ON THE EQUIVALENCE OF
MECHANICAL EVALUATION
STRATEGIES

by

Clement McGowan

Technical Report
No. 69-37
July 1969

Department of Computer Science
Cornell University
Ithaca, New York 14850
ON THE EQUIVALENCE OF MECHANICAL EVALUATION STRATEGIES

by

Clement McGowan

ABSTRACT

Two mechanical evaluation strategies are shown to correctly implement the \( \lambda \)-\( K \)-calculus and to be equivalent.

* This research was supported in part by the National Science Foundation Grant CJ-103.
Landin [LC] was the first to demonstrate that the \( \lambda \)-calculus models some ALGOL-like features of programming languages as block structure with declaration and scope of identifiers, procedure definitions and calls (including recursive calls), parameter call by name and call by value. The \( \lambda \)-calculus has also played an important role in the formulation of the programming languages LISP, CPL, and PAL. Further, as Feldman and Gries [FG] note, it is "the most popular vehicle for attempting to formalize semantics of programming languages." This paper will treat the \( \lambda \)-calculus as a programming language and will investigate two distinctive implementation strategies.

Following Church [C] we define the calculus of \( \lambda \)-\( \varepsilon \)-conversion as a formal system having primitive symbols \( \lambda ( ) \) and an infinite list of symbols

\[ a, b, c, \ldots, x, y, z, a_1, b_1, \ldots, a_2, \ldots \]
called variables. A formula is any finite sequence of primitive symbols. Certain formulas are distinguished as being well-formed formulas (wff), and each occurrence of a variable in a wff is designated as free or bound according to the following recursive definition:

1. A variable \( x \) is a wff, and the occurrence of the variable \( x \) in this formula is free.

2. If \( P \) and \( A \) are wffs, then \( (PA) \) is a wff, and an occurrence of a variable \( y \) in \( P \) is free or bound in
(FA) according as it is free or bound in M, and an occurrence of a variable y in A is free or bound in (FA) according as it is free or bound in A.

3. If M is a wff, then $\lambda xM$ is a wff for any variable $x$, and an occurrence of a variable $y$, other than $x$, in $\lambda xM$ is free or bound in $\lambda xM$ according as it is free or bound in M. All occurrences of $x$ in $\lambda xM$ are bound.

4. A formula is a wff, and an occurrence of a variable in it is free or bound, only when this follows from 1., 2., 3.

The wffs give a syntactic specification of the $\lambda$-calculus as a language and can be rephrased in the more familiar BNF notation by:

$$<\text{\lambda-expression}> ::= \text{<variable>} | (\langle\text{\lambda-expression}\rangle\langle\text{\lambda-expression}\rangle) | \lambda<\text{variable}><\text{\lambda-expression}>$$

To give the semantics of this language we introduce the notation $S^x_M$ which shall stand for the formula which results by substitution of the formula M for the variable x throughout the formula M. And we define a part (of a formula) to mean consecutive well-formed subexpression not immediately following an occurrence of the symbol $\lambda$. We introduce now the two following conversion rules on wffs:

I. Let M be any part of a wff. Then if $x$ is a bound variable of $M$, $M$ can be replaced by $S^y_M$ provided $M$ contains no free occurrences of $x$, and $y$ does not occur in $M$.

II. The part $(\lambda xMA)$ can be replaced by $S^A_M$ provided $M$ contains no bound occurrences of $x$ and provided the wff A does not contain any free variables that are bound in M.
Rule I is called α-conversion or the Renaming Rule and
Rule II is referred to as β-reduction or the Reduction Rule.
A wff is said to be in normal or reduced form if it contains
no part of the form \((\lambda x M)\). The Renaming Rule since it is
its own inverse defines an equivalence relation on the set of
wffs and an equivalence class determined by this renaming re-
lation will be called a value class. Two wffs \(M\) and \(N\) from
the same value class are said to be equivalent up to renaming,
which will be denoted by \(M \equiv N\). If \(A\) is a wff and \(B\) results
from \(A\) by a single application of the Reduction Rule, then we
write \(A \rightarrow B\), and if \(M\) is in the same value class as \(A\) and
\(N\) is in the same value class as \(B\), then we write \(M \equiv N\); fur-
ther, if the conversion of \(A\) into \(B\) is obtained by reducing
the leftmost part of \(A\) of the form \((\lambda x CD)\) then we write \(A \rightarrow B\)
and \(M \equiv N\).

Let \(A = (L_0, L_1, \ldots)\) be a sequence of wffs such that \(L_1\)
not in reduced form and in \(A\) implies \(L_{i+1}\) is in \(A\) and
\(L_i \not\equiv L_{i+1}\). Then \(A\) is called the leftmost reduction sequence
of the wff \(L_0\). The specification of the \(\lambda\)-calculus as a pro-
gramming language is taken to be that given a wff \(L_0\) as the in-
puted program the leftmost reduction sequence of \(L_0\) is to be
computed. If this reduction sequence is finite, then the compu-
tation halts and has as the outputed result (a member of the value
class of) the normal form wff so obtained.

The mechanical evaluation strategies to be investigated are
designed to implement this leftmost reduction version of the
$\lambda$-calculus. Now in reducing a part ($\lambda x M A$) the pairing of $A$ with $x$ and its substitution for occurrences of $x$ in $M$ correspond to the definition and calling of a procedure in ALGOL-like languages. Section 4.7.3 of the Revised ALGOL Report [AR] states that the effect of a procedure call is equivalent to inserting a properly modified copy of the procedure body at the point of call. (Proper modification consists of actual-formal parameter correspondence and possible renaming of inserted identifiers to avoid conflicts.) However, the standard approach in practice is that of Randall and Russell [RR] where:

"instead of implementing the actual process of copying out and modifying the procedure body, a single copy is made of the object program representation of the procedure body, in which the use of a formal parameter is, in general, represented by a call on a subroutine generated from an actual parameter."

This contrast between the semantic specification of literal substitution and the actual implementation in terms of simulated substitution will be a distinguishing characteristic of the $\lambda$-calculus mechanisms investigated. For the principal "computation" of the $\lambda$-calculus is application of the Reduction Rule which semantically specifies the literal substitution of a wff for all occurrences of a bound variable within the body of the $\lambda$-expression constituting the scope. However, the implementation strategies considered effectively realize these substitutions through such information structures as stacks, pointers, counters, static links, etc. which are more
tractable computationally (i.e., when evaluation is to be performed mechanically as on a computer). In what follows the \( \lambda \)-calculus strategies will be described and the results relating to their correctness and equivalence will be given.

First, some notational conventions are introduced for use in defining these mechanisms. In the set of variables (i.e., those wffs having syntactic type \(<\text{variable}>\) ) the following infinite subsets are excluded from appearing in inputted wffs:

\[ \beta = \{ b_i \}_{i=1}^{\infty}, \quad \eta = \{ n_i \}_{i=1}^{\infty}, \quad \rho = \{ p_i \}_{i=1}^{\infty} \]

so that they may serve special purposes during computations. If \( X \) and \( Y \) are strings of symbols, then let \( X;:Y \) denote the concatenation of \( X \) with \( Y \). And if \( Z \) is \( X;:Y \), then \( X/Z \) is the string \( Y \). The standard \( \vdash \) will be used to indicate assignment.

The Basic Lambda-Calculus Machine, denoted BLM, was introduced by Wegner [W] and takes a pre-processed \( \lambda \)-expression as input. This pre-processing involves assigning unique names from the set \( \eta \) to the bound variables so that each binding expression introduces a new name \( n_i \). And for every part of the form \( \lambda<\text{variable}><\lambda\text{-expression}> \) a terminating \( \cdot \) is inserted following the \( \lambda\text{-expression} \). The \( \lambda \) serves as a begin symbol and a declaration of the \( <\text{variable}> \) which together constitute the blockhead, the \( \lambda\text{-expression} \) is the scope of the declaration (or the body of the block) and \( \cdot \) is the end symbol delimiter. A semi-colon is used as an end-marker for inputted \( \lambda \)-expressions. Effectively this adds the production \(<\text{program}>::=\langle\lambda\text{-expression}\rangle;\) and changes

\[ \langle\lambda\text{-expression}>::=\lambda<\text{variable}><\lambda\text{-expression}> \]

\[ \langle\lambda\text{-expression}>::=\lambda<\text{variable}><\lambda\text{-expression}>\cdot \]

in the BNF syntactic specification. The obvious modifications are to be made in the fore-
going definitions to account for these changes.

For example, pre-processing converts

\[(\lambda x\ y(x(y)))\ \lambda x\ \lambda z(z(x)))\]

into

\[(\lambda n_1\ \lambda n_2\ (n_1\ n_2))\ \cdots \lambda n_3\ \lambda n_4\ (n_3\ n_4))\]

and

\[(\lambda x(\lambda y(x(y)(x))))\]

into

\[(\lambda n_1\ (\lambda n_2\ n_1\ n_2)\ \cdots \lambda n_3\ n_4\ n_3\ n_4))\]

Members of the set \(\rho\) are used to denote pointers to parts during a ELCM computation. And \(p_1 = A\) will mean pointer \(p_1\) points to part \(A\) and \(p_1\) will signify the part pointed to by \(p_1\). (It is assumed that any replacement of a variable by a pointer during a computation will not increase storage requirements.) Members of the set \(\beta\) will be employed as the outputted bound variables. The meta-variables \(i\) and \(j\) are used as counters for selecting new members from \(\rho\) and \(\beta\) respectively in the course of a computation.

The ELCM makes use of three principal run-time information structures: a WORKSTACK, a left-parenthesis counter \(P\), and an OUTPUT_STREAM. In the subsequent formal definition WORKSTACK and OUTPUT_STREAM will be symbol string valued variables and \(P\), \(i\) and \(j\) will be integer valued variables but this is for ease of expression and WORKSTACK should be thought of as true stack into which pointers will be inserted and whose top (or leftmost) symbol configuration determines the next action of the ELCM and variable length modifications are performed at the top of the stack only.
DEFINITION OF THE BLM (explanatory comments will be enclosed within $'$s)

 Initialization:

\[ P := 0 \]
\[ i := 0 \]
\[ j := 0 \]

\[ \text{OUTPUT\_STREAM := } \]
\[ \text{WORKSTACK := input } \]

$\text{OUTPUT\_STREAM}$ is initially empty and \text{WORKSTACK} contains a pre-processed \( \lambda \)-expression as input $\$

Top of \text{WORKSTACK} \hspace{1cm} \text{Actions}

\[ ( \hspace{1cm} \text{WORKSTACK := } (/\text{WORKSTACK } \]
\[ P := P + 1 \]

\[ \downarrow \]
where \( v \) is some \( b_j \) in \( \beta \)
or some variable \( x \) not in \( \rho \)

\[ \text{WORKSTACK := } v/\text{WORKSTACK } \]
\[ \text{OUTPUT\_STREAM := } \text{OUTPUT\_STREAM} \downarrow (v \overbrace{\ldots (v}^{P\text{-times}} \)

$\text{P := 0 }$

$\text{If the top stack symbol is a bound variable from the set } \beta \text{ or a variable that was free in the inputted expression—i.e., not from } \rho \text{ or } \eta — \text{ then } P \text{ opening parentheses followed by the scanned variable symbol are added to the OUTPUT\_STREAM. } \$

An $\text{CASE I: } P > 0$

\[ \text{WORKSTACK := } \lambda_{n M \cdot A}/\text{WORKSTACK } \]
\[ i := i + 1 \]

\[ \text{WORKSTACK := } S_{n M \cdot i} \text{ WORKSTACK } \]
\[ P_1 = A \]

\[ P := P - 1 \]
M and A are well-formed parts which must follow \( \lambda n \) since the input is by assumption well-formed and \( P > 0 \); in the original formulation of the BLM the pointer \( p_1 \) which replaces all instances of \( n \) in the expression \( M \) pointed to the beginning in the stack of the part A and the \( o \) matching the scanned \( \lambda n \) was replaced by a transfer pointer pointing to the stack location immediately following the closing parenthesis of the combination \( \lambda n M \cdot A \) at the top of the WORKSTACK. Here the same pointer substitution is performed but the symbol sequence \( \cdot A \) is deleted to model the transfer pointer with the pointer information retained by \( p_1 = A \). The present equivalent approach is for formal convenience only.

CASE II : \( \ P = 0 \)

\[
\begin{align*}
\text{WORKSTACK} & := \lambda n M \cdot / \text{WORKSTACK} \\
\ j & := j + 1 \\
\text{WORKSTACK} & := \varepsilon^j M \cdot : : \text{WORKSTACK} \\
\text{OUTPUT\_STREAM} & := \text{OUTPUT\_STREAM} :: \lambda b_j
\end{align*}
\]

\( M \) is a part due to the original well-formedness of the input and the \( o \) matches the scanned \( \lambda n \).

\[
\begin{align*}
p_1
\end{align*}
\]

where \( p_1 \) is in \( \rho \)

\[
\begin{align*}
\text{WORKSTACK} & := p_1 / \text{WORKSTACK} \\
\text{WORKSTACK} & := E_1 :: \text{WORKSTACK}
\end{align*}
\]

$ recall that \( E_1 \) is such that \( p_1 = E_1 $

[transfer pointer]

$ does not occur by earlier remarks; in original BLM the stack is popped until the position pointed to is reached $
where σ is } or [ or ; OUTPUT_STREAM := OUTPUT_STREAM ++ σ

Note that if } is scanned, then after the actions are performed the WORKSTACK will be empty and so the computation will terminate. 

We illustrate the operation of the ECOM with a computation on the λ-expression \((λf(λf) λg λx(gx)) h\) which interestingly models the recursive call of a procedure and requires an application of the Renaming Rule in its reduction sequence despite there being no initial naming conflicts. One might think of the above λ-expression as corresponding to the ALGOL-like program:

```
begin
procedure f(g,x) ; g(x) ; f(f,h) ;
end
```

whose computation results in the function \(h(\ ))\ or, more precisely, \(λx h(x)\). Now pre-processing transforms \((λf(λf) λg λx(gx)) h\) into 
\((λn_1(n_1n_1) λn_2λn_3(n_2n_3) e) h\) ;

Example ECOM Computation :

<table>
<thead>
<tr>
<th>STEP</th>
<th>P</th>
<th>WORKSTACK</th>
<th>OUTPUT_STREAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>((λn_1(n_1n_1) λn_2λn_3(n_2n_3) e) h)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>((λn_1(n_1n_1) λn_2λn_3(n_2n_3) e) h)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>(λn_1(n_1n_1) λn_2λn_3(n_2n_3) e) h)</td>
<td></td>
</tr>
<tr>
<td>STEP</td>
<td>P</td>
<td>WORKSTACK</td>
<td>OUTPUT_STREAM</td>
</tr>
<tr>
<td>------</td>
<td>----</td>
<td>-----------</td>
<td>---------------</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>(p_2p_3 h)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>where ( p_1 = \lambda n_2 \lambda n_3 (n_2n_3) )</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>(p_2p_1) h</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>( \lambda n_2 \lambda n_3 (n_2n_3) ) ( \lambda p_1 ) h</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>( \lambda n_3 (p_2n_3) ) h</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>where ( p_2 = p_1 )</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>(p_2p_3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>where ( p_3 = h )</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>P_2P_3</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>P_1P_3</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>( \lambda n_2 \lambda n_3 (n_2n_3) ) ( \lambda p_3 )</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>( \lambda n_3 (p_3n_3) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>where ( p_4 = p_3 )</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>(p_4b_1)</td>
<td>( \lambda b_1 )</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>p_4b_1</td>
<td>( \lambda b_1 )</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>p_3b_1</td>
<td>( \lambda b_1 )</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>h b_1</td>
<td>( \lambda b_1 )</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>b_1</td>
<td>( \lambda b_1 (h) )</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>( )</td>
<td>( \lambda b_1 (h b_1) )</td>
</tr>
<tr>
<td>18</td>
<td>0</td>
<td>( )</td>
<td>( \lambda b_1 (h b_1) )</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
<td>( )</td>
<td>( \lambda b_1 (h b_1) )</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>( )</td>
<td>( \lambda b_1 (h b_1) )</td>
</tr>
</tbody>
</table>

Computation halts with result (or output): \( \lambda b_1 (h b_1) \), since the WORKSTACK is now empty.
Thus the BLCM implements procedure definition by inserting pointers to the procedure body at the points of call. And procedure calls involve placing a copy of the procedure in the calling environment. What distinguishes the BLCM operation from the literal substitution semantically specified by the Reduction Rule for the λ-calculus is that replacement by pointers is a fixed length substitution which does not increase the stack storage requirements. Further, the only variable length operations of the BLCM are performed at the top of the WORKSTACK.

In proving that the BLCM does indeed properly implement the λ-calculus the following results are useful.

**Lemma 1:** For any input pre-processed λ-expression, at each step of the computation the WORKSTACK of the BLCM contains no part of the form \( \lambda n \, M \) where \( M \) is a λ-expression and \( n \) is bound in \( M \).

**Proof:** (By induction on the computational steps)

**Step 0:** The WORKSTACK contains the input λ-expression which has been pre-processed by introducing a new name from the set \( \eta \) for each binding expression and applying the Renaming Rule. So the conditions of the Lemma are met.

**Inductive Assumption:** Assume for all \( i < j \) that at computational step \( i \) the WORKSTACK contains no part of the form \( \lambda n \, M \) with \( M \) being a λ-expression and \( n \) bound in \( M \).

If the computation terminates after \( k \) steps and \( k < j \), then by assumption the Lemma holds. If after \( j \) steps the computation has not halted (i.e., the WORKSTACK is not empty), then consider all possible symbol configurations at the top of the WORKSTACK.
\[
\begin{align*}
\{ \text{a variable not in \( \rho \)} \} \\
\} \\
\} \\
\} \\
\{ \text{all result in deletions from the WORKSTACK and nothing is added or inserted} \}
\end{align*}
\]

\[ P_1 \text{ (from } \rho \text{)} \]

Is replaced by \( E_1 \) where \( p_1 = E_1 \).

Since \( E_1 \) is a well-formed part there is no need to consider what then follows it on the WORKSTACK. Now the correspondence \( p_1 = E_1 \) arose at an earlier computational Step so \( E_1 \) satisfies the inductive assumption.

\[ \lambda_M \text{ if } P > 0 \text{, then } \lambda_M M_\cdot A \] is replaced at the top of the WORKSTACK by \( S^n M_\cdot p_1 \) where \( M \) and \( A \) are well-formed \( \lambda \)-expressions and \( p_1 \) is a new pointer variable from \( \rho \).

By the inductive assumption all instances of \( n \) in \( M \) are free; hence all instances of \( p_1 \) in \( S^n M_\cdot p_1 \) are free since \( p_1 \) is a newly created variable and since \( S^n M_\cdot p_1 \) is a well-formed part the conditions of the Lemma are met.

The argument for \( \lambda_M \) and \( P = 0 \) is similar.

Thus in any case the WORKSTACK at Computational Step \( j + 1 \) contains no part of the form \( \lambda_M M \cdot A \) where \( M \) is a \( \lambda \)-expression and \( n \) is bound in \( M \).

Q.E.D.
addition to some of the originally inputed symbols. The total state of the BLCM at Computational Step i, denoted $T_i$, is given by the OUTPUT_STREAM, $P$, and WORKSTACK at Step i.

In what follows extensive use will be made of a symbol string $B(i)$ obtained from a mapping of the total state $T_i$.

DEFINITION: For any computation by the BLCM on an inputed pre-processed $\lambda$-expression, the Basic Normal Form at Computational Step i, denoted $B(i)$, is given by

$$B(i) := OUTPUT\_STREAM \cdot \{ \ldots \cdot \cdot \text{trace}[WORKSTACK] \cdot \cdot \cdot \}$

P-times

where OUTPUT_STREAM, $P$, and WORKSTACK comprise $T_i$ and

$$\text{trace}[SMSTRING] =$$

- head[SMSTRING] not in $\rho$ or null
- $\text{head}[SMSTRING] := \text{trace}[\text{tail}[SMSTRING]]$
- $\text{head}[SMSTRING]$ is $p_j$ in $\rho$ (for some $j$) and $p_j = E_j$
- $\text{trace}[E_j] := \text{trace}[\text{tail}[SMSTRING]]$
- head[SMSTRING] is null

Here the function trace is defined by a LISP-like conditional statement where head and tail are used as selectors on the WORKSTACK treated as a symbol string list and correspond to the respective list selectors car and cdr of LISP.

Intuitively $B(i)$ is formed by concatenating the $T_i$ values of the OUTPUT_STREAM with $P$ opening parentheses and then with
the trace of the WORKSTACK, where the trace simply copies symbols unless a substitution pointer is encountered in which case the part pointed to is traced at the location of the pointer (i.e., at the point of call).

For example, consider the above BLCM computation at STEP 13:

\[ B(13) \leftarrow \lambda b_1 :: \left( :: \text{trace} \left( p_4 b_1 \right) . ; \right) \]

and \( \text{trace} \left( p_4 b_1 \right) . ; \) = \( \text{trace} \left( p_3 \right) :: \text{trace} \left( b_1 \right) . ; \)

= \( \text{trace} \left( h \right) :: b_1 :: \text{trace} \left( . \right) . ; \)

= \( h \) \( s_1 : b_1 :: \) \( \text{trace} \left( . \right) . \)

= \( h \) \( b_1 :: \) \( \text{trace} \left( . \right) \)

= \( h \) \( b_1 \).

So \( B(13) \leftarrow \lambda b_1 (h b_1) . ; \)

Note that \( B(13) \) is a \( \lambda \)-expression (more precisely, \( B(13) \) is a properly formed program since it terminates with a semi-colon, but "program" and "\( \lambda \)-expression" are here used interchangeably).

If a BLCM computation terminates in \( k \) steps, then the Basic Normal Form \( B(i) \) is extended to values of \( i > k \) by the definition \( B(i) \leftarrow B(k) \) if \( i > k \).

If \( C \) and \( D \) are symbol strings, then by \( C \triangleright D \) we mean \( C \) and \( D \) are symbol for symbol equivalent, that is, they consist of the same finite sequence of symbols.

LEMMA 2: Let \( M \) and \( N \) be \( \lambda \)-expressions from \( \left( \ldots \left( :: \text{WORKSTACK} \right. \right) \) \( \triangleright \) \( \text{P-times} \)

of a BLCM computation. Then
(I) \( M \equiv N \implies \text{trace}(M) \equiv \text{trace}(N) \)

(II) \( M \not\equiv N \implies \text{trace}(M) + \text{trace}(N) \)

**Proof:** (Sketch)

(I) The function trace merely replaces variables from \( \rho \) with \( \lambda \)-expressions and by COROLLARY 1 all variables from \( \rho \) are free in \( iM \) and \( N \) and so they are not affected by any sequence of renamings leading from \( M \) to \( N \). Now rename the bound variables in each expression \( E_i \) pointed to by a pointer variable \( p_i \) (to insure disjointness). Apply the same renaming sequence which lead from \( M \) to \( N \) to \( \text{trace}(M) \) and then rename the bound variables back to reobtain the \( E_i \)'s. The reader can easily convince himself that this renaming sequence properly leads from \( \text{trace}(M) \) to \( \text{trace}(N) \). Hence, \( \text{trace}(M) \equiv \text{trace}(N) \).

(II) If \( M \not\equiv N \) then there is a leftmost part of the form \((\lambda n \ C \cdot D)\) which is replaced by \( S^n_{D,C} \) which by LEMMA 1 and COROLLARY 1 is a proper application of the Reduction Rule. Now \( \text{trace}(\lambda n \ C \cdot D) \equiv (\lambda n \ \text{trace}(C) \cdot \text{trace}(D)) \).

Let \( C' \) be \( \text{trace}(C) \) with the bound instances of \( n \) renamed to some newly created variable \( n' \) from \( \eta \). No free variables in \( \text{trace}(D) \) are from \( \eta \) and the only bound variables in \( C' \) are from \( \eta \) and \( n \) is not bound in \( C' \).

Therefore, \( \text{trace}(\lambda n \ C.D) \equiv (\lambda n \ \text{trace}(C) \cdot \text{trace}(D)) \equiv (\lambda n \ C' \cdot \text{trace}(D)) \).

\( S^n_{\text{trace}(D)C'} \equiv \text{trace}(S^n_{\text{trace}(D)C}) \equiv \text{trace}(S^n_{D,C}) \)
So since \( M \vdash N \) by the part \((\lambda n \ C \cdot D) \Psi D_n\) we conclude that \( \text{trace}(M) + \text{trace}(N) \).

Q.E.D.

The principal result on the Basic Normal Form can now be given.

**THEOREM 1:** For any computation by the BLCM on an inputed pre-processed \( \lambda \)-expression the following hold for all \( i \):

(I) \( B(i) \) is a \( \lambda \)-expression

(II) If at Step \( i \), \( \lambda n \) is at the top of the WORKSTACK where \( n \) is in \( \eta \) and \( P > 0 \), then \( B(i) \uparrow B(i+1) \)

otherwise \( B(i) \not\equiv B(i+1) \).

**PROOF:** (By induction on the Computational Steps)

**STEP 0** Initially \( P = 0 \), the OUTPUT_STREAM is empty, and the WORKSTACK is the inputed pre-processed \( \lambda \)-expression. Therefore, by definition \( B(0) \) is the inputed \( \lambda \)-expression and so (I) is satisfied. Since \( B(0) \) is well-formed the only possible symbol configurations at the top of the WORKSTACK are \( ( \) or \( \lambda n \) with \( n \) in \( \eta \) or a variable \( v \) not in \( \eta \), \( \rho \), or \( \beta \) and all with \( P = 0 \).

\( (\) is deleted from the top of the WORKSTACK and \( P \) is set to \( 1 \); so by definition \( B(1) \) is \( B(0) \).

\( v \) is deleted from the top of the WORKSTACK, zero opening parentheses followed by \( v \) are placed in the OUTPUT_STREAM and so again by definition \( B(1) \) is \( B(0) \).
Since \( B(0) \) is well-formed the WORKSTACK must have the form \( \lambda n \, M \), where \( M \) is a \( \lambda \)-expression. \( \lambda n \) is deleted, \( \lambda b_1 \) is placed in the OUTPUT_STREAM and \( M \) is replaced by \( S^n_{b_1} M \). Therefore, \( B(1) = \lambda b_1 \, S^n_{b_1} M \), and since all instances of \( n \) are bound in \( \lambda n \, M \) and \( b_1 \) is a newly created variable \( B(1) \) results from \( B(0) \) by a proper application of the Renaming Rule.

Thus, in any event \( B(0) \equiv B(1) \) and (II) is satisfied.

**Inductive Assumption**: Assume for all \( i < k \) that (I) and (II) of the THEOREM hold.

If the BLCM computation terminates in \( m \) steps and \( m < k \) then for \( i < m \) \( B(i) \) is a \( \lambda \)-expression by assumption and for \( i > m \) \( B(i) := B(m) \) by definition. So, condition (I) is met. And (II) is satisfied by assumption for \( i < m \). Now the WORKSTACK is empty at Step \( m \) and since \( B(m) \) is well-formed \( P = 0 \) for \( P > 0 \) would imply that the rightmost symbol of \( B(m) \) would be a '(''. Hence for all \( i > m \) \( P = 0 \) and \( B(i) := B(m) \), so \( B(i) \equiv B(i+1) \) and (II) holds for all \( i \).

It remains to consider those BLCM computations which have not halted by Step \( k \) (i.e., have a non-empty WORKSTACK). The proof now proceeds by cases considering the possible symbol configurations at the top of the WORKSTACK.

'( ' is deleted from the WORKSTACK, \( P \) incremented by 1 which implies that \( B(k+1) \) is \( B(k) \) by definition of the Basic Normal Form.
By the inductive assumption where $\sigma$ is ), } or $\epsilon$ or $;$

Therefore, $P = 0$ else we would have one of the symbol sequences () or $\epsilon$ or $;$ appearing in $B(k)$.

Now $\sigma$ is deleted from the WORKSTACK and added to the OUTPUT_STREAM while $P$ remains zero. So again $B(k+1)$ is $B(k)$.

Variable $v$ not in $\eta$ $v$ is deleted from the WORKSTACK, $P$ opening parentheses followed by $v$ are placed in the OUTPUT_STREAM and $P := 0$. Again by definition of the Basic Normal Form $B(k+1)$ is $B(k)$.

$p_j$ where $p_j = E_j$ $p_j$ is replaced at the top of the WORKSTACK by the $\lambda$-expression $E_j$. By definition $B(k)$ is constituted by performing such substitutions and so $B(k+1)$ is $B(k)$.

$\lambda n$ and $P = 0$ Since $B(k)$ is well-formed $\lambda n$ will be followed on the WORKSTACK by a $\lambda$-expression $M$ and the matching $\epsilon$. $\lambda n$ is deleted, $\lambda b_j$ is placed in the OUTPUT_STREAM where $b_j$ is a newly created variable and $M$ is replaced on the top of the WORKSTACK by $\lambda b_j^n M$. So, $B(k)$ differs from $B(k+1)$ only in that $\lambda n \text{trace}(M)$ is replaced by $\lambda b_j \text{trace}(\lambda b_j^n M)$.
But by definition \( \lambda n \text{trace}[M] \in \text{trace}[\lambda n M \cdot] \) and 
\( \lambda b_j \text{trace}[S^n_{b_j} M] \in \text{trace}[\lambda b_j S^n_{b_j} M \cdot] \) 
\( \in \text{trace}[S^n_{b_j} \lambda n M \cdot] \).

Now \( S^n_{b_j} \lambda n M \cdot \equiv \lambda n M \cdot \) since it is a proper application

of the Renaming Rule for \( n \) is bound in \( \lambda n M \cdot \) and \( b_j \)
is a new variable and hence not in \( M \). And as argued
above \( B(k) \) differs from \( B(k+1) \) only in that equivalently

\( \text{trace}[\lambda n M \cdot] \) is replaced by \( \text{trace}[S^n_{b_j} \lambda n M \cdot] \). So

by LEMMA 2 \( B(k) \neq B(k+1) \) as in all of the above cases.

\( \lambda n \) and \( P > 0 \) The assumed well-formedness

of \( B(k) \) and \( P > 0 \)

implies that \( \lambda n \) is followed on the WORKSTACK by \( M \cdot A \)
where \( M \) and \( A \) are \( \lambda \)-expressions. A newly created point-
er variable \( p_j \) is introduced and \( p_j = A \) and 
\( \lambda n M \cdot A \) is replaced on the top of the WORKSTACK by

\( S^n_{p_j} M \) and \( P \) is decremented by 1. So \( B(k) \) differs

from \( B(k+1) \) only in that \( (\lambda n \text{trace}[M] \cdot \text{trace}[A \cdot]) \)
is replaced by \( \text{trace}[S^n_{p_j} M] \). But by definition

\( (\lambda n \text{trace}[M] \cdot \text{trace}[A \cdot]) \in \text{trace}[\lambda n M \cdot A \cdot] \)
and \( \text{trace}[S^n_{p_j} M] \in \text{trace}[S^n_{A} M] \) for \( p_j = A \)
hence \( \text{trace}[p_j] = A \). Now \( (\lambda n M \cdot A \cdot) \in S^n_{A} M \)
since

by LEMMA 1 and COROLLARY 1 \( n \) is not bound in \( M \) and
there is no variable \( y \) free in \( A \) but bound in \( M \).

Therefore, by LEMMA 2 \( \text{trace}[\lambda n M \cdot A \cdot] \in \text{trace}[S^n_{A} M] \)
Now $\text{trace}((\lambda n \; M \cdot A))$ or equivalently $(\lambda n \; \text{trace}[M] \cdot \text{trace}[A])$ is the leftmost part of $B(k)$ of the form $(\lambda x \; C \cdot D)$ since by definition of the BLCH an opening parenthesis never precedes a $\lambda b_j$ in the OUTPUT_STREAM. Therefore, we conclude that $B(k) \not\in B(k+1)$.

So condition (II) is satisfied for STEP $k+1$ and this in turn implies condition (I).

Q.E.D.

A direct consequence of THEOREM 1 is:

COROLLARY 2: For all $i$ the following holds:

(I) $B(i)$ not in reduced form $\iff (\exists m)$ such that $B(i) \not\in B(i+m)$

(II) $B(i)$ in reduced form $\iff \forall m \geq i \; B(i) \in B(m)$

PROOF: (Sketch)

(I) $\iff$ by definition of reduced form

$\Rightarrow$ OUTPUT_STREAM never contains a part of the form $(\lambda n \; M \cdot A)$ so there must be a leftmost one in the WORKSTACK by definition of the Basic Normal Form. It is then straightforward to argue from the well-formedness of $B(i)$ that the condition $\lambda n$ at the stack top with $P \geq 0$ must eventually arise after some $m$ steps in the BLCH computation and then apply THEOREM 1.

(II) immediate from (I).

Q.E.D.

One further result is needed prior to giving the major theorem concerning the BLCH.
COROLLARY 3: If $B(i)$ is in reduced form, then for some $m$ the corresponding BLCM computation terminates in $m$ steps and $B(m)$ is the OUTPUT_STREAM.

PROOF: (Informally given; to formalize one would show by induction that the WORKSTACK is eventually empty.)

If the WORKSTACK is empty at Step 1, then the computation has already terminated so just consider the case where $B(1)$ is in reduced form and the WORKSTACK is non-empty. Now by COROLLARY 2 $\forall m \geq 1 B(i) \equiv B(m)$ and so by THEOREM 1 for no $m \geq 1$ will a $\lambda n$ be at the top of the WORKSTACK with $P > 0$. It remains to show that under these circumstances the WORKSTACK will eventually be empty, but all operations now permissible reduce the WORKSTACK with the exception of the substitution of an expression $E_j$ for a pointer variable $P_j$. But substitution eventually terminates leaving no further variables from $P$ on the WORKSTACK as can be seen by the definition of the trace function and the fact (I) of THEOREM 1. Therefore, the WORKSTACK will eventually be empty and so the BLCM computation must terminate after some finite number of steps, say $m$. Clearly $P$ will be 0 since $B(m)$ is a wff by THEOREM 1 and so $B(m)$ is by definition the OUTPUT_STREAM.

Q.E.D.

The above results show that with each Computational Step a highly structured and significant symbol string can be corresponded to the run-time information structures generated by an
interpretive mechanism. Clearly this provides a handle for proving properties dependent upon execution since the total state has been compressed into a manageable form. Specifically we now show that the BLCM correctly implements the λ-K-calculus where by "correctly implements" we mean for any pre-processed λ-expression $L_0$ and any sequence of applications of the Reduction and Renaming Rules which leads to a reduced form λ-expression $L_m$ that the BLCM with input $L_0$ halts after finitely many Computational Steps with $L_m$ as output (i.e., constituting the OUTPUT_STREAM).

**THEOREM 2**: Let $L_0$ be a pre-processed λ-expression and

$$A = \{ L_0, L_1, \ldots \}$$

the leftmost reduction sequence of $L_0$. If $L_0$ is inputted to the BLCM then for all $i$

$$(*) \quad L_i \in A \Rightarrow (\exists j)(B(j) \equiv L_i)$$

**PROOF**: (By induction on $i$)

$i = 0$. $B(0)$ is $L_0$ hence $B(0) \equiv L_0$

Inductive Assumption: Assume for all $1 \leq k$ that $(*)$ holds.

By assumption there exists a $j$ such that $B(j) \equiv L_k$. If $L_k$ is in reduced form, then by definition of $A$ we are done. So suppose $L_k$ is not in reduced form then $L_k \nmid L_{k+1}$ and $L_{k+1}$ is in $A$. Also $B(j)$ is not in reduced form and by COROLLARY 2 there exists an $m$ such that $B(j) \nmid B(j+m)$.

Now $B(j) \equiv L_k$ implies that $B(j+m) \equiv L_{k+1}$ since for any λ-expression not in reduced form the leftmost part of the form $(\lambda n. C . D)$ is uniquely determined.

Q.E.D.
COROLLARY 4: The BLCM correctly implements the \( \lambda \)-calculus.

PROOF: By THEOREM 2 and COROLLARY 3 the BLCM correctly implements the leftmost reduction sequence of any \( \lambda \)-expression. (Recall that pre-processing is a special case of applying the Renaming Rule.) Now by a theorem cited in [W], which is based upon the Church-Rosser Theorem if any reduction sequence of a \( \lambda \)-expression \( L \) results in a reduced form then it will be equivalent up to renaming to the result obtained by the leftmost reduction sequence of \( L \).

Q.E.D.

This fundamental fact that the BLCM correctly implements the \( \lambda \)-calculus can now be used as a basis for investigating and even constructing another mechanism for realizing the \( \lambda \)-calculus. However, as a model for the implementation of procedure evaluation in ALGOL-like programming languages the BLCM is not particularly realistic since its programs are self-modifying and each procedure call gives rise to another copy of the procedure being created. With this in mind, the author has constructed a fixed-program machine, henceforth denoted by FPM, which as its name implies, treats an inputted program as "pure procedure" (or reentrant code). The FPM, differing markedly from one proposed by Wegner [W], is in fact inspired by the BLCM and its known correctness.

The FPM takes a pre-processed \( \lambda \)-expression as input and loads it into a control area. Certain information about the program is gathered prior to execution and stored with the
appropriate instruction (i.e., symbol configuration). Specifically, the control location of a right parenthesis is stored with its matching left parenthesis, the control location of a terminating \( \lambda n \) and also stored with each declaration is the location of the immediate statically enclosing declaration (if any). For example, the pre-processed \( \lambda \)-expression

\[
((\lambda n_1(n_1 n_1) \cdot \lambda n_2 \lambda n_3(n_2 n_3) \cdot \cdot ) \ h ) \]

when loaded into the control area and pre-execution information is gathered has the form
<table>
<thead>
<tr>
<th>CONTROL LOCATION</th>
<th>INSTRUCTION</th>
<th>MATCH</th>
<th>STATIC LINK</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>(</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>\ln_1</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>(</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>n_1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>n_1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>\ln_2</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>\ln_3</td>
<td>15</td>
<td>9</td>
</tr>
<tr>
<td>11</td>
<td>(</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>n_2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>n_3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

where STATIC.LINK 0 denotes that there is no statically enclosing declaration.
In addition to the CONTROL, the information structures employed by the FPM are: an activation record stack AR for the name-value pairings entailed by declarations and actual-formal parameter correspondences, a return stack R used for parameter passing and return linkage, an instruction pointer IP indicating the CONTROL location of the present instruction, a current environment pointer ENVR Designating the current activation record in AR, a pointer TOP to the top of the AR stack, a left parenthesis counter P, an OUTPUT stream and a counter j for generating newly created variables from $\beta$ for output purposes (ARG, FIND.V, S.LNK and RTRNS are used as temporary variables and GET.ARG, GET.LNK, LOOK.UP, RET.CHECK are subroutines in the definition of the FPM).

In defining the FPM information structures composed of several fields will be used. Landin [LM] calls such structures constructed objects and the field names when applied to the objects act as selectors. For example, if object Ob has the form $| F_1 \ F_2 \ F_3 |$ where the $F_i$'s are field names and we assign Ob the value $| v_1 \ v_2 \ v_3 |$ then $F_i(\text{Ob}) = v_i$ for $i = 1,2,3$. This use of field names as selectors is similar to the programmer-defined data types of SNOBOL4. The i-th entry in the AR stack will be denoted by AR<i>.

The next step in a computation of the FPM is determined by the contents of the CONTROL location designated by the current value of the instruction pointer IP. We specify the FPM
by giving the actions for each of the possible forms the contents of a CONTROL location may take.

FORMAL DEFINITION OF THE FPM

Initialization:

AR :=
R :=
IP ≥ 1
ENVR := 0
TOP := 0
P := 0
OUTPUT :=
j := 0

Contents of current CONTROL location

<table>
<thead>
<tr>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>P := P + 1</td>
</tr>
<tr>
<td>IP := IP + 1</td>
</tr>
</tbody>
</table>

An MATCH STATIC.LINK

<table>
<thead>
<tr>
<th>Case I : P &gt; 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOP := TOP + 1</td>
</tr>
<tr>
<td>ENVR := TOP</td>
</tr>
<tr>
<td>S.LNK := GET.LNK(STATIC.LINK, TOP)</td>
</tr>
<tr>
<td>ARG := GET.ARG(MATCH)</td>
</tr>
</tbody>
</table>
ARG will then have the form
| BEGIN END ENVIRONMENT |
Stack as AR< TOP >
| n BEGIN(ARG) END(ARG) ENVIRONMENT(ARG) S.LNK |
Stack on 'R
| MATCH [END(ARG) + 2] ENVIRONMENT(ARG) |

This corresponds to the transfer pointer of the
BLCM

\[ P := P - 1 \]
\[ IP := IP + 1 \]

Case II : \[ P = 0 \]
\[ TOP := TOP + 1 \]
\[ ENVR := TOP \]
\[ S.LNK := GET.LNK(STATIC.LINK, TOP) \]
\[ j := j + 1 \]

Stack as AR<TOP>
| n b_j 'constant' S.LNK |
\[ OUTPUT := OUTPUT :: lb_j \]
\[ IP := IP + 1 \]

where \( x \) is a variable

\[ FIND.V := \text{LOOK.UP}(x, ENVR) \]

Case I : FIND.V is a variable

\[ OUTPUT := OUTPUT :: \ldots \ldots :: \text{FIND.V} \]
P-times
\[ P := 0 \]
IP := IP + 1

Case II: (x gives rise to a call-by-name)
FIND.V (obtained from some AR-entry) has the form

| BEGIN   END ENVIRONMENT STATIC.LINK |

On R stack (return linkage)

| [END(FIND.V)+1] [IP + 1] ENVR |

$ Transfer of control and environment adjustment which follow are analogous to the substitution of an expression for a pointer in the BLCM $

ENVR := ENVIRONMENT(FIND.V)
IP := BEGIN(FIND.V)

$s$ where $s$ is $) or $s$

RTRNS := RET.CHECK(IP,ENVR,R)

Case I: No returns made
OUTPUT := OUTPUT :: s
IP := IP + 1

Case II: Return made
Then RTRNS has the form

| LOCATION ENVIRONMENT |

ENVR := ENVIRONMENT(RTRNS)
IP := LOCATION(RTRNS)
OUTPUT := OUTPUT ;

Computation Halts
(To make truly re-entrant initialization must again be performed)

SUBROUTINES : The following four subroutines were employed in defining the FPM

GET.LNK(STATIC.LINK, TOP)

IF STATIC.LINK is 0 , give 0 as the value of GET.LNK .
Otherwise starting at AR<TOP - 1> GET.LNK progresses down the dynamic chain of the AR-stack (i.e., keeps decrementing by one the level of the AR-entry to be examined) until the variable declared in the CONTROL location STATIC.LINK is found, the dynamic level (or AR-stack level) of this entry is returned as the value of GET.LNK .

GET.ARG(POINT)

G.ARG := GET.CHECK([POINT + 1], ENVIR, R)

Case I : No returns made
Give as the value of GET.ARG

[POINT + 1] finish ENVIR |

where finish is

\[
\begin{cases} 
\text{MATCH([POINT + 1]) if it exists} \\
[POINT + 1] \text{ otherwise} 
\end{cases}
\]
Case II: Return made

Then G.ARG has the form

| LOCATION ENVIRONMENT |

so give as the value of GET.ARG

| LOCATION(G.ARG) finish ENVIRONMENT(G.ARG) |

where finish is

\[
\begin{cases}
\text{MATCH(LOCATION(G.ARG)) if it exists} \\
\text{LOCATION(G.ARG) otherwise}
\end{cases}
\]

LOOK.UP(VARIABLE,ENVNR)

Check for VARIABLE in the activation record AR<ENVNR> and all those AR-entries statically chained to it. If VARIABLE does not have a current AR-entry, then VARIABLE is returned as the value of LOOK.UP (i.e., VARIABLE is a free variable in the current environment). Otherwise an AR-entry corresponding to VARIABLE is found.

Case I: VARIABLE is a 'constant'

Give \( b_j \) for the appropriate \( j \) as the value of LOOK.UP.

Case II: VARIABLE corresponds to an AR-entry of the form

| VARIABLE BEGIN END ENVIRONMENT STATIC.LINK |

Give as the value of LOOK.UP.

| BEGIN END ENVIRONMENT STATIC.LINK |
RET.CHECK(POINT,ENVR,1)

The 1-stack has entries of the form
| BEGIN       END       EV |

Compare BEGIN of the top 1-entry with POINT.

Case I: BEGIN ≠ POINT

Give
| POINT       ENVR |

as the value of RET.CHECK and no returns are made.

Case II: BEGIN = POINT

Pop the top 1-entry and give
RET.CHECK(END,EV)

as the value of RET.CHECK and a return is made.

Note that the stack 1 is a parameter of RET.CHECK

We give now an example FPM computation on the same pre-
processed input \(((\lambda n_1(n_1 n_1) \cdot \lambda n_2 \lambda n_3(n_2 n_3) \cdots) h)\); which when loaded into the CONTROL area and pre-execution
information is gathered has the form given earlier (on page 26).

The activation record stack AR for the computation is:

<table>
<thead>
<tr>
<th>ENTRY</th>
<th>VARIABLE</th>
<th>BEGIN</th>
<th>END</th>
<th>ENVIRONMENT</th>
<th>STATIC.LINK</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>n_3</td>
<td>b_1</td>
<td>'constant'</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>n_2</td>
<td>13</td>
<td>13</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>n_3</td>
<td>18</td>
<td>18</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>n_2</td>
<td>6</td>
<td>6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>n_1</td>
<td>9</td>
<td>16</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>STEP</td>
<td>IP</td>
<td>P</td>
<td>ENVR</td>
<td>R</td>
<td>AR</td>
</tr>
<tr>
<td>------</td>
<td>----</td>
<td>---</td>
<td>------</td>
<td>-----</td>
<td>----</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>1</td>
<td>1</td>
<td>17</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>0</td>
<td>4</td>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>0</td>
<td>.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>12</td>
<td>1</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>13</td>
<td>3</td>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>18</td>
<td>1</td>
<td>1</td>
<td>19</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13</td>
<td>5</td>
</tr>
<tr>
<td>16</td>
<td>19</td>
<td>0</td>
<td>1</td>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>13</td>
<td>0</td>
<td>5</td>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>20</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>14</td>
<td>0</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>15</td>
<td>0</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>16</td>
<td>0</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STEP</td>
<td>IP</td>
<td>P</td>
<td>ENV</td>
<td>R</td>
<td>AR</td>
</tr>
<tr>
<td>------</td>
<td>----</td>
<td>---</td>
<td>-----</td>
<td>---</td>
<td>----</td>
</tr>
<tr>
<td>21</td>
<td>20</td>
<td>0</td>
<td>1</td>
<td>**</td>
<td>$\lambda b_1(h\ b_1)$</td>
</tr>
<tr>
<td>22</td>
<td>HALT</td>
<td></td>
<td></td>
<td></td>
<td>$\lambda b_1(h\ b_1)$</td>
</tr>
</tbody>
</table>

So the FPM implements a procedure definition by creating an AR stack entry and a procedure call by transferring control to the beginning of the procedure, adjusting the current environment pointer and stacking a return link on R. Formal-actual parameter correspondences also give rise to creating AR entries and each parameter call is implemented as a call by name. The AR entries are linked by static chains, the pointer ENV designates which chained AR entries are in the current environment and IP is used to fetch the next instruction while the CONTROL itself remains unaltered throughout the computation. As a model then for implementing procedure evaluation in an ALGOL-like language the FPM is considerably more realistic than the BLCM which formed the basis from which the FPM was constructed.

As with the BLCM a symbol string called the fixed normal form and denoted $F(i)$ can be associated with the information structures generated at the i-th step of a FPM computation. $F(i)$ is formed by concatenating the OUTPUT with P opening parentheses and then with a trace from the CONTROL location indicated by the current IP to the terminating semicolon using variable look-up and return transfers to determine the successive
portions of the CONTROL program to be traced and the environment in which look-ups are to be made.

DEFINITION: For any computation by the FPM on a pre-processed \( \lambda \)-expression loaded into the CONTROL area with pre-execution information gathered, the Fixed Normal Form at Step 1, denoted \( P(1) \), is given by

\[
P(1) := \text{NORMAL.FORM}(\text{AR}, \text{R}, \text{ENVR}, \text{IP}, \text{OUTPUT}, P) =
\]

\[
\text{OUTPUT} := (\ldots(\ldots(\text{tracef}(\text{IP}, \text{ENVR})
\]

\[
p\text{-times}
\]

where \( \text{tracef}(\text{IP}, \text{ENVR}) \) is evaluated by first performing the assignments:

\[
\text{RTS} := \text{R} \\
\text{IPT} := \text{IP} \\
\text{EVMT} := \text{ENVR}
\]

$ Done to save the values of \( \text{R}, \text{IP}, \text{ENVR} \) from being changed while evaluating \( \text{tracef} \) $

Now consider the possible contents of CONTROL location IPT:

\[
(\text{MATCH IPT} := \text{IPT} + 1
\]

\[
- \text{give } (\ldots(\text{tracef}(\text{IPT}, \text{EVMT})) \text{ as value}
\]

\[
\lambda \text{ MATCH STATIC.LINK IPT} := \text{IPT} + 1
\]

\[
\text{give } \lambda \text{ as value}
\]

\[
\text{give } \ldots \text{ as value}
\]
\( \sigma \) where \( \sigma \) is \( \) or \( 

\text{RC} := \text{RET.CHECK}(\text{IPT}, \text{EVMT}, \text{RTS})

\text{Case I: No return made}
\text{IPT} := \text{IPT} + 1
\text{give } \sigma := \text{tracef}(\text{IPT}, \text{EVMT}) \text{ as value}

\text{Case II: Return made}
Then \text{RC} has the form
| LOCATION \text{ ENVIRONMENT} |
\text{IPT} := \text{LOCATION}(\text{RC})
\text{EVMT} := \text{ENVIRONMENT}(\text{RC})
\text{give } \text{tracef}(\text{IPT}, \text{EVMT}) \text{ as value}

\( x \)
\text{VAR} := \text{LOOK.UP}(x, \text{EVMT})

\text{Case I: VAR is } x \text{ or } b_j \text{ for some } j
\text{IPT} := \text{IPT} + 1
\text{give } \text{VAR} := \text{tracef}(\text{IPT}, \text{EVMT}) \text{ as value}

\text{Case II: VAR has the form}
| BEGIN \text{ END} \text{ ENVIRONMENT} \text{ STATIC.LINK} |
On \text{RTS} stack
| [\text{END(VAR)} + 1] \ [\text{IPT + 1}] \ [\text{EVMT}] |
\text{EVMT} := \text{ENVIRONMENT}(\text{VAR})
\text{IPT} := \text{BEGIN}(\text{VAR})
\text{give } \text{tracef}(\text{IPT}, \text{EVMT}) \text{ as value}.\)
For example, consider STEP 17 in the above FPM computation.

\[ P(17) := \lambda b_1 (h \cdot \text{tracef}(13,5)) \]

and

\[ \text{tracef}(13,5) = b_1 \cdot \text{tracef}(14,5) \]

\[ = b_1 \cdot \text{tracef}(15,5) \]

\[ = b_1 \cdot \text{tracef}(16,5) \]

\[ = b_1 \cdot \text{tracef}(20,1) \]

\[ = b_1 \cdot ; \]

Therefore, \( P(17) := \lambda b_1 (h b_1) \cdot ; \) which we note is a well-formed \( \lambda \)-expression.

As with \( B(i) \), if the FPM computation terminates in \( k \) steps, then the Fixed Normal Form \( P(i) \) is extended to values of \( i > k \) by the definition

\[ P(i) := P(k) \quad \text{if} \quad i > k \]

In view of the design philosophy behind the FPM the next results are not too surprising.

THEOREM 3 : For any computation by the FPM on a pre-processed \( \lambda \)-expression loaded into the CONTROL area with pre-execution information gathered the following hold for all \( i \)

(I) \( P(i) \) is a \( \lambda \)-expression

(II) If at Computational Step \( i \) the contents of the CONTROL location designated by the instruction pointer IP have
the form \( \lambda n \) MATCH STATIC.LINK and \( F(i) > 0 \)
then \( F(i) \neq F(i+1) \)
otherwise \( F(i) = F(i+1) \).

PROOF: (Not given; a proof would be analogous to that of
THEOREM 1 and would proceed by induction on the Com-
tentional Steps considering by cases the possible forms
which the contents of CONTROL location IP might have
at Step 1.)

COROLLARY 5: For all \( i \) the following hold:
(I) \( F(i) \) not in reduced form \( \leftrightarrow \) \( (\exists m) \) such that \( F(i) \neq F(i+m) \)
(II) \( F(i) \) in reduced form \( \leftrightarrow \) \( \forall m \geq 1 F(i) = F(m) \)
(III) \( F(i) \) in reduced form \( \rightarrow \) \( (\exists m) \) such that the correspon-
ding FPM computation terminates in
\( m \) Steps and \( F(m) \) is the OUTPUT.

PROOF: (Not given; a proof would be analogous to the proofs
of COROLLARIES 2 & 3.)

THEOREM 4: For any pre-processed \( \lambda \)-expression inputed to
both the BLCH and the FPM the following hold:
(I) \( (\forall i)(\exists j)[ B(i) = F(j) ] \)
(II) \( (\forall j)(\exists i)[ F(j) = B(i) ] \)

PROOF: (Induction on normal forms)
(I) [By induction on \( i \)]
\( i = 0 \) \( B(0) \) and \( F(0) \) are by definition the in-
puted \( \lambda \)-expression hence \( B(0) = F(0) \).

Inductive Assumption: Assume for all \( i < k \) that (I) holds.
Then there exists a \( j_k \) such that \( B(k) \equiv F(j_k) \).

**Case 1**: \( B(k) \) is in reduced form

By **COROLLARY 2** \( \forall m \geq k \ B(k) \equiv B(m) \).

In particular \( B(k) \equiv B(k+1) \equiv F(j_k) \).

**Case 2**: \( B(k) \) not in reduced form

(a) If \( B(k) \equiv B(k+1) \), then \( F(j_k) \equiv B(k+1) \).

(b) If \( B(k) \nmid B(k+1) \), then by **COROLLARY 5** (\( \exists m \)) such that \( F(j_k) \nmid F(j_{k+m}) \) and \( F(j_{k+m}) \equiv B(k+1) \) since the leftmost part of the form \( (\lambda n \ C \cdot D) \) in a non-reduced wff is uniquely determined.

(II) is established by a straightforward argument modeled on that of (I) above.

**Q.E.D.**

**COROLLARY 6**: The FPM correctly implements the \( \lambda \)-K-calculus.

**PROOF**: Immediate from **THEOREM 4** and **COROLLARIES 4 & 5**.

**Q.E.D.**

Two mechanical evaluation strategies for implementing the \( \lambda \)-calculus \( S_1 \) and \( S_2 \) will be said to be equivalent if for every inputed wff \( x \):

(I) \( S_1 \) applied to \( x \) halts \( \iff \) \( S_2 \) applied to \( x \) halts

and (II) if the computations on \( x \) halt, then the \( \lambda \)-expressions outputed by \( S_1 \) and \( S_2 \) are equivalent up to renaming.

Informally equivalence of strategies expresses the idea of "same output for the same input." In general the equivalence of mechanical evaluation strategies is recursively unsolvable.
COROLLARY 7: The BLCM and the FPM are equivalent.

PROOF: Immediate from THEOREM 4 and COROLLARIES 3 & 5.

Q.E.D.

The pronounced differences between these two implementation strategies add to the interest of the above results.
Both the BLCM and the FPM essentially implement a call by name semantic specification of the \(\lambda\)-calculus as a programming language which requires in reducing a subexpression of the form \((\lambda x M \cdot A)\) that the wff \(A\) be substituted unevaluated for each occurrence of \(x\) in \(M\). Now if \(A\) were evaluated just prior to substitution for instances of \(x\), this would be the \(\lambda\)-calculus analogue of call by value. Landin [LM] takes this approach in providing an evaluating mechanism for the \(\lambda\)-calculus which is by now well known.

For purposes of comparison a suitably modified version of Landin’s evaluation strategy, called herein the SECDM, could be made to operate on pre-processed \(\lambda\)-expressions (which include terminating \(\_\:\_\:\_\_\)’s). The author claims that as with the BLCM and the FPM a normal form for the \(i\)-th state of a SECDM computation could be defined and then used in an inductive proof that the SECDM correctly implements a call by value semantic specification of the \(\lambda\)-calculus.

As noted in [W] call by value evaluation for the \(\lambda\:K\) calculus is slightly less general than call by name since there exists \(\lambda\)-expressions whose reduction sequences terminate in the latter but not the former approach (e.g., \((\lambda z\lambda yz(\lambda x(xx)))(\lambda x(xx))\)) where
an irreducible wff is to be substituted for no instances of s in 1yy . A notion of restricted equivalence of evaluating mechanisms can be introduced by requiring output equivalence only for those inputed programs whose computations terminate for both mechanisms. It would then follow that the SECDEM is restrictedly equivalent to the BLCM and hence to the FPM as a consequence of the Church-Rosser Theorem.

The restricted equivalence of the FPM and the SECDEM has especial interest in the light of the elegant result by Lucas [L] on the equivalence of a "static chain" with a "dump" implementation for the PL/I block structure. His technique of fusing two strategies into one and proving redundant information then arises is dependent upon both strategies processing the input identically (while creating different information structures) which is not true of the FPM and the SECDEM (or the BLCM ). In fact equivalence was deduced from the properties of the normal forms and the Church-Rosser Theorem (i.e., a structured and manageable representation of the run-time total states and a result on the equivalence of different computational sequences).

The λ-calculus is due to Church [C] and Landin [LM] first presented a mechanical evaluator for it. Wegner in section 3.8 of [W] introduced the comparative study of evaluation strategies for the λ-calculus as a means for gaining "insight into actual function-evaluation procedures in 'real' programming languages." This work is a further development of that study and part of the still inchoate theory of programming languages. Other results
within the theory on correctness and equivalence of implementations are those of McCarthy and Painter [MP] and Lucas [L].

And in addition to the principal contributions of Landin [LM,LC] the \( \lambda \)-calculus has also been used for investigating the formal definition and semantics of programming languages by Böhm [BC], Burge [B], Burstall [BS], Morris [ML] and Strachey [S].

The normal forms for the BLCM and the FPM are related to the instantaneous descriptions of a Turing machine computation and the general snapshots proposed by Naur [N] in that they express the total state of a computation at different stages but have the important additional feature of being \( \lambda \)-expressions. In using the known correctness of the BLCM to pass to a pure-procedure statically chained stack implementation of the \( \lambda \)-calculus we follow the approach of Dijkstra [D] in constructing a mechanism, the FPM, to correctly meet some pre-specified desired dynamic behavior. Induction on computational steps has been used previously by Floyd [F] and induction applied to normal forms is somewhat similar to the structural induction of Burstall [BP] which apparently is subsumed by the more general recursion induction of McCarthy [N].

Realizations of the three \( \lambda \)-calculus implementing mechanisms discussed have been programmed by the author in SNOBOL4 [G] in which intermediate print-out gives the respective values of the total state variables and the appropriate normal form at each computational step.
REFERENCES


