon',stack)$
(4) M is M_1M_2 (i.e. application)

resultis $\rho_\lambda(M_1, \mathcal{E}, <\llbracket M_2, \mathcal{E} \rrbracket, s_1, s_2, .., s_n>)$

---

note:

†1 Type atomic is given a priori; number, a sequence of characters, e.g. *abcd*, and structured data, e.g. list are atomic.

†2 An atom is a *terminal object* if it is non-variable and not δ.

†3 "resultis *exp*" means that this algorithm terminates with *exp* as a result.

†4 *newstack*() creates a new stack.

†5 A symbol is δ if it is associated with a reduction rule, i.e. δ-rule.

†6 Functional notation for M indicates that M is an externally given function. The order of reductions of i arguments are left to the semantics of M.

†7 *lookup* searches for the *bounded pair* $(M, |M|)$ in the environment list from left and returns $|M|$ if such a pair exists.

†8 "update environment $\mathcal{E}$ with $(M.w)$" implies the replacement of the bounded pair $(M.|M|)$ (first from left) in $\mathcal{E}$ by $(M.w)$, where $|M|$ is a term formerly bound to M. That is, $\mathcal{E}=(...(M.|M|)....) \rightarrow \mathcal{E}=(...(M.w).....)$

Algorithm ρ_λ shows that the objects it handles are closures rather than λ-terms. Hence, we call this λ-reduction system *closure reduction system* (or *closure reducer* for short). The machine traverses the tree that represents the term leftwards from top to bottom, pushing the closures of the subtrees on the reduction stack (cf. Fig. 2). When a bottom, presumably some δ-constant, is reached, the machine reduces necessary number of operands which are on the reduction stack and the δ-rule is applied.

A note to the implementation of algorithm ρ_λ

(i) For clarity of the presentation, a closure is formed when it is stacked onto the reduction stack.

(ii) We can avoid the unnecessary formation of closures, if we realize a stack that consists of two fields; one is for storing a term, and the other for an environment.

(iii) The parameter binding in β-reduction (cf. case (2) in algorithm ρ_λ) can be extended to allow for the simultaneous binding of variables to the values on the stack. That is, the execution of $\rho_\lambda(\llbracket \lambda\vec{x}_n.M, \mathcal{E} \rrbracket, <s_1, .., s_m>)$ results in

$\rho_\lambda(\llbracket M, ((x_n.s_n)..(x_1.s_1).\mathcal{E}) \rrbracket, <s_{n+1}, .., s_m>)$ when $m \geqq n$, or

$\rho_\lambda(\llbracket \lambda x_{m+1}..x_n.M, ((x_m.s_m)..(x_1.s_1).\mathcal{E}) \rrbracket, <>)$ when $m<n$.

From now on, we assume algorithm ρ_λ is modified as above.

(iv) Turner's system employs a graph to represent a term. A graph is constructed

because a recursion function is realized by a circular pointer to the term defining the function. The graph is converted to a tree by cutting the circular link to the function and by letting the pointer to point to an atom of the function name which is outside the world of our discourse, and hence is regarded as a constant. (cf. Fig. 3)

Observations II

- Forming a closure with the current environment in each push of terms M_n, M_{n-1}, .., M_2 onto the stack may be superfluous since the stacked environment may not be used. Imagine the case that M_i is a constant. Even if the improvements of (i) and (ii) are made, the push of an environment onto the stack is superfluous in the cases that terms do not need an environment.
- Whether a particular term M_i needs the environment for its reduction can be determined beforehand. Simply check the presence of the variable that are free in M_i.

Therefore, we elaborate machine L0 into L1 by introducing a tag in each node.
The types of tags are following:
s to denote that the right and left subtrees need an environment.
b to denote that only the right subtree needs an environment.
c to denote that only the left subtree needs an environment.
See Fig. 4.

Machine L1 and algorithm ρ'_λ

Machine L1 checks the tag stored in each node and the closure is formed on the basis of the comparison. Either the subtree itself or the closure are stacked during the traversal of the tree. The rest of the workings are the same as machine L0. The reduction algorithm ρ_λ is modified to process a tagged tree, The modified algorithm is called ρ'_λ. It is further developed into algorithm $\rho_{C\lambda}$ in section 7.

Claim I

Machine L1 is more efficient than L0. The underlying assumption in this claim is

that forming a closure is more expensive than providing the tag bits and checking the node in the traversal.

3. Combinatory system and machine models

Combinators are defined as terms without free variables. We consider following combinators most important from practical point of view; **S** $\equiv \lambda xyz.xz(yz)$, **K** $\equiv \lambda xy.x$, **B** $\equiv \lambda xyz.x(yz)$, **C** $\equiv \lambda xyz.xzy$ and **I** $\equiv \lambda x.x$.

We recapitulate the method for translating terms to combinators. [10, 3]
Let the term to be translated be $\lambda x.M$. We eliminate x using the following algorithm.

Case 1: M is a single symbol

$M \equiv x$	**I**
$M \not\equiv x$	**K**M

Case 2: Otherwise, let $M \equiv M_1M_2$

- $x \notin N_1$ and $x \notin N_2$ **K**(N_1N_2)

  $x \in N_1$ means that N_1 contains free occurrences of x.

  Likewise $x \notin N_1$ is defined to the contrary.
- $x \in N_1$ and $x \notin N_2$ **C**$\tilde{N}_1N_2$

  where the term with ~ is the translated term by this algorithm.
- $x \notin N_1$ and $x \in N_2$ **B**$N_1\tilde{N}_2$
- $x \in N_1$ and $x \in N_2$ **S**$\tilde{N}_1\tilde{N}_2$

In the case of an n-variate term, $\lambda x_1..x_n.M$, recursively apply the above algorithm starting with $\lambda x_n.M$.

Machine CO (Turner's combinatory reduction machine)

Machine CO reduces the translated combinatory terms using the leftmost reduction strategy. The combinatory terms are represented in the same way as λ-terms. [10] described in detail the reduction algorithm.

Essentially, the reduction is performed in two phases; first remove the combinators by copying the nodes and distributing the arguments to proper places (distribution phase), and then reduce the constructed subtree by δ-rules (reduction phase). This

reduction is performed bottom-up. It is instructive to note the similarity between the closure constructed in ρ_λ (and ρ'_λ) and the subtree constructed immediately after the removal of combinators. The picture will become clear as we introduce a Σ-labelled tree.

4. Σ-labelled tree and labelled reducers

A Σ-labelled tree is a tree where at each node (both terminal and non-terminal) of a labelled tree a sequence of symbols in the set Σ can be attached as a label. A distinguished symbol ε is used to denote an empty sequence.

A tree in Fig. 4a is a $\{\lambda x\}$-labelled tree. (We simple call it $\{\lambda\}$-labelled tree.) From $\{\lambda\}$-labelled tree, we construct a {**S**, **K**, **I**}-labelled tree using following equations. We consider first a single variable case.

$\lambda x.M_1M_2 = \boldsymbol{S}(\lambda x.M_1)(\lambda x.M_2)$ (1)

$\lambda x.M = \boldsymbol{I}$ if $M \equiv x$ (2)

$\lambda x.M = \boldsymbol{K}M$ if M is an atom other than x (3)

Translating the term of the lefthand side of the equations to the righthand side and regarding combinators as a label, we obtain a {**S**, **K**, **I**, λ}-labelled tree. Figure 4b shows a {**S**, **K**, **I**, λ}-labelled tree constructed from a λ-term tree. Variable names bound by λ are entirely eliminated by the repeated application of the above rules. Eventually we get {**S**, **K**, **I**}-labelled tree. (cf. Fig. 4c)

The leaf nodes of the {**S**, **K**, **I**}-labelled tree are either labelled as **I** or **K**. The **I**-labelled nodes are empty, and **K**-labelled nodes have a constant. For technical reasons (such as enumeration of leaves, or debugging), instead of an empty **I**-labelled node, we use an **I'**-labelled node in which a variable name to be eliminated in Turner's translation scheme is retained.

That is, $\lambda x_i.M$ is translated to $\boldsymbol{I}'x_i$ when $M \equiv x_i$ and the corresponding reduction rule is $\boldsymbol{I}'x_i a \rightarrow a$. Here the variable is now regarded as a constant. Mostly in our discussion the presence of the variables is ignored, however.

In the case that one of M_1 and M_2 does not have free occurrences of variable x, we have the following:

$x \notin M_1 \quad \lambda x.M_1M_2 = \mathbf{B}M_1(\lambda x.M_2)$ (4)

$x \notin M_2 \quad \lambda x.M_1M_2 = \mathbf{C}(\lambda x.M_1)M_2$ (5)

B and C make more optimized reduction possible, as we see in the following reduction algorithm ρ_C.

We now extend the labelled tree construction to the case of an n-variate λ-term. We first consider the following lemma.

Lemma 1

$\lambda\vec{x}_n.M_1M_2 = \mathbf{S}'^{n-1}\mathbf{S}(\lambda\vec{x}_n.M_1)(\lambda\vec{x}_n.M_2)$ (6)

Here we use the following notation:

$\mathbf{S}'^{n}M \equiv \mathbf{S}'(\mathbf{S}'^{n-1}M)$, and $\mathbf{S}'^{0}M \equiv M$ for any M,

where S' is defined as $\lambda kxyz.k(xz)(yz)$.

Proof: By systematic renaming of variables we rewrite $\lambda\vec{x}_n.M_1M_2$ as $\lambda\overline{t}_n.\overline{M}_1.\overline{M}_2 \equiv \lambda t_n..t_1.\overline{M}_1.\overline{M}_2$ and prove

$\lambda\overline{t}_n.\overline{M}_1\overline{M}_2 = \mathbf{S}'^{n-1}\mathbf{S}(\lambda\overline{t}_n.\overline{M}_1)(\lambda\overline{t}_n.\overline{M}_2)$

Induction on n.

When n=1, obvious from equation (1).

$\lambda\overline{t}_n.\overline{M}_1\overline{M}_2 \equiv \lambda t_n.(\lambda\overline{t}_{n-1}.\overline{M}_1\overline{M}_2)$

$= \lambda t_n.(\mathbf{S}'^{n-2}\mathbf{S}(\lambda\overline{t}_{n-1}.\overline{M}_1)(\lambda\overline{t}_{n-1}.\overline{M}_2)) = \mathbf{S}'(\mathbf{S}'^{n-2}\mathbf{S})(\lambda t_n.(\lambda\overline{t}_{n-1}.\overline{M}_1))(\lambda t_n.(\lambda\overline{t}_{n-1}.\overline{M}_2))$

$= \mathbf{S}'^{n-1}\mathbf{S}(\lambda\overline{t}_n.\overline{M}_1)(\lambda\overline{t}_n.\overline{M}_2)$ □

We have a label $\mathbf{S}'^{n-1}\mathbf{S}$ at the root node of the tree $\lambda\vec{x}_n.M_1M_2$. We can generate 'optimized' labels using B' ($\equiv\lambda kxyz.kx(yz)$) or C' ($\equiv\lambda kxyz.k(xz)y$) instead of S' depending upon the presence of a free occurrence of a variable to be eliminated in the subtrees. That is, we have, for example,

$\lambda\vec{x}_n.M_1M_2 = \mathbf{S}'(\mathbf{B}'\mathbf{S}'^{n-3}\mathbf{S})(\lambda x_1x_3\ldots x_n.M_1)(\lambda\vec{x}_n.M_2)$ when $x_2 \notin M_1$

and

$\lambda\vec{x}_n.M_1M_2 = \mathbf{S}'(\mathbf{C}'\mathbf{S}'^{n-3}\mathbf{S})(\lambda\vec{x}_n.M_1)(\lambda x_1x_3\ldots x_n.M_2)$ when $x_2 \notin M_2$

In the label, a prime of the combinator is omitted since by the way of the genera-

tion of combinators in labels all the combinators in the sequence except for the rightmost one are always with a prime. Hereafter all the combinators with a prime are treated as if ones without a prime.

Machine C1 (Σ-labelled combinatory reducer)

Machine C1 reduces the Σ-labelled tree by the leftmost reduction. We use Σ = {**S**, **B**, **C**, **K**, **I**} and Algorithm ρ_C below for reduction.

Algorithm ρ_C

$\rho_C(t,stack) \equiv$
/* t is initially a tree (later modified to a graph) to be reduced.
$stack$ is either empty or $<t_1,t_2,..,t_n>$ where t_i is a subgraph so far stacked. */
if t is atom
then if t is a terminal object
then return

$$\begin{cases} t \text{ when } stack \text{ is empty} \\ t\rho_C(t_1,newstack()),..,\rho_C(t_n,newstack()) \text{ when } stack \text{ is} <t_1,t_2,..,t_n> \end{cases}$$

else /* t is δ */
return

$$\begin{cases} tt_1t_2..t_n \text{ when } n \text{ is less than the number of the terms} \\ \text{required in this } \delta\text{-reduction} \\ \rho_C(w,stack') \text{ otherwise} \end{cases}$$

where w is the result of the δ-reduction
i.e. $w=t(t_1,\ldots,t_i)$ (cf. note †6 in Algorithm ρ_λ)
and $stack'$ $(=<t_{i+1},..,t_n>)$ is the $stack$ after the reduction.

else
let L,l,r be the label, left, right subgraphs of graph t, respectively.
/* In the case of a terminal node only the left subgraph field is used */
if L of t is ε
then return $\rho_C(l \text{ of } t, <t,t_1,..,t_n>)$
else
if $stack$ is empty then return t
else
let $label$ of t is $X_1..X_n$
(1) X_1 = **S**
l of t_1 ← $newnode^{\dagger}(X_2..X_n, l \text{ of } t, r \text{ of } t_1)$
r of t_1 ← $newnode(X_2..X_n, r \text{ of } t, r \text{ of } t_1)$
return $\rho_C(l \text{ of } t, <(l \text{ of } t_1), t_2, .., t_n>)$
(2) X_1 = **B**
r of t_1 ← $newnode(X_2..X_n, r \text{ of } t, r \text{ of } t_1)$
l of t_1 ← l of t
return $\rho_C(l \text{ of } t, <t_1,t_2,..,t_n>)$
(4) X_1 = **C**
l of t_1 ← $newnode(X_2..X_n, l \text{ of } t, r \text{ of } t_1)$
r of t_1 ← r of t
return $\rho_C(l \text{ of } t, <(l \text{ of } t_1), t_2, .., t_n>)$
(5) X_1 = **K**
l of t_1 ← l of t
return $\rho_C(t_1, <t_2, .., t_n>)$
(6) X_1 = **I**
return $\rho_C(l \text{ of } t_1, <t_2, .., t_n>)$

note

† *newnode*(l, lt, rt) constructs a node whose label is l, left field is subtree lt, and right field is subtree rt.

Now an intuitive interpretation of the working of algorithm ρ_C is due.
Let T_C be a labelled tree corresponding to the combinatory term translated from $\lambda x_1..x_n.M$. When $\lambda x_1..x_n.M$ is applied to some term a_1, algorithm ρ_C distributes a_1 to the leaves where x_1 is stored (the node with label I' and the constant x_1) using the first combinator of the labels on the distribution path as a directive [6]. On its way to the destination, ρ_C constructs a subtree. (Since a subtree is shared by other subtrees, the resulting structure is in fact a acyclic graph.) On return from the leaf, ρ_C reduces the subgraph (by δ-rule) if possible. Let the resulting tree be T_C^1. Similarly, x_j is distributed to the leaves of x_j in T_C^{j-1} for j = 2,3, .., n.

Assume that the cost of the provision of a label in each node and the check of the label at the reduction time is lower than that of Turner's system.

Claim II: With proper hardware support which makes the above assumption valid, machine C1 with Σ={**S**, **B**, **C**, **K**, **I**} is faster than machine C0.

It is clear that a machine with Σ = {**S**, **B**, **C**, **K**, **I**} is more efficient than a machine with Σ = {**S**, **K**, **I**} since the former makes use of the prior knowledge of the presence of the occurrences of free variables in the subtrees, and does not construct redundant nodes.

With the same hardware arrangement for the λ-term representation we can realize machine L1 as a $\{\lambda,s,b,c,\square\}$-labelled tree reducer, where $\square$ is a label for a closure. Note the correspondences of the roles of s and **S**, b and **B**, and c and **C** in machines L1 and C1.

5. Initial comparison of λ-reducers and closure reducers

We are ready to make a comparison between machines L1 and C1. We first give a data

structure for a labelled tree. A *cell* is a structure which consists of three fields; *label* field, *l* field, and *r* field. In *l* and *r* fields, a pointer to the left subtree and a pointer to the right subtree are stored, respectively. For the label of the node at most $n\log_2|\Sigma|$ bits are needed to accommodate a label, where n is the number of λ-variables in the original term $\lambda\vec{x}_n.M$ and $|\Sigma|$ is the cardinality of the Σ. We assume a fixed bit field for the label in the following treatment. Because the bit requirement for the label does not increase during the reduction, this assumption does not affect the validity of the following analysis. A cell can also accommodate a dotted pair and a closure; in the former case the label field is not used, and in the latter case *l* field contains a pointer to a subtree, *r* field an environment and *label* field is marked as $\square$.

Let T_λ be a tree of the $\{\lambda\}$-labelled tree and T_C be a corresponding translated {**S**, **B**, **C**, **K**, **I**}-labelled tree.

We define an underlying tree T' of tree T as a tree with all the labels and leaves removed.

Lemma 2

T'_C and T'_λ are equivalent, hence T_C and T_λ are similar.

For the notion of 'similarity' and 'equivalence' of trees, see [7, p. 326].

Proof: Since the underlying tree of T_λ is intact during the process of the construction of T_C as illustrated in (6), T'_C is obviously equivalent to T'_λ.

We first consider the reduction $(\lambda\vec{x}_n.M)\vec{N}_n \rightarrow M[\vec{x}_n:=\vec{N}_n]$, where M does not contain λ. The original β-reduction $(\lambda\vec{x}_n.M)\vec{N}_n \rightarrow M[\vec{x}_n:=\vec{N}_n]$ is fully simulated by ρ'_λ when $\vec{N}_n$ consists only of terminal objects. (In this case $M[\vec{x}_n:=\vec{N}_n]$ is a normal form.) The simulation by ρ'_λ is the same as the preorder traversal of the labelled tree representing $\lambda\vec{x}_n.M$. The corresponding reduction by ρ_C is also characterized by the visits of nodes of the corresponding {**S**, **B**, **C**, **K**, **I**}-labelled tree T_C. Under the above conditions, the orders of the traversal of T_C by ρ_C and T_λ by ρ'_λ, i.e. the orders of the visits of the nodes of the trees, are the same since the underlying

trees are equivalent.

Therefore the comparison of the efficiency can be made for the cost, e.g. storage consumption, involved in the traversal between any subtrees S_C of T_C and S_λ of T_λ when both represent the same subtree of a λ-term.

We first take up a general case.

Suppose we reduce $(\lambda\vec{x}_n.M_{\sigma(\vec{x}_n)})\vec{N}_n$ (7)

where $M_{\sigma(\vec{x}_n)}$ is represented by a tree $T_{\sigma(\vec{x}_n)}$ with leaves $\sigma(\vec{x}_n)$. $\sigma(\vec{x}_n)$ $(\equiv\sigma(x_1)..\sigma(x_n))$ is a permutation of $\vec{x}_n$, and $\vec{N}_n=N_1..N_n$ where $N_i, i=1,2,..,n$ is a term consisting only of terminal objects. Let $T_{\sigma(\vec{x}_n)C}$ and $T_{\sigma(\vec{x}_n)\lambda}$ respectively be a $\{\lambda\}$-labelled tree and {**S**, **B**, **C**, **K**, **I**}-labelled tree, both representing $(\lambda\vec{x}_n.M_{\sigma(\vec{x}_n)})\vec{N}_n$. During the reduction of (7) by ρ'_λ, all the nodes of $T_{\sigma(\vec{x}_n)}$ are visited. In the $\{\lambda\}$-labelled tree, a closure is made at every node (including leaf). Therefore, the number of closures created is $2n - 1$ (cells). The total number of cells consumed during the traversal is therefore $4n-1$, since $2n$ cells are required by the construction of the environment of n variables. (See below)

We have more general statements (Lemmas 3 and 4) about the storage consumption by ρ'_λ and ρ'_λ in the case of $(\lambda\vec{x}_n.M)\vec{N}_n$ in which M is represented by $\{\lambda\}$-labelled tree T.

Lemma 3

In ρ'_λ, binding $\mathbf{[}(\lambda\vec{x}_n.M)\vec{N}_n,\varepsilon\mathbf{]}$ -> $\mathbf{[}M,((x_n.N_n)(x_{n-1}.N_{n-1})\ ..\ (x_1.N_1).\varepsilon)\mathbf{]}$ requires $2n$ cells on machine L1.

(The lemma relies on the representation of the environment list which is in our case a list of dotted pairs.)

Lemma 4

Given a $\{\lambda\}$-labelled tree T_λ with m leaves. The total number of closures created during the traversal of all the nodes by algorithm ρ_λ is $2m-1$. In the case of $\{\lambda,s,b,c,\square\}$-labelled tree, that number by algorithm ρ'_λ is $2m-1$, maximum

Proof:

Since in algorithm ρ_λ a closure is created at each node of tree T_λ, the total number of closures is equal to the number of the nodes of tree T_λ. □

Proposition 1

The maximum number of cells consumed during reduction (7) by algorithm ρ'_λ on $\{\lambda, s, b, c, \square\}$-labelled tree is $2n+2m-1$.

Proof: Obvious from lemmas 3 and 4. □

All the nodes of $T_{\sigma(\vec{x}_n)C}$ are visited once for each distribution of x_i, $i=1,2,..,n$, by {S, B, C, K, I}-reducer. Hence, we have the following proposition.

Proposition 2

The average number of cells consumed during the reduction (7) by algorithm ρ_C on {S, B, C, K, I}-labelled tree is $0(n^{1.5})$.

Proof:

During the distribution phase of the reduction each variable creates a cell whenever it visits the node. (In fact, in this case all the labels except for leaves consist of only B's and C's.) The total number of visits to reach the leaves summed over all the variables is equal to the path length of the tree. Since the average path length over all binary trees is $0(n^{1.5})$, the average number of cells consumed is $0(n^{1.5})$. □

The amount of the storage consumption by ρ_C varies greatly depending upon the shapes of trees.

The worst case is with the tree

$T_{\sigma(\vec{x}_n)} = (x_1 (x_2 \ldots (x_{n-1} x_n) \ldots))$

in which the storage consumption is

$\frac{n(n+1)}{2} - 1$ for $n \geqq 3$.

and the best case is with

$T_{\sigma(\vec{x})} = x_1 x_2 .. x_n$,

in which case the storage consumption is none, since $\lambda\vec{x}_n.\vec{x}_n$ = I by extensionality.

In practice, some leaves are constants. Moreover, the cell consumptions on both

machines C1 and L1 are different depending upon the pattern of variable occurrences. Consider first the traversal of the trees in Figs. 5a and 5b. During the reduction, **S**, **B** and **C** consume 2, 1 and 1 cells, respectively by algorithm ρ_C and s, b and c consume 1, 1 and 0 cell, respectively by algorithm ρ'_λ. Hence the numbers of cells consumed during the traversal of the trees are as follows for the cases shown in Figs. 5a and 5b:

$2+n$ for the traversal of the $\{\lambda,s,b,c\}$-labelled tree representing $\lambda x.k_1(k_2..(k_n x)...)$ in Fig. 5a by algorithm ρ'_λ, and

n for the traversal of the corresponding {**S**, **B**, **C**, **K**, **I**}-labelled tree in Fig. 5b by algorithm ρ_C.

On the other hand, the traversal of the trees representing $\lambda x.xk_n..k_2k_1$ (Figs. 6a and 6b) consumes 2 cells by ρ'_λ and $2n$ by ρ_C. In the case of single variate λ-terms ρ_C only outperforms ρ'_λ when the number of **S** and **C** encountered during the traversal is less than 2.

The disadvantage of ρ'_λ is the creation of an environment list during the parameter binding. Therefore, it is possible to think of unusual examples such as $\lambda x_1..x_n.x_1x_2$ in which some λ-variables do not appear in the body of the λ-term. In this case ρ_C consumes 2 cells whereas ρ'_λ consumes $2n+1$.

6. Elaboration of λ-reducers

The investigation in the previous section shows that in simple cases considered in section 5:

- Generally speaking, machine L1 is more advantageous than machine C1 in terms of the storage consumption.
- As the number of arguments increases, the difference of the amount of the storage consumption become large; and hence machine L1 becomes more advantageous than machine C1.

On the other hand we observe the following advantages for machine C1:

(1) No search for variable names is needed at the reduction time.

(2) Uniform treatment of application of terms is possible, and hence we can avoid the extra level of complication incurred by forming an environment at the time of binding.

(3) Partial evaluation is automatically in effect in machine C1, whereas in L2 not. As for the point (1), variable names can be eliminated before the reduction in machine L1. The method for variable elimination is similar to the one adopted by Automath [2] and SECD machine.

Nameless-λ-reducer L2

For each variable x_i in $\lambda\vec{x}_n.M$, the relative position (the order from left) of its bounded pair in the environment to be formed when $\lambda\vec{x}_n.M$ is applied is known statically. We replace free occurrences of x_i in M by that relative position, say $\underline{k}$. $\underline{k}$ is treated as a function to be called *envfun* to get the kth element (from left) in the environment list. The numbering rule is similar to the ones adopted by block structured languages such as Algol 60. For example,

$\lambda x_1 x_2.(x_1(\lambda x_1 x_2.x_1 \ \ldots\ x_2)(\lambda y_1 y_2.x_2 y_2))$

2	1	2	2	1	2	1	2	1	3	1

λ is replaced by λ_n where n is the number of λ-variables. The environment formerly defined as a list of dotted pairs of a variable name and a corresponding value are now consisting of only values. During the reduction, $\underline{k}$ gets the kth element of the current environment list, and update the kth element after the reduction. λ_n is changed to λ_{n-1} when that term is applied to an argument. Slight modifications of algorithm ρ'_λ in the environment handling (i.e. environment search and environment formation *) are sufficient for machine L2 to operate (see $\rho_{C\lambda}$ in section 8).
The cell consumption in the reduction by machine L2 incurred by parameter binding is now n for n variables. Moreover, the speed of the environment look-up is increased.

Point (3) mentioned at the beginning of this section needs more explanation.

---

* When λ-terms are nested as above, the search time for the outer level of λ-variables become large, i.e. proportionate to the product of the number of λ-terms and the number of recursions of the same level of λ-terms. This trouble can be taken care of easily in the combinatory reducer. In the λ-reducer, it can be handled by introducing a LABEL function as used in Lisp 1.5. The technique is similar to static chaining used in compiling block structured languages. We do not discuss this point further since this is besides the point of our argument.

Suppose we have $\lambda x_1 x_2.+\ (+\ x_1\ 1)x_2$. When term $X(\equiv(\lambda x_1 x_2.+\ (+\ x_1\ 1)x_2)10)$ is reduced, a {**S**, **B**, **C**, **K**, **I**}-labelled tree corresponding to $(\lambda x_2.+\ (+\ 10\ 1)x_2)$ is constructed. Suppose multiple copies of term X (actually a pointer to X) is distributed to several places. When X is further reduced, e.g. when X is applied to 3, the expression + 10 1 is reduced to 11 and the effect is felt by all the terms which reference the term X in the combinatory reducer. On the other hand, on machine L1 (also on L2) the closure $\langle(+\ x_1\ 1),((x_1.10).\xi)\rangle$ created during the reduction of $\langle(\lambda x_2.+(+\ x_1\ 1)\ x_2),((x_1.10).\xi)\rangle$ is computed every time X is applied to some value. The effect of this partial evaluation may be great if a partially applied function such as above is distributed and reduced many times.

Fortunately, even on machine L2 the same effect is achieved by rearranging terms. The method is due to [4] . We identify a *maximum free term* i.e. term consisting only of constants and free variables, and shift it outside the enclosing function body. In this case the maximum free term is $(+\ x_1\ 1)$ in $\lambda x_2.+\ (+\ x_1\ 1)\ x_2$. We change $\lambda x_1 x_2.+\ (+\ x_1\ 1)\ x_2$ to $\lambda x_1.((\lambda x_3 x_2.+\ x_3\ x_2)(+\ x_1\ 1))$. Then the reduction of x_3 only once induces the reduction $(+\ x_1\ 1)$, and the effect of this reduction is felt by all the terms that reference the closure of $\lambda x_1.((\lambda x_3 x_2.+\ x_3\ x_2)(+\ x_1\ 1))$.

The other aspect of the partial evaluation is self-optimization [10] in conjunction with the local definition of a recursive function. The self-optimization is equally well taken care of by a λ-reducer. We discuss this point in the example *foldr* in the next section.

7. Examples of comparisons

In the following analysis, the comparisons of the performances are made by measuring the difference of the numbers of the cells consumed during the reduction. In doing so, we assume that the amount of time used for δ-reduction is the same in both reducers. We use the following three examples all of which are given in [10] ; *factorial*, *foldr* and *twice*.

factorial

def *factorial* = λn. *cond* (*eq n 0*) *1* (*times n* (*factorial* (- *n 1*)))

Figure 7-a shows {λ,s,b,c}-labelled tree representing *factorial*. The number of cells consumed during the reduction of *factorial n* on machine L2 is:

4 (3 for the traverse and 1 for the environment) when n = 0, and 6*n*+4 when n ≧1 .

Figure 7-b shows the corresponding {**S**, **B**, **C**, **K**, **I**}-labelled tree. The number of cells during the reduction of the tree on machine C1 is

6 when $n = 0$, and $12n + 6$ when $n \geqq 1$.

foldr

def *foldr* = λ*fkx.cond* (*null x*) *k* (*f* (*hd x*) (*foldr f k* (*tl x*)))

foldr is a list manipulating function used in conjunction with a binary function *f*. For example,

def *sum* = *foldr plus 0*

def *product* = *foldr times 1* .

Figures 8-a and 8-b show the corresponding labelled trees representing *foldr*. The numbers of cells consumed are $14n + 7$ on machine L2 and $24n + 9$ on machine C1, respectively for the given list whose length is *n*.

However, this comparison may not be fair since in machine C1 the self-optimization is not workable in this definition. To realize the self-optimization, we introduce the following local definition of function *g*.

def *foldr* = λ*fkx*. *gx*

where def *g* = λ*y*. *cond* (*null y*) *k* (*f* (*hd y*)(*g* (*tl y*)))

The reduction of *foldr plus 0* e.g., using the above definition of *foldr* is performed only once. The local definition of a recursive function such as the above can be transformed to a λ-term using combinator **Y** (**Y** is a combinator with the following reduction rule **Y***f* = *f*(**Y***f*)).

def *foldr* = λ*fk*. **Y**(λ*gx.cond* (*null x*) *k* (*f* (*hd x*)(*g* (*tl x*))))

The {**S**, **B**, **C**, **K**, **I**}-labelled tree representing the above term is shown in Fig. 8-c. In this case the cell consumption becomes $11n + 15$, which is comparable to the cell consumption on L2.

Similar optimization effect can be achieved by introducing δ-constant *label* which is similar to LABEL function of Lisp. *label* adds a new local definition of a

function to the current environment.

def $foldr = \lambda fk.\ label\ g\ (\lambda x.cond\ (null\ x)\ k\ (f\ (hd\ x)(g\ (tl\ x))))$

Figure 8-d shows the corresponding labelled tree.

In this case the number of consumed cells is

2 for creating the environments for f and k,

2 for the execution of *label*

(1 for constructing a closure, and

1 for adding the definition of a function into the current environment),

$10n + 5$ for execution of the locally defined function g,

hence in total

$10n + 9$.

twice

def $twice = \lambda fx.f(fx)$

twice is a function which applies a given function twice.

Figures 9-a and 9-b show $\{\lambda,s,b,c\}$-labelled tree and {**S**, **B**, **C**, **K**, **I**}-labelled tree representing *twice*, respectively. The numbers of the cell consumption are 4 and 5 respectively. However, on machine C1 we can generate a more optimized combinatory expression for *twice*. That is, we have $\lambda fx.f(fx) = SBI$, in which case the cell consumption for each application of *twice* becomes only 3. In the above example, the extensionality plays an essential role in simplifying the resulting combinatory expression:

$\lambda fx.f(fx) = \lambda f.(Bf)(\lambda x.fx)$

$= \lambda f.(Bf)f$ by extensionality on $\lambda x.fx$

$= S(\lambda f.(Bf))\lambda f.f$

$=$ **SBI** by extensionality on $\lambda f.Bf$

Thus using the extensionality we can effectively transform the original labelled tree to a more optimized (simpler) tree. In such a case, our method of comparison does not work well since the shapes of the initial trees to be reduced are different in the closure and combinatory reduction schemes. However, in real programs, δ-constants are scattered in the terms representing the programs and prevent the extensive application of the extensionality, as we see in the example of *factorial*.

In Turner's measurement *twice* is the only case in which Turner's combinatory reducer outperforms SECD machine in the applicative order evaluation. In this case the performance is susceptible to a small optimization. For example, if we counted 2 cells for the construction of the environment of a variable, the cell consumption would be 6 for machine L2. Therefore, the performance of machine C1 could be twice as high as that of L2.

8. Towards a more efficient reducer

To make our argument clear we use trees illustrated in Fig. 10. Suppose we reduce Ma_1a_2 by a combinatory reducer. In Fig. 10, the leaves x_1 and x_2 in M are initially empty. When Ma_1 is reduced, a_1 is placed in the leaf x_1, and a new tree is constructed (left subtree T_λ is shared). Further when La_2 (where $L \equiv Ma_1$) is reduced, another new tree is constructed. Hence we actually have three trees T_1, T_2 and T_3. T_λ in Fig. 10 is shared since T_λ is not "contaminated" by these applications of the terms. This observation leads us to have a *pure tree* and separate working storage (taken from heap) for variables. A pure tree is a tree with all the leaves of variables being replaced by a relative location for the variable within the working storage. A term to be reduced can be represented as a pair of a pure tree pt and separate working storage w for variables. Furthermore we need status bits to tell whether the storage for variables is filled with actual parameters. Let us call the machine which reduces this tree machine C2.

The pair above has a remarkable similarity to the labelled closure constructed in the labelled closure reduction system with the following correspondence:

pt	nameless labelled tree
w	the portion of the environment currently accessible
status bits	n in label λ_n
	(n indicates the number of remaining arguments to be applied)

As for the label of nodes of pt, we have two choices:

case 1, Use {S, B, C, K, I}.

In each node, each variable is inspected whether it is used in the subtrees and the closure should be constructed.

e.g. label=SB with $\mathcal{E}=(v_2\ v_1.\mathcal{E}')$

We create $\mathcal{E}_r=(v_2\ v_1.\mathcal{E}')$ for the right subtree and $\mathcal{E}_l=(v_1\ .\mathcal{E}')$ for the left subtree.

case 2, Use $\{s, b, c\}$.

On each node the use of an environment by the left and right subtrees is examined.

e.g. label=s with $\mathcal{E}=(v_2\ v_1.\mathcal{E}')$

For the same tree as the above example, label s is attached because the left and right subtrees have at least one variable occurrences and $\mathcal{E}$ is passed to both subtrees.

In case 1, we construct an environment for the more efficient access, whereas in case 2 we sacrifice speed for the economy of storage if the same variable is accessed several times. Although we departed from machine C1 in order to decrease the storage consumption, in case 1 we again return to a scheme in which the storage consumption is equal to that of C1. The storage is consumed in preparing the proper environment. One can easily see that what is performed in case 1 is essentially the same as in ρ_C. On the other hand, when we take case 2 we see that machine C2 is converged into machine L2.

To sum up, by improving the combinatory reducers we come to a nameless $\{\lambda,s,b,c,\square\}$-labelled tree reducer. The reduction algorithm $\rho_{C\lambda}$ for the $\{\lambda,s,b,c,\square\}$-labelled tree, is derived from ρ_C and ρ'_λ.

Let a closure representing term $(\lambda\vec{x}_n.M)\ [\vec{x}_i:=\vec{N}_i]$ by a labelled node be λ_{n-i}; $(\underline{M}.(N_1\ N_2\ \ldots\ N_i.\mathcal{E}))^*$ where $\underline{M}$ is a nameless λ-term.

Algorithm $\rho_{C\lambda}$:

$\rho_{C\lambda}\ (t,\ stack) \equiv$
let l; $(M,\mathcal{E})=t$ and $stack=\langle s_1,..,s_n\rangle$ or $\langle\rangle$, i.e. empty
(1) l is ε, i.e. M is atomic
if M is a terminal object
then resultis

$$\begin{cases} \varepsilon;M \text{ when } stack \text{ is empty} \\ \varepsilon;M\rho_{C\lambda}(s_1,(),newstack())..\rho_{C\lambda}(s_n,(),newstack()) \text{ otherwise} \end{cases}$$

---

* A symbol in front of the semicolon (in this case λ_{n-1}) is a label of the node.

else if M is δ
then resultis
$$\begin{cases} \varepsilon;Ms_1s_2..s_n \text{ when } n \text{ is less than the number of the terms required in this } \delta\text{-reduction} \\ \rho_{C\lambda}(w,\mathcal{E},stack') \text{ otherwise} \end{cases}$$

where w is the result of the δ-reduction, i.e. $w=M(s_1,...s_i)$.
and $stack'$ $(=<s_{i+1},..,s_n>)$ is the $stack$ after the reduction.

else /* M is *envfun* $\underline{k}$ */
let e_k = kth item on the environment list $\mathcal{E}$
$w = \rho_{C\lambda}(e_k,(),newstack())$
replace kth item on $\mathcal{E}$ with w
resultis $\rho_{C\lambda}(w,\mathcal{E},stack)$

(2) l is $\lambda_m l'$
resultis
$$\begin{cases} l;M \text{ when } stack \text{ is empty} \\ \lambda_{m-n} l';(M,(s_n\ s_{n-1}\ ...s_1.\mathcal{E})) & \text{when } m \geq n \\ \rho_{C\lambda}(l';M,(s_{n+m-1}s_{n+m-2}...s_n.\mathcal{E}),<s_{n+m},s_{n+m+1},..>) & \text{when } m \leqq n \end{cases}$$

(3) l is $\square$
let M = $(M',\ \mathcal{E}')$
resultis $\rho_{C\lambda}(M',\mathcal{E}',stack)$

(4) l is b, c or s in which case M is M_1M_2 (i.e. application)
l is b
resultis $\rho_{C\lambda}(M_1,<\square;(M_2,\mathcal{E}),s_1,s_2,..,s_n>)$
l is c
resultis $\rho_{C\lambda}(M_1,\mathcal{E},<M_2,s_1,s_2,..,s_n>)$
l is s
resultis $\rho_{C\lambda}(M_1,\mathcal{E},<\square;(M_2,\mathcal{E}),s_1,s_2,..,s_n>)$

9. Conclusion

Major work involved in reduction is the traversal, copying and reconstruction of the tree to be reduced. In this paper we analyzed the complexity involved in the reduction in terms of the storage consumption. Two known methods, the λ-reducer and the combinatory reducer, are reviewed and elaborated. When a term is represented as a labelled tree, the behavior of the reductions by both reducers become quite similar. The difference of the performances of the two reducers is then measured in terms of the storage consumed during the traversal of the tree. Contrary to the arguments for combinatory reducers claimed by Turner and Peyton Jones, our conclusion in the comparison is that the advantages of the combinatory reducer diminish when due considerations are made for the formations of closures before the reduction.

Our analysis also shows that there is a trade-off relationship between the access speed to variables and the storage consumption in the implementation of the left-

most reduction strategy of Λ-calculus. Fast access is enjoyed by the combinatory reducer at the expense of extra work on the distribution of arguments and small storage consumption by the labelled λ-reducer.

In this paper we will stop the elaboration of machine models at the labelled tree reducer, since the introduction of the labelled tree reducer reveals the essential work to be done on the reduction. Optimizations of machine models on both reducers would be fruitful. On the closure reducer side an obvious optimization is to 'compile' pt to make the traversal of the tree faster. Since the shape of pt is fixed, we can traverse the pt beforehand, and generate linear code from a tree. Therefore the design of the code is a next theme to pursue in speeding up the reduction. It is interesting to observe that similar approach is taken by Warren in speeding up Prolog (known as structure sharing) although the objects of the study are Prolog clauses [11].

Peyton Jones'es experiment compared the performance of machine C0 with machine L0, and measured the efficiencies of only Curried functions. (All the functions are treated as Curried in the closure reducer, but not decomposed into single-argument functions.) But the efficiency advantage of the combinatory reducer claimed in the paper [9] will be much smaller when algorithm ρ_λ is properly implemented as suggested in this paper. The comparison of the reduction stack depth is not essential in the performance analysis since both the closure reducer and the combinatory reducer traverse and reduce *similar* labelled trees.

Our observation is limited to the case of a single-processor machine. The advantage of functional programming in multi-processor environment is pointed out, since functional programming enjoys the property of referential transparency. One may argue that in the multi-processor environment the combinatory reducer is more advantageous than the closure reducer, since arguments are copied and distributed in the combinatory reducer. To prove the validity of this statement we need further analysis based on concrete multi-processor machine models. The invenstigation in this direction will be a future theme.

References

[1] Barendregt, H. P., *The Lambda Calculus, its Syntax and Semantics*, North-Holland Pub. Co., 1981

[2] de Bruijn, N. G., Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, *Indag Math*, 34, 381-392

[3] Hikita, T., On an average size of Turner's translation to combinator programs, *Journal of Information Processing*, to appear

[4] Hughes, R. J. M., Super combinators: A new implementation method for applicative languages, *Conference Record of the 1982 ACM symposium on LISP and functional programming* (Aug. 1982), 1-10

[5] Keller, R. M., FEL (Function-Equation Language) Programmer's Guide AMPS Technical Memorandum No. 7, Department of Computer Science, Univ. of Utah, March 1982, Revised April 1983

[6] Kennaway, J. R., The complexity of a translation of λ-calculus to combinators, University of East Anglia, Report CS/82/23/E, 1982

[7] Knuth, D. E., *The Art of Computer Programming, 2nd edition*, Vol.1, Addison-Wesley Pub. Co.

[8] Landin, P. J., The mechanical evaluation of expressions, *The Computer Journal*, Vol. 6, (1964), 308-320

[9] Peyton Jones, S. L., An investigation of the relative efficiency of combinators and lambda expressions, *Conference Record of the 1982 ACM symposium on LISP and functional programming* (Aug. 1982), 150-158

[10] Turner, D., A new implementation technique for applicative languages, *Software-Practice and Experiences*, Vol. 9, (1979), 31-49

[11] Warren, D., Implementing PROLOG - Compiling logic programs, 1 and 2, D.A.I. Research Report No. 39 and 40, University of Edinbough, 1977

Figure Captions

Figure 1 Representation of term $M_1M_2...M_n$

Figure 2 A snapshot of the reduction $(\lambda f(\lambda x.f(fx)))\ add1\ 0$

Figure 3 Conversion from graph to tree

Figure 4-a $\{\lambda\}$-labelled tree representing $\lambda x.cond\ (eq\ x\ 0)\ 1\ (-1)$

Figure 4-b {**S**, **K**, λ}-labelled tree representing $\lambda x.cond\ (eq\ x\ 0)\ 1\ (-1)$

Figure 4-c {**S**, **K**, **I**}-labelled tree representing $\lambda x.cond\ (eq\ x\ 0)\ 1\ (-1)$

Figure 5-a $\{\lambda,s,b,c\}$-labelled tree representing $\lambda x.k_1(k_2..(k_nx)..)$

Figure 5-b {**S**, **B**, **C**}-labelled tree representing $\lambda x.k_1(k_2..(k_nx)..)$
(Best case for ρ_C)

Figure 6-a $\{\lambda,s,b,c\}$-labelled tree representing $\lambda x.xk_n...k_2k_1$
(Case favorable to ρ'_λ)

Figure 6-b {**S**, **B**, **C**}-labelled tree representing $\lambda x.xk_n...k_2k_1$

Figure 7-a $\{\lambda,s,b,c\}$-labelled tree representing *factorial*

Figure 7-b {**S**, **B**, **C**}-labelled tree representing *factorial*

Figure 8-a $\{\lambda,s,b,c\}$-labelled tree representing *foldr*

Figure 8-b {**S**, **B**, **C**, **K**, **I**}-labelled tree representing *foldr*

Figure 8-c {**S**, **B**, **C**, **K**, **I**}-labelled tree representing *foldr*
(using **Y** combinator)

Figure 8-d $\{\lambda,s,b,c\}$-labelled tree representing *foldr*
(using function *label*)
Since *label* needs the current environment, *c*'s
are generated in the ancestory nodes of *label*.

Figure 9-a $\{\lambda,s,b,c\}$-labelled tree representing *twice*

Figure 9-b {**S**, **B**, **C**, **K**, **I**}-labelled tree representing *twice*

Figure 10 Construction of trees by a combinatory reducer

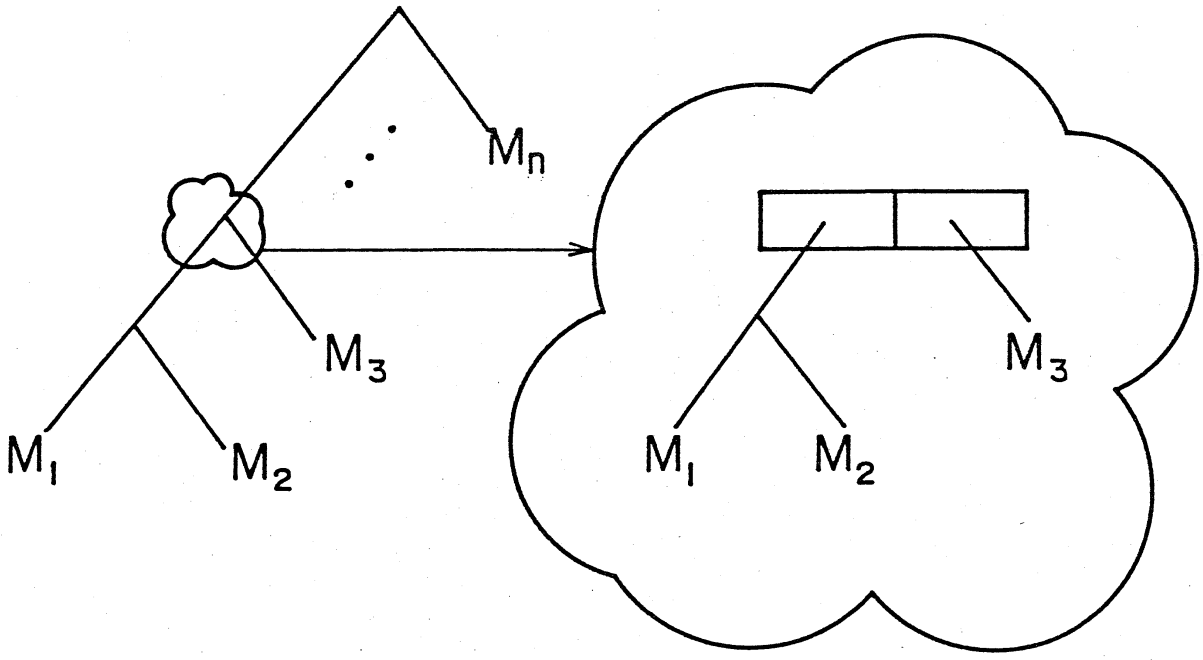

Fig. 1

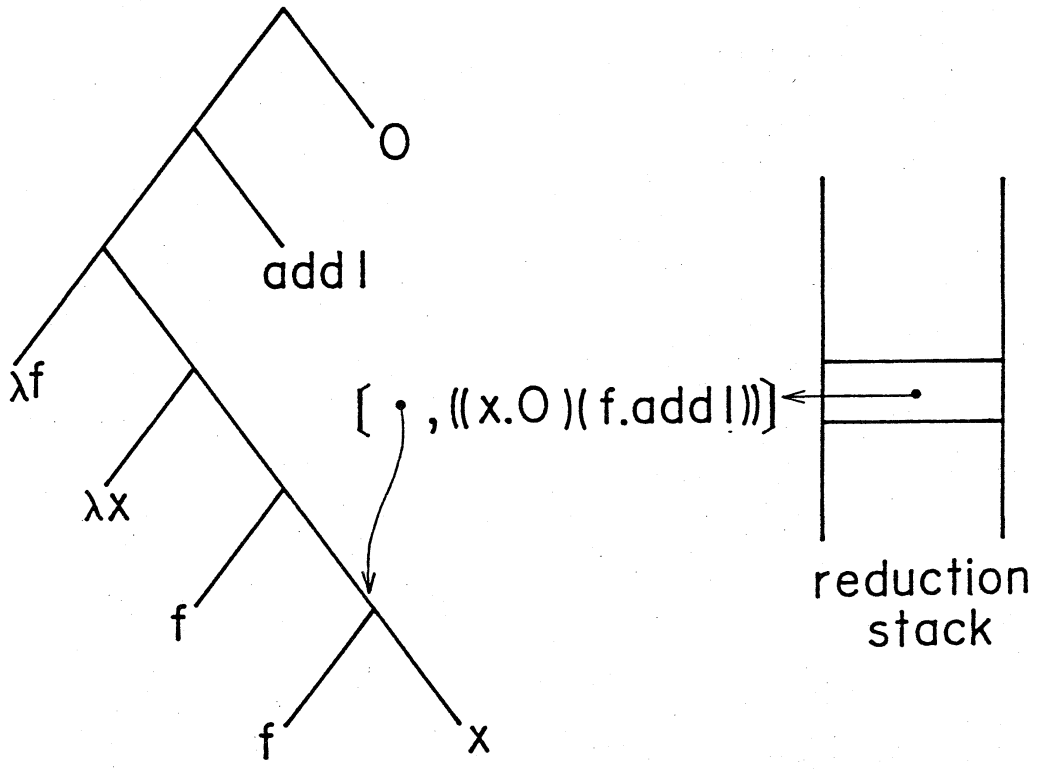

Fig. 2

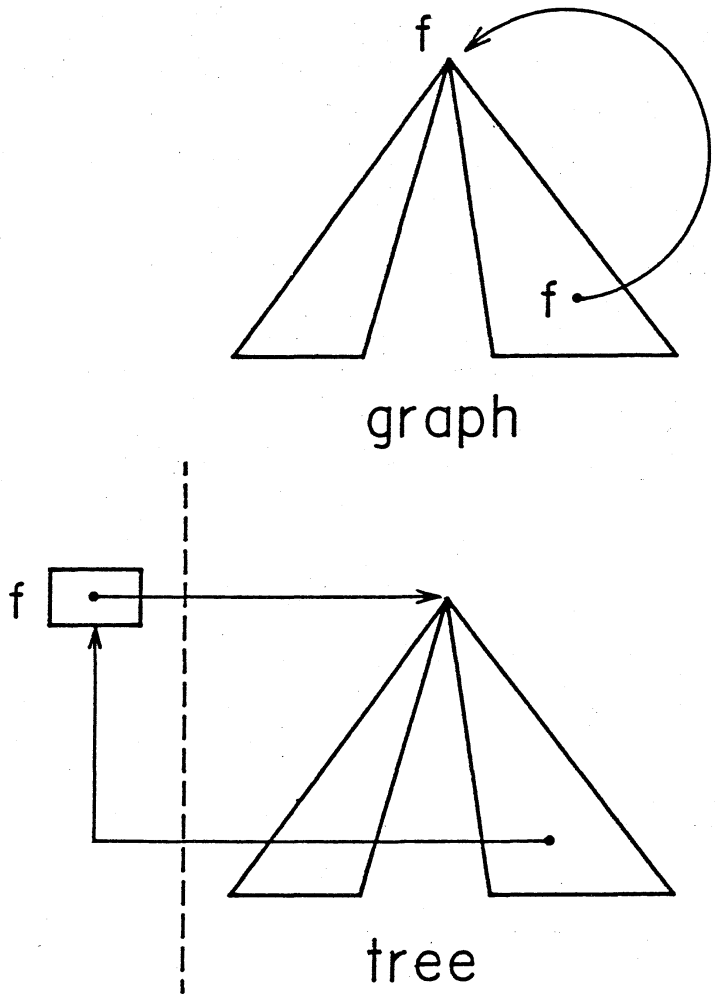

Fig. 3

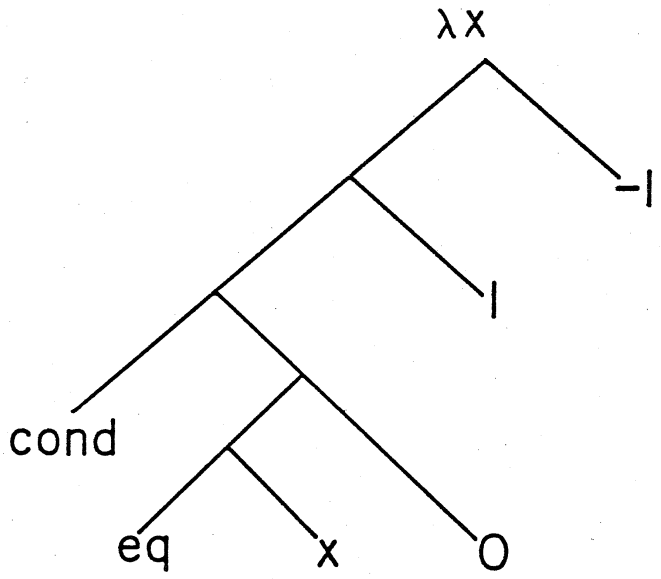

Fig. 4-a

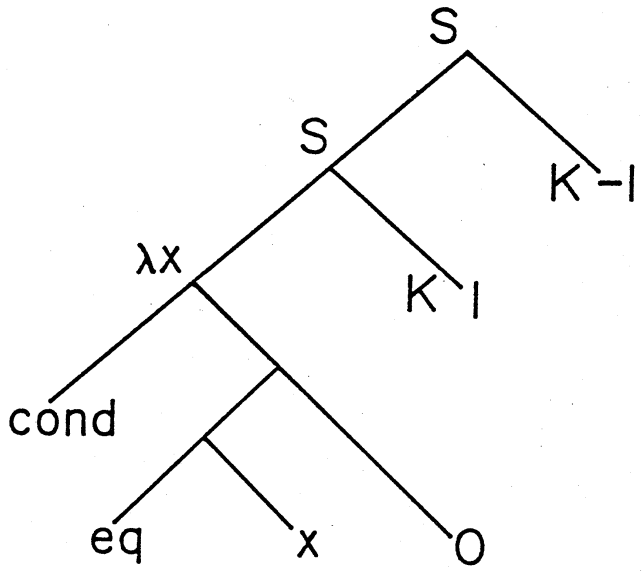

Fig. 4-b

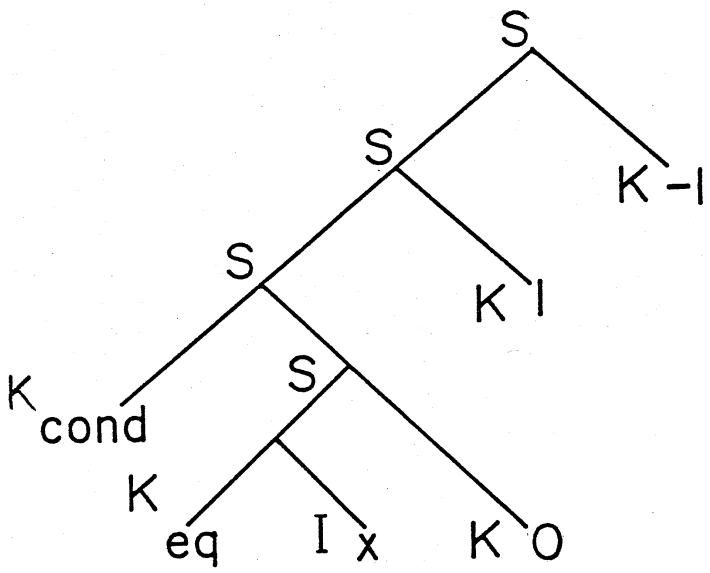

Fig. 4-c

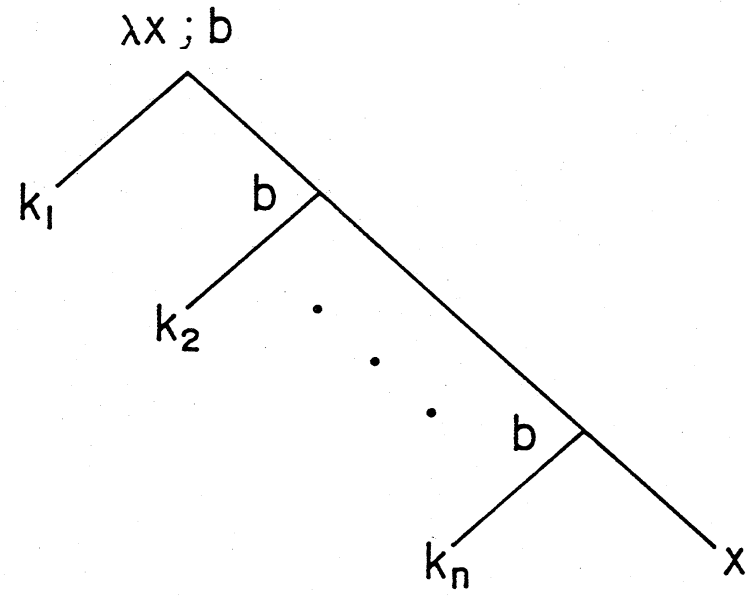

Fig. 5-a

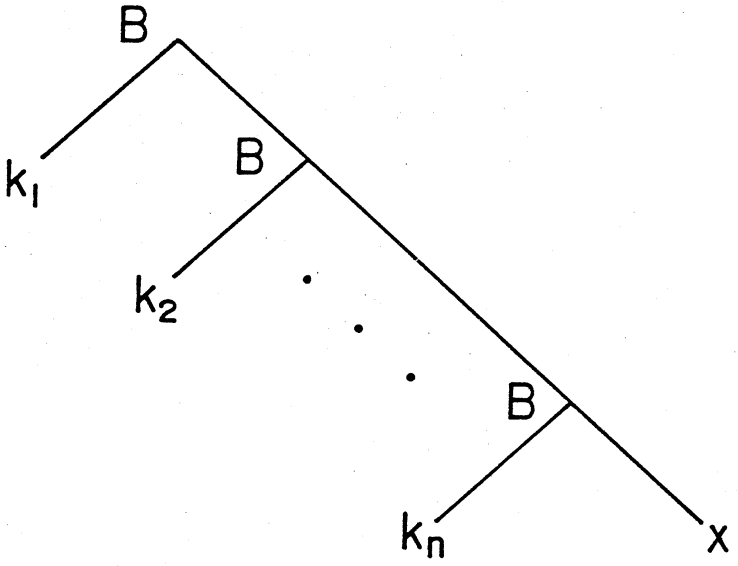

Fig. 5-b

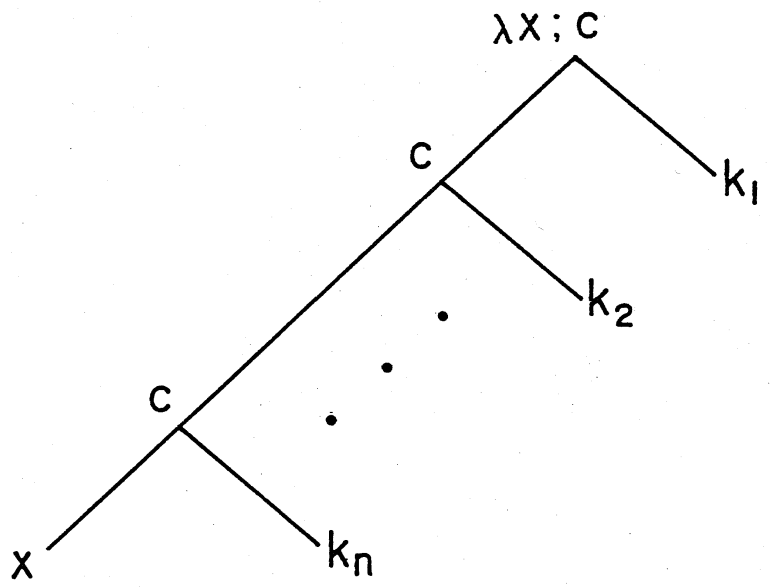

Fig. 6-a

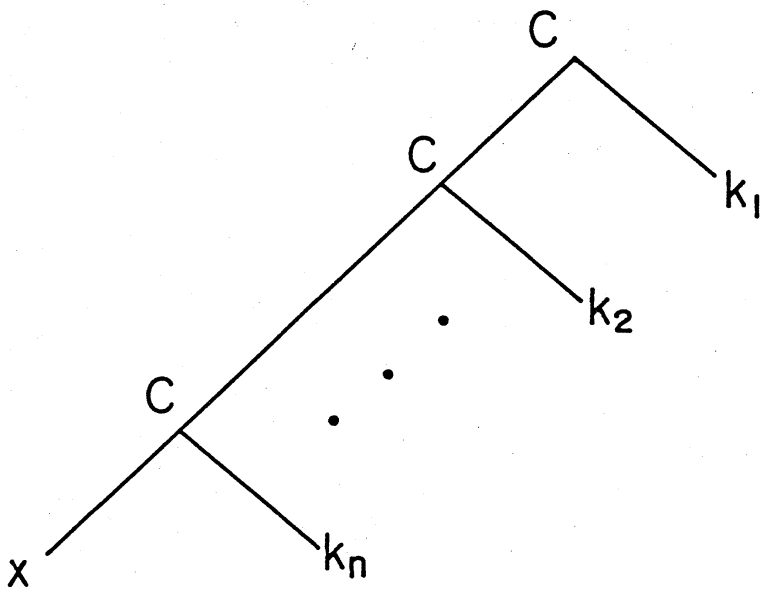

Fig. 6-b

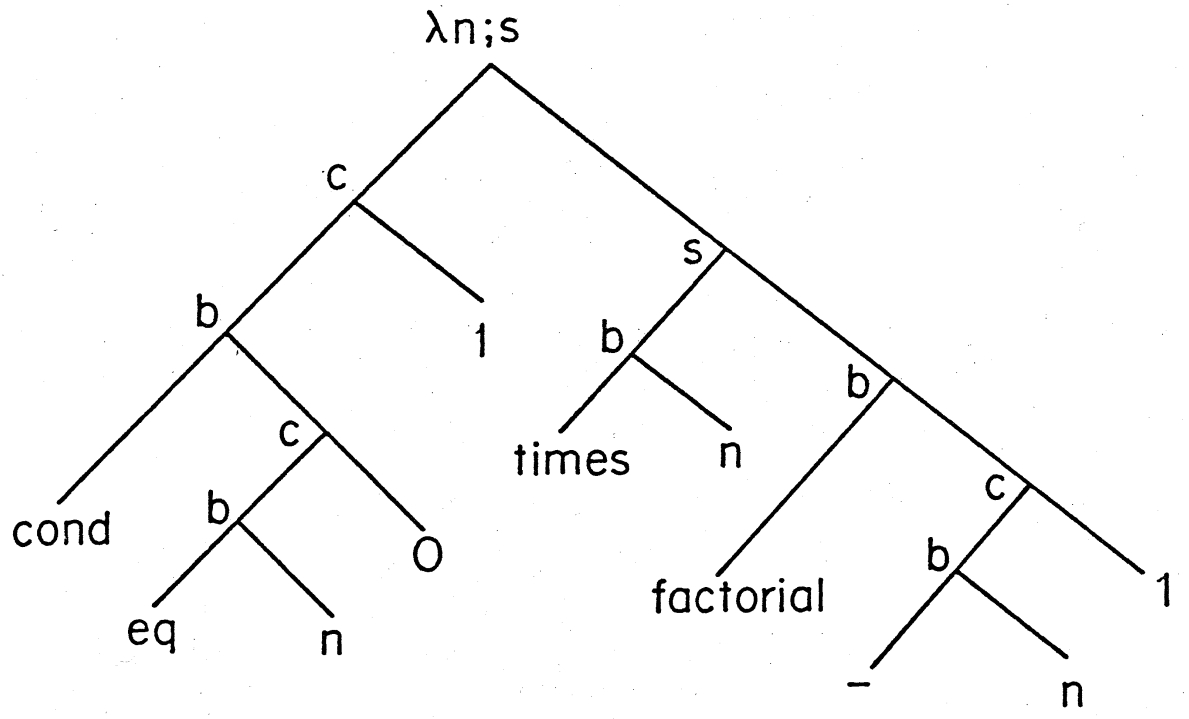
λn;s
c
s
b
1
b
b
c
times
n
cond
b
c
b
1
O
factorial
eq
n
b
n

Fig. 7-a

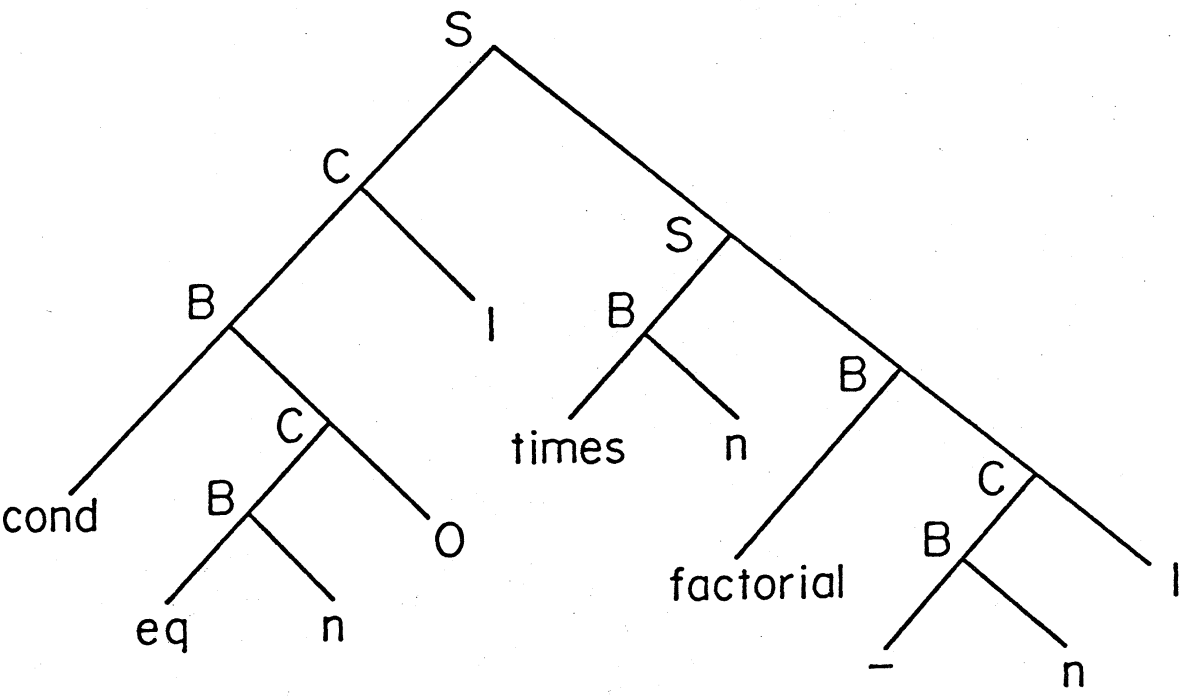
S
C
S
B
I
B
B
C
times
n
cond
B
C
B
O
factorial
I
eq
n
B
n

Fig. 7-b

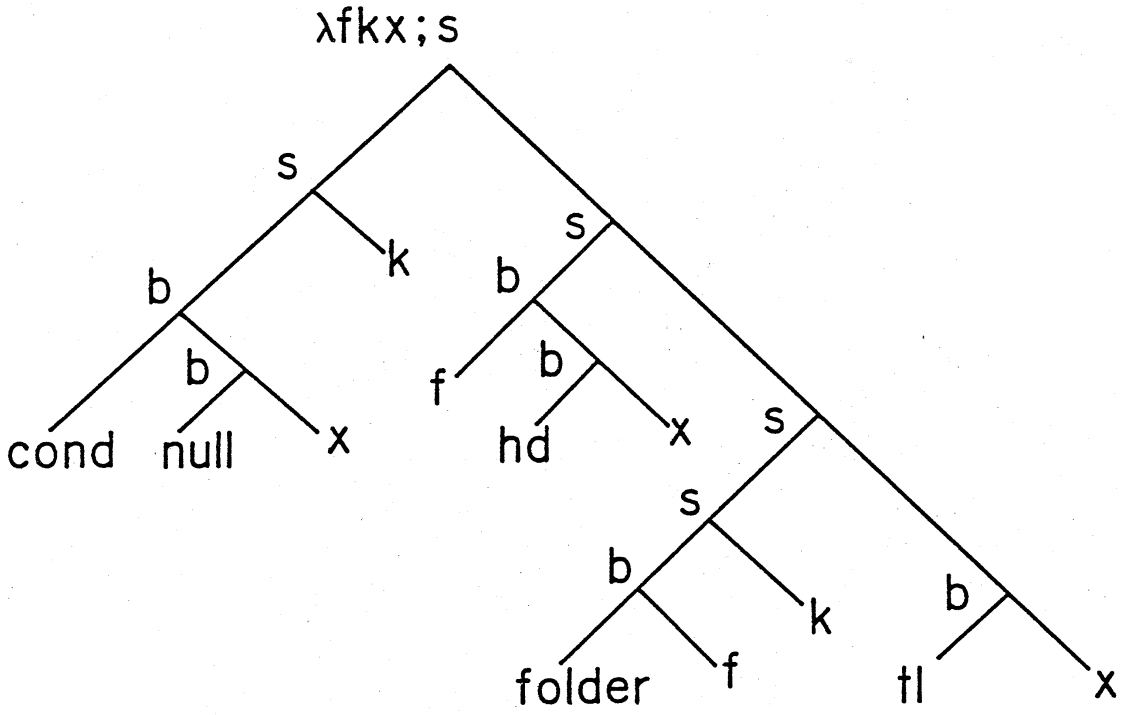
λfkx;s
s
k
b
b
cond
null
x
s
b
f
b
hd
x
s
s
b
k
b
folder
f
tl
x

Fig. 8-a

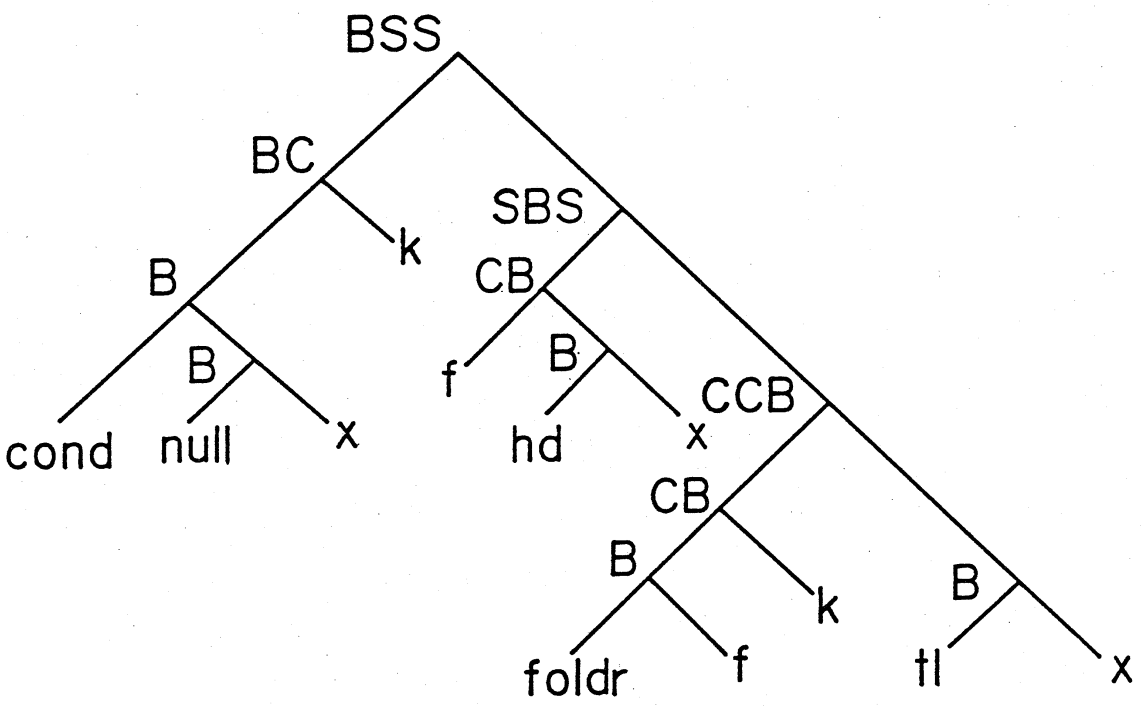
BSS
BC
k
SBS
B
CB
B
f
B
CCB
cond
null
x
hd
x
CB
B
k
B
foldr
f
tl
x

Fig. 8-b

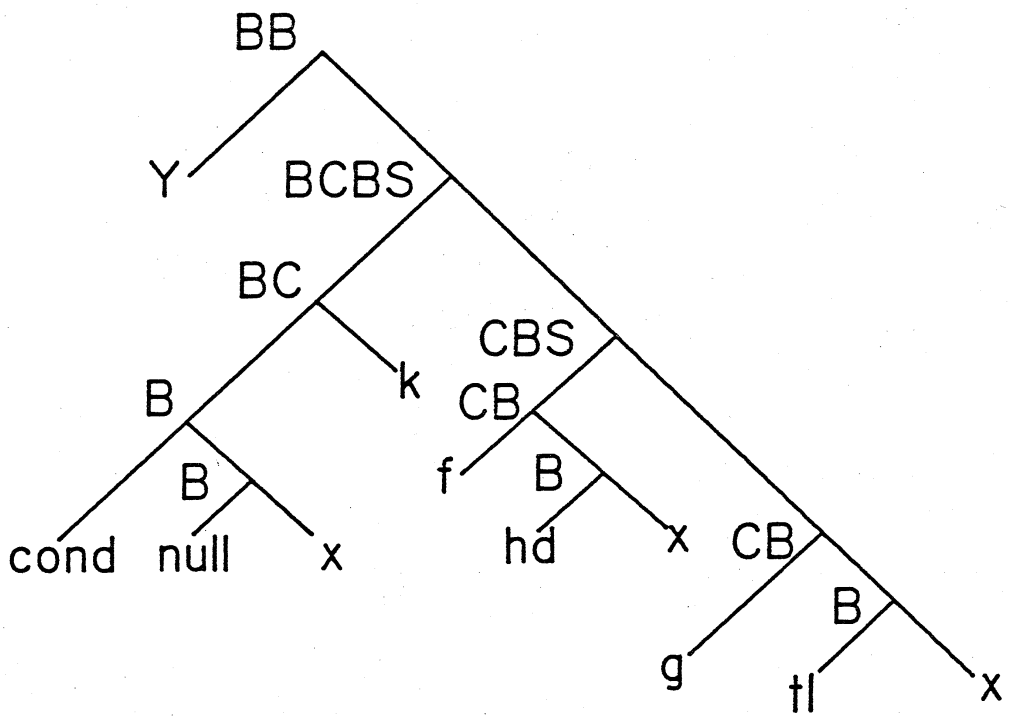
BB
Y
BCBS
BC
k
CBS
B
CB
B
f
B
cond
null
x
hd
x
CB
B
g
tl
x

Fig. 8-c

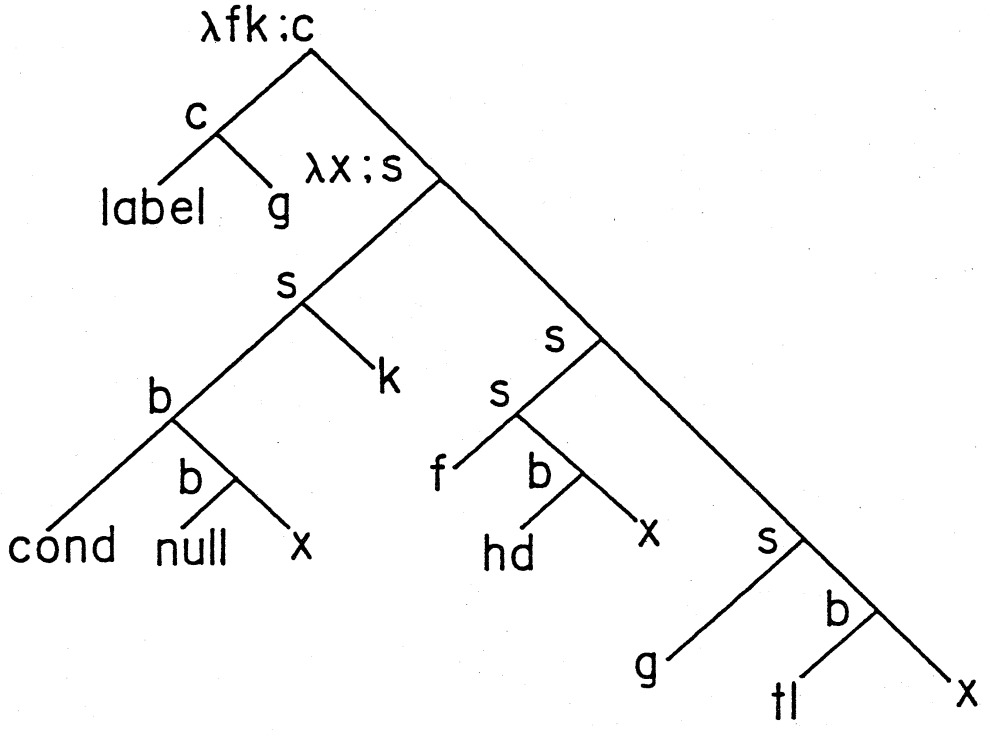
λfk:c
c
λx:s
label
g
s
k
s
b
s
b
f
b
cond
null
x
hd
x
s
b
g
tl
x

Fig. 8-d

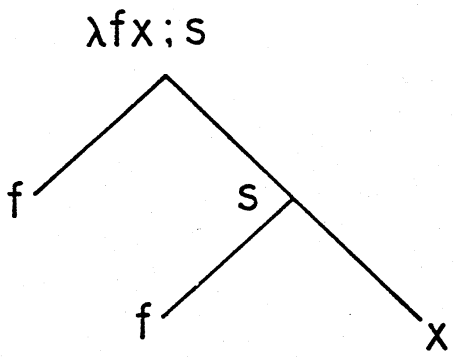

Fig. 9-a

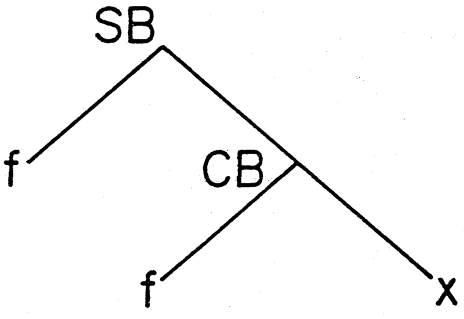

Fig. 9-b

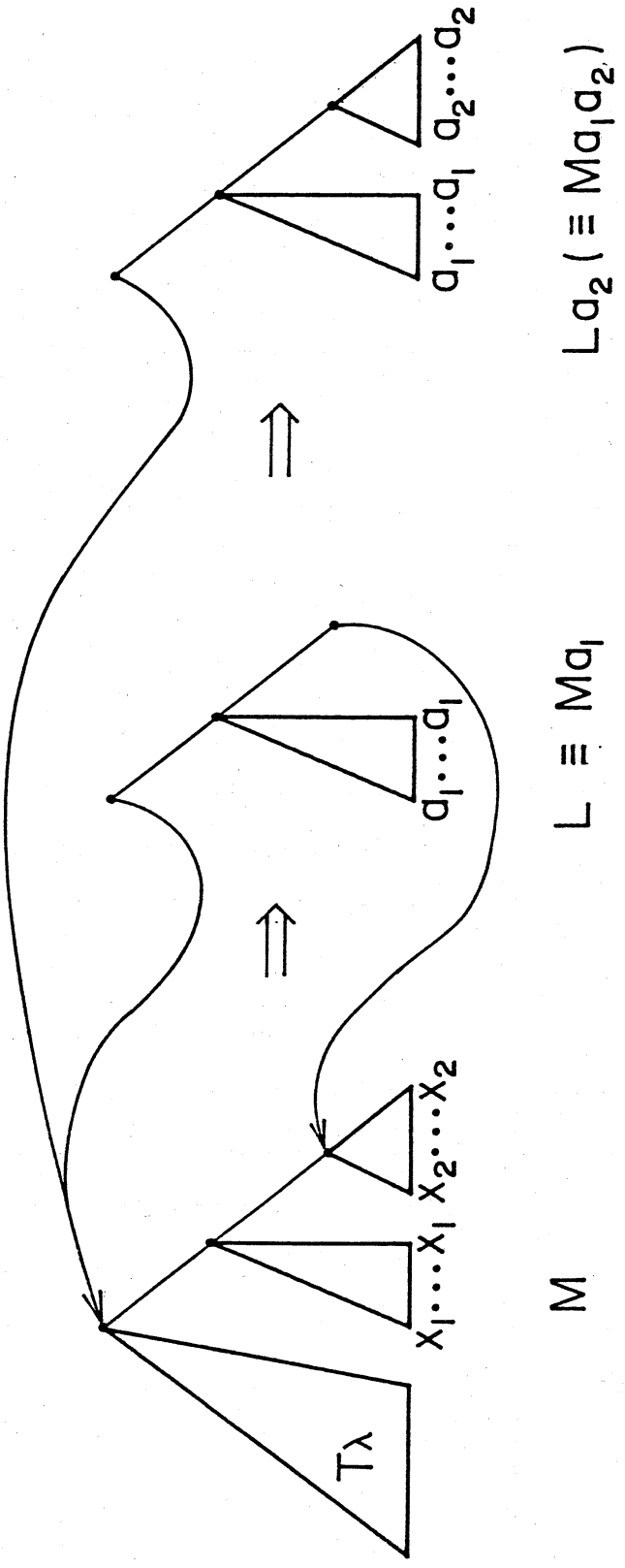
$a_1 \ldots a_1\ a_2 \ldots a_2$
$La_2\ (\equiv Ma_1a_2)$
$a_1 \ldots a_1$
$L \equiv Ma_1$
$x_1 \ldots x_1\ x_2 \ldots x_2$
M
$T\lambda$

Fig. 10