Proceedings 2nd Annual Symposium on Computer Architecture, Houston Texas Jan 20-22, 1975

REDUCTION LANGUAGES FOR REDUCTION MACHINES

K.J. Berkling

Institut fuer Informationssystemforschung
Gesellschaft fuer Mathematik und Datenverarbeitung

St.Augustin, Schloss Birlinghoven, Germany

Summary

This paper describes a particular realization of a Lambda-Red language as a machine language. Parts of it resemble the Lambda Calculus or a transposed form of it. A constructor syntax is employed such that a linearized preorder representation of the syntax tree is the information structure on which the machine operates. Instances of reduction rules are recognized by combinations of constructors and atoms. Reduction rules with 1, 2 and 3 constructors and/or atoms have been described.

A recursive control structure forms the essential part of the implementation. There is a one to one correspondence between constructor syntax and control structure rather than a simulation of recursive structure by von Neumann type instruction sequencing. Thus the system is easily expandable by new constructors and atoms. Execution, that is the application of reduction rules, is subsumed under editing. Emphasis has been on the following pragmatic concepts: locality of action, directness, and security. There are no error messages, errors appear as irreducible expressions. There is no "RUN" instruction, instead the number of reductions to be performed is specified. Whatever happens - it is restricted to subexpressions. The user has the security that all his actions are limited and predictable.

The machine language resembles a higher level programming language, but it is "variable-free": expressions are named, not boxes which contain expressions. There seems to be a concept of late binding. However, this is not made a matter of principle, but a matter of degree. Between a general statement of a problem and the result are many intermediate representations - all in the same language - corresponding to various degrees of binding parameters. The user has complete freedom in the choice of variable names. There is a specially developped protection system which avoids confusion of variables.

The efficiency of the system is limited by the rate characters can be processed which is essentially determined by the control store cycletime. Measurements will be performed as soon as a simulation of the system is available. Three other important subjects, namely arithmetic, the definition of constants, and source sink input-output will be reported later.

1 Introduction

Programming has become the outstanding problem in computing. While technology has provided ever cheaper and faster hardware, the methodology to construct software is just emerging. Current hardware and software may be major obstacles towards a solution of the programming problem. However, there are several programming languages which are attractive because of their simplicity and expressive power: LISP[1], TRAC[2], and GPM[3]. Recently John Backus has added to this list his RED-languages. "RED" stands for reduction and denotes a special type of execution and evaluation.

The above mentioned programming languages have a common denominator which seems to be the reason for their attractiveness, namely, they are "substitutive". The concepts on which conventional hardware - and programming languages, too - are built contradict the concept substitution. Thus, implementations of substitutive systems on conventional hardware are bound to be inefficient.

It is the purpose of this paper to show

1) that it is possible to design hardware which is suitable for substitution systems,
2) that substitution is an important concept the implementation of which deserves further investigation, and
3) that the issue of higher level language machine architecture arises from the apparent inability of conventional hardware to support substitution economically.

2 Problem Identification

The following is an opiniated discussion of fundamental problem areas which are believed to be the cause of the difficulties in programming.

1) Long Range Effects

Writing a program generates a net of causes and effects which is not easily derivable from the visible representation of that program making it difficult to comprehend it. The effect caused by an instruction is long range: any part of the system may be involved.

2) Indirect Access

Operands and operators are seldom directly connected. Operands are rather referenced, or called by names and addresses. This separation permits the abstraction of programs from particular data. The construction of an algorithm therefore necessitates the construction of data access paths, algorithms to manipulate them, and algorithms to manage storage space. These latter tasks often overwhelm the construction of the algorithm itself.

3) Central Control

The control structure of a machine is that part which implements the next-state function of the system. This concept is based on the notion that a "present" state is defined at all. Thus, large systems with a central control resemble more a simulation than a "life" system. The many parts of such a system are merely enacted by a single agent which is mainly occupied with finding out what the "present" state of the system is.

4) Addressibility

As a consequence of points 1), 2) and 3) there is a need to generate, compute, and store addresses. Technology will provide larger and faster memories. Because of the limited address length of conventional hardware it is a problem to provide addressibility.

5) Machine Indenpendent Languages.

This notion is a misconception. Experience has shown that such a machine independence cannot be achieved mainly because of details and exceptions. Is machine independence a reasonable objective at all? A program consists of "instructions" directed to something real. To hide the real machine from the user may cause insecurity about what really happens.

3 Conceptual Approach to Problem Solution

We first formulate our attempts of solving the problems in rather general terms. Experience shows that theories employing short-range interactions have more success in explaining reality and are easier understood than long-range theories. The following ideas about a computing system seem therefore promising with respect to the problems listed in section 2:

Control is distributed to many parts cooperating over well defined local interfaces. These exist only to neighbour parts. The restrictions and boundary conditions which result from a realization of the system are an essential and determining factor. There are no unexplicable and opaque restrictions.

Programs and data are cohesive structures. They are modified and changed only locally. Substitution is always literal and not simulated. The system is interactive: program and data may be inspected and may be changed and modified at the discretion of the user. He will not be able to initialize changes he cannot predict or cannot oversee. There will be no addresses or variables.

4 Reduction Languages

J. Backus[4,5] has described a class of languages which seem to be prime candidates for our considerations. We disregard for the moment any particular realization of a reduction language. A reduction language L = (E,C,M) consists of a set E of expressions, a set C of expressions, and a partial function M from E onto C such that C is the set of fixed points of M. A simple example of a reduction language is the language of arithmetic expression without variables:

$$M((3+4)*7) = 49$$

Considering relizations we first have to explain M in terms of smaller steps.

A complete realization CR = (E,C,T), is a reduction language, where T is a total function from E into E, and C is the set of fixed points of T, the transition function. If Me = c then there is an integer n such that $T^n e = c$. With respect to our example we may write

$$T((3+4)*7) = (7*7)$$

$$T(7*7) = 49$$

Reduction languages constitute a shift of emphasis: the meaning is no longer in the states of an automaton taken on while reacting on input signales, the meaning is in the expressions and only there. But, if T is enacted by an automaton, this automaton will go through a sequence of states starting at and returning to an initial state. During that period e is not defined.

This should be contrasted with conventional instruction execution: to find out the meaning of one instruction execution the state of the whole system has to be inspected.

John Backus[5] lists six interrelated informal axiom which seem necessary and sufficient to characterize a reduction language. Property (5) needs special attention:

(5) The extended Church-Rosser property.

(a) Every terminating sequence of reductions of an expression yields the same meaning for it, and
(b) if an expression has a meaning, then every sequence of reductions on it terminates. (Property (b) is restricted to selected sequences in the usual form of the Church-Rosser Theorem.)

If anything equivalent to recursion, which cannot be relinquished, is implemented, there will always exist non-terminating reduction sequences, which a blind deterministic enactor of the transition function cannot detect. Without an enactor no expression does anything. So, we only need a human observer who always can force termination by switching off the enactor or setting a time limit. The advantage of a reduction system becomes particularly evident in this context. The result after each reduction-step is an expression e, which "means" something to the observer. He can usually decide if further reductions will terminate or not.

We have to distinguish further between "reduction" and "evaluation". On first sight these concept seem to denote the same thing. However, evaluation has the connotation that each and everything has a value. This is another pragmatic point supporting the case for a reduction system: it returns a "constant" expression c rather than an error message if it cannot apply reduction rules, and does not try to decide if this c is the result the user expected or the consequence of an error committed by the user. The expression c will be self-explanatory and exhibit why further reductions are impossible.

5 Syntax of a Reduction Machine Language

We talk about a "machine language" although there is nothing like a conventional machine language in a reduction machine. Instead, there are character strings which are transformed if certain characters or character combinations occur in these strings.

In particular, we employ a constructor syntax CS = (A,K) to define the set of expressions E. CS is a constructor syntax if

CS1) There is a subset A⊂E the elements of which are called atoms.

CS2) There is a set K of constructors k which are functions from a subset Sk of E^n into E ($n \geq 0$).

CS3) For every expression e∈E, either e∈A or there is a unique k∈K and unique e1,...,en∈E such that k[e1,...,en] = e.

These axioms regulate the construction of elements of the set E.

To be more specific, we only allow two-place (and one-place) constructors. So far a particular encoding or representation for constructor functions has not yet been given. We choose a direct, explicit one. Let k e1 e2 = e with juxta position as the only syntactic tool. The set of expressions becomes therefore the set of linear representations of binary trees in preorder form[6].

It is our intention to implement a Lambda-Red language, that is, a language which resembles the lambda calculus[7]. Lambda-Red languages are closed applicative languages. A reduction language L = (F,C,M) with a constructor syntax S = (A,K) is applicative iff:

AL1) A⊂C

AL2) There is a two-place constructor ap∈K such that for all e,f∈E:

M ap e f = M ap Me Mf

AL3) For all other constructors:

M k e f = k Me Mf

Using these three axioms we may reduce an expression e to a constant where the constructor ap occurs only with constant expressions: ap c d. The applicative language is closed iff there is a function

R∈ [C →[C → E]]

such that:

CAL1) R is total over C

CAL2) For every c∈C, Rc∈[C →E] is total over C.

CAL3) For all c,d∈C:

M ap c d = M ap Rc d

The function R is called representation function. A major part of a hardware implementation would consist of an implementation of this representation function R.

The function R will be undefined for many elements of C, numbers for example. The machine will take care of these cases in a manner consistent with the reduction concept. This means the machine will not stop with an incomplete reduction of the ap, on the contrary the machine will continue with other reductions after restoring the ap with its components. This has pragmatic significance as a direct in-place error indication.

Generally, only atomic elements of C will be associated to meaningful mappings C → E. In Lambda-Red languages, however, we have lambda expressions which are "applied" to arguments by substituting the argument at designated places of the lambda expressions.

5.1 Substitution

Lambda-Red languages resemble the lambda calculus and inherit all its problems caused by the introduction of variables.

We introduce a new class V of atoms called variables, and a constructor lambda represented by the character \. Variables are represented by strings of letters externally and their encodings internally. Variables are constants, thus if v ∈ V: Mv = v. This is in sharp contrast to conventional programming languages where variables are internally represented by a more or less complicated network of references to memory cells.

A lambda expression \ v e may be a component of an application

ap \ v e f

The reduction of this expression corresponds to the beta-conversion rule of the lambda calculus: f is substituted into e for free occurrences of v.

We have found by experimenting that it is convenient to have several ap-type constructors. Firstly, every constructor has a transposed form such that

ap' f \ v e

is an instance of beta-conversion and

M ap' f \ v e = M ap \ v e f

The use of transposed constructors may improve the readability of expressions.

The other differences of the ap-type constructors may be explained with reference to axiom AL2:

M ap e f = M ap Me Mf

This form of the axiom applies if e is not a lambda expression, since e has to be reduced to a constant before the representation function R can be employed. If, however, e is a lambda expression, various sequences of reductions are possible since substitution does not require constant components. The variety of ap-constructors arises if not both of their components are reduced to constants before substitution. Let g be a lambda expression in:

AL2L)
M ← g f = M R2 R1g f
M . g f = M R2 R1g Mf
M ▵ g f = M R2 R1 Mg f.

The function R1 prepares the lambda expression for substitution while the two-parameter function R2 performs it.

It is not necessary to introduce a fourth constructor for the case where the first component is not a lambda expression, since the machine is designed to detect this. So we have if e does not start with a \ constructor

AL2E)
M ← e f = M ← Me f
M . e f = M . Me Mf
M ▵ e f = M ▵ Me f

The constructor . corresponds to the usual ap while the constructors ← and ▵ yield still other cases. The same statements hold for the group of transposed ap-constructors, which are →, :, and ▽, respectively.

The contextual dependencies of constructors are rich enough to model all typical higher level language constructs. Moreover, the combinational properties of the constructors are paralleled by combinational properties of the machine components related to these constructors. This yields a very economic design for a rather rich structure.

So far we did not mention the problems with free and bound variables in substituton procedures. Our solution[8], which differs from all known ones, is not only effective but also efficient, because a substitution may be executed without any preparations. This method protects possible free occurrences of variables by a constructor lambda-bar from being bound erroneously while substituting. The lambda-bar constructor is represented by the character /. The user will find /v where he might have expected v alone. A lambda-bar expression can only appear as a component of a lambda expression with the same variable. The lambda-bar will automatically disappear if this lambda expression is applied to some argument.

We may now return to chapter 5.

There is also a constructor • corresponding to the DOT-operator of LISP[1] which serves to compose data structures.

A few other constructors are implemented for special purposes. The constructors # and ~ play the role of escape characters to denote numbers, the constructor ♦ is a special binding constructor and is dealt with later.

Expressions are linear representations of binary trees in preorder form. Constructors correspond to nodes, atoms to leaves, and subtrees to subexpressions. Constructors may be in a predecessor - left successor or predecessor - right successor relationship with respect to the tree structure. All

combinations are allowed with the following two exceptions:
1) A binding constructor (\↑) may only have variables as left-successor.
2) A number constructor may only have number parts as successors.

Primitive Atoms. The system is equipped with a set of primitive, one character atoms (leaves). Single letters serve as variables and may appear as leaves everywhere. The digits 0 to 9 are treated as integers and comprise the only numbers without the number constructors. If single digits appear in operator position, namely as that component of an application constructor which is subjected to the representation function, they are interpreted as constant functions which yield themselves if applied to arbitrary arguments.

Truthvalues are represented by single characters. They act in operator positions of the constructors (.,:,←, →) as constant functions, in operator position of the constructors (∧,∨), however as selectors in the conditional device.

We have the NIL atom represented by the right bracket] which acts as the identity function in operator position.

Members of the set of number-related operators and logical connectives are treated as constants if they appear in operand position. In operator position, however, the representation function is invoked according to their usual meaning.

There are three decomposition operators (○ ⊃ ⊂) which delete the constructor, the left subtree, or the right subtree, respectively.

Compound Atoms. Variables are formed by a sequence of letters terminated by any non-letter. (The set of non-letters contains the blank). Compound numbers start with the constructor #. Arithmetic will be handled as in the programming language TRAC².

Expressions. We are now in a position to form arbitrary binary tree structures from constructors as nodes and primitive and compound atoms as leaves. The machine accepts the preorder linear representation of such tree structures. The machine will reject any character string which is not a binary tree or can be augmented by the machine to such binary tree using NIL atoms.

6 Use of the Reduction Machine

Before we outline a possible implementation of the proposed reduction machine we will exhibit the representation of familiar higher level language constructs. We believe the proposed realization represents a rather close fit to the common usage of programming languages, such that the user does not really have to learn a new language. Although the rules are more stringent, we believe that a programm written in the proposed realization of a reduction language is more transparent and comprehendable, too.

Infix Expressions. The input of the machine accepts fully - parenthesized infix expressions. The printout appears also in this form. The internal representation, however, is a binary tree with two constructors corresponding to the left and right parenthesis and the two terms and the operator as leaves.

external	(e + f)
internal	( ) e + f
	. : e + f

The parenthesis ( ) and constructors . : , respectively, are made synonyms for the reduction processor. The printout processor, however, uses them to restore the external representation. The examplifying plus sign may be replaced by any expression.

The Polish prefix notation for arithmetic expression - which should not be confused with the preorder linear representation for program structures - has the following representation:

. . + e f

We are already in a position to implement the logical connectives without resorting to actual hardware. The concepts of constant and identity function are sufficient if we let the primitive constants (∧ ∨ ⊤⊥) play their various roles in operator and operand positions. In Figure 6.1 we have exhibited the evaluation of (A∧B) employing four reduction rules. Corresponding reduction rules for the OR function are M: ⊥ ∨ =] and M: ⊤ ∨ = ⊤. The transposed forms of all these reduction rules are provided, too.

Conditional. The constructor ∆ indicates a complete reduction of the operator component before the reduction process continues. This property allows us to construct a conditional

e	T^2e
∆ ∆ f g	f
∆ ∆ f g	g

In general a predicate expression, that is an expression which reduces to a truth value, will be constructed as operator of the constructors (∆ ∇). The truth value acts then as selector. Finally, the reduction of f or g takes place. The conditional can be used correspondingly to model either the conditional expression or the conditional statement of conventional programming languages.

Lists. The constructor • corresponds to the DOT operator, . ⊂ and . ⊃ correspond to CAR (head) and CDR (tail), respectively. To facilitate listprocessing, the characters •, [, and , are made synonyms for the reduction processor such that a tree structure like

• e • f • g • h]

may be typed in and printed out as

[e, f, g, h]

The reason for encoding the NIL-atom by a closing bracket is now obvious. The list elements may be any permissible expressions including lists represented in this way.

Call by value. The constructors . and : correspond to a call by value since the operand is first reduced, then inserted, and finally the function is reduced.

Call by name. Similarly, the constructors ← and → perform call by name, since the operand is first inserted, and then the function is reduced.

It should be noted, however, that because of the lack of any side-effects whatsoever the result will be in either case the same, only the number of reductions might differ.

Functions. Functions will be constant expressions in general. Several parameters are simply listed with a lambda constructor each.

\v1\v2 -- \vn f v1,...,vn ∈ V

Arguments follow in the same sequence as the parameter variables, the corresponding ap's, however, in opposite direction:

apn -- ap2 ap1 \v1\v2 -- \vn f e1 -- en

The constructors ap1,...,apn may be selected from the set (. ← ∆). Constructor ap1 and its components are the first to be reduced, because it is the only one which has a lambda expression as operator component.

After all ap's have been reduced f has been changed such that all parameter variables are replaced by actual parameters. The reduction

of this expression results in a constant expression, that is, the value returned by the function.

Infix expressions are a special case of the application of two parameter functions. Constructors may be mixed with their transposed versions such that the same f

(e1 f e2) = ..f e1 e2

may be used in either context.

Since we do not have box-like variables in the reduction machine, it is not possible to fill a box the name of which has been transmitted to an expression by reducing this expression. However, the ap constructors offer a large variety in sequencing reductions. One sequencing method resembles procedural language.

PROCEDURAL Reduction. Assignment statements of conventional programming languages fill a box. The effect of this action extends up to the next filling. A reduction machine resembles this process by reducing one expression and substituting the result in an other expresseion. The constructor : allows as to write this down as an expression g

```
g =  : e1 \ v1
     : e2 \ v2
     : e3 \ v3
       ---
     : en \ vn
     f
```

This clearly resembles a sequence of assignment statements. The expression f, however, is not a GOTO !

The sequence of reductions is top-down because of the constructor :. The reductions affect only expression g. The results of the computations in g are either available as a modified f which may be a subexpression of another expression. Of, the expression f may be a function call. Using the constructor . a function call has a pattern like

. . . v e4 e5 e6

The reduction of g would then change f to a constant expression with results deposited in the arguments of the function call f.

The transposed form of this call

f = : e6 : e5 : e4 v

yields a pragmatically better pattern for g: there is a computation part, a memory part, and a connector part.

Further reductions become possible if a function gets substituted for v

: e6 : e5 : e4 \ v4 \ v5 \ v6 e

The memory part keeps e4, e5, e6 queued up for transmission to e.

To construct something which resembles a block or procedure we bind the connector v which gets the appropriate variable name END. The block can be inserted in a sequence of "assignment" reductions:

```
---
: e7  \ v7                "assignment"
                          reduction
^  \ END                  equivalent to begin
     : e1  \ v1  }
       ---       }  computation part
     : en  \ vn  }
     : e6        }
     : e5        }  memory part
     : e4        }
     END                  connector
\  v4           }
\  v5           }  argument for block
\  v6           }
:  e8  \ v8     }
---
```

The sequencing properties of the constructor ^ effects the complete reduction of the computation and memory part before the block is applied to the argument (surrounding block) following the END variable. Note that the object in the reduction language which resembles a block must have an explicit interface for its output.

Recursion and Iteration. These constructs can be effected by using the binding constructor ↑, which is similar to the Lambda constructor \. There is a reduction rule associated with this constructor

↑ v e = - \ v e ↑ v e

which reproduces an expression within itself. This is sufficient to construct recursion and iteration.

7 Implementation by Hardware

A reduction language machine may be realized in various different ways. In our first attempt we will choose the conventional separation of an active component operating on information stored in a memory component. In a next step one would try to combine these two components in one. The reduction concept seems particularly suited for a "processing in memory". But, for the time being, appropriate hardware concepts are not available.

The expressions in the foregoing sections are linear representations of binary trees. We store expressions as character strings in memory components which operate as stacks, that is, they act on push and pop signals and do not need addresses as input. An attempt[10] to directly represent trees was discarded.

We need an active component which scans expressions to find instances of reduction rules. It is not appropriate to perform this scan like an instruction counter scans an instruction stream, because the expressions are structured in subexpressions. The scan has to work forward and backward. An economic solution consists of generating in a stack called A the transpose of an expression in a stack called E.

before	after
E k e1 e2 A	E A k' e2' e1'

The transpose of an atomic expressions is usually the atom itself. Compound atoms may require special rules. The correspondence to a conventional von Neumann control structure is as follows: the constructors and atoms are the operation-codes, the expressions are the operands. There is a major difference, however. The control structure to be designed effects the transpose of an expression. This is asked for twice within this control structure. This cannot be transformed into a loop structure. The control structure is therefore fitted with a system stack called MU which serves to hold return points. The system stack has the same hardware properties as the other stacks in the system and is potentially infinite, too. We cannot tolerate a bound on the depth of expressions.

We will give a description of the control structure in the form of flowdiagrams.

Figure 7.1 shows the flow diagram for the transpose algorithm FA. Only one branch of each type is exhibited.

The basic pattern of control flow shows a character popped by stack E and "sorted" into its proper branch. This will be abbreviated by a horizontal line and solid triangles denoting the exit down for the annotated character. Pushing something into a stack is indicated by a solid triangle pointing upward. Stack markers are usually not annotated to the corresponding triangles. The correspondence is established by geometry. The particular choice of stack markers is a matter of the next lower design level.

The following concepts have proven useful in this context: A yes-exit consumates the tested character, there is no need to store the character, conveyed information is now in

the locus of control in the diagram. Once something is pushed into a stack, the control structure loses information and control is directed to a neutral point.

There is also a transpose algorithm AE. Execution of algorithm AE after algorithm EA restores all stacks to the original state.

The basic hardware to realize these algorithms consists of a processor component P and three memory components acting as stracks E, A and MU. Figure 7.2 shows a state chart of the processor component. There is one control store which translates stack marker encodings into control words. These control words contain in this simple basic model either a transposed constructor to be pushed into stack A and a signal to pop MU, or a stack marker to be pushed into stack MU and a signal to pop E. The other control store translates expression characters. Its control words contain either atoms to be pushed into stack A and a signal to pop MU, or a stackmarker to be pushed into stack MU and a signal to pop stack E. One-character atoms get simply moved from stack E to A, compound atoms will have special start and stop symbols around them. The transpose processor component may be realized as separate component interfacing with memory components, or as special state of the reduction processor.

We have seen in the preceding sections that an instance of a reduction rule can be characterized by constructor combinations with constructors or atoms. Although an expression is undefined while being transposed, its components are displayed during that process in the stacks E, A and MU in a way which is convenient for recognizing instances of reduction rules. We have represented in Figure 7.3 an arbitrary expression in a mixed form. Some parts of the expression are given as subexpressions e1 to e7, some parts as tree structure where we have to visualize the nodes as constructors. It also represents the state of the expression while being transposed. Stack A contains top-down the transposed subexpressions e4, e3, e2, e1, stack E contains top-down the expressions e5, e6, e7. Stack MU, however, contains stack markers instead of constructors. The stack markers contain also the information of being left- or right-successor. The position shown corresponds to the point * in Figure 7.1. Being at that position encodes the constructor k, which is not explicitly represented otherwise. Thus the expression is completely represented by the contents of stacks and the position of control. By inspecting the top elements of the stacks, we may determine the left- and rightsuccessor of k and its predecessor, from which we also learn whether k is a leftsuccessor or rightsuccessor. Instances of reduction rules are singled out in this manner while transposing an expression.

Execution of a reduction rule means a rearrangement of the top ends of the stacks. For example, the top element of stack MU is a marker indicating the left-successor of some constructor k2. The expression below k2 is now

k2 k e4 e5 e6

If we replace the marker by the corresponding one indicating the right-successor of k2, the expression below k2 has been changed to

k2 e4 k e5 e6

Another simple case is the identity function. Let the atom] be deleted from stack E and the locus of control in the corresponding branch of the control structure. Now we test the top element of MU for a leftsuccessor marker of the constructor . . This stack marker is automatically deleted if the test result is yes. We can see from Figure 7.3 that the resulting tree structure now contains e5 instead of .] e5.

Generally, one or more auxiliary stacks might be necessary to execute more complicated reductions. A subset of the lambda and ap-type constructors is exhibited in Figure 7.4. The only complicated operation is the transposition R1EB, which replaces free occurrences of a designated variable in an expression by a special one-character marker. The control flow brings into stack A the reduced or unreduced argument expression. The operation R2ABE is a simple transposition from B to E, except that the source is switched to A if the special one-character marker mentioned above appears in B. Actually BE is identical to R2ABE, since the special marker appears only in this context. The recursion constructor ' uses R1EB and R2ABE with the only difference that the recursion constructor together with its two components has been copied and transposed by EA to stack A. The copying process effects the duplication necessary for recursion.

The examples demonstrate the basic design methods for a reduction machine. New reduction rules can be added easily as long as there are enough codes for distinct stack markers available.

8 Interactive Computing

The most interesting property of a reduction language in the context of interactive computing is the return to the set E of expressions after each elementary reduction step. There is nothing to learn beyond the construction of elements of set F and the reduction rules. Editing is essentially designed to walk the tree represented by the input. Any editing operation can be applied to any subtree, including the order to perform reductions. Any finite number of reductions may be specified for a subtree. This solution insures that the machine always terminates at well defined points.

Acknowledgement

I am indebted to C.A. Petri who made this work possible by creating favorable conditions. Special thanks go to Mrs. E. Pless for preparing the paper on the BITS text editor.

References

[1] J.McCarthy et al
LISP 1.5 Programming Manual, Cambridge, Mass.: MIT Press, 1964.

[2] C.N. Mooers, L.P. Deutsch
TRAC, A Text Handling Language. Proc. ACM 20th Nat'l Conf. (1965) 229

[3] C. Strachey
A General Purpose Macro Generator. The Computer Journal 8(1965)3

[4] J. Backus
"Reduction Languages and Variable-free Programming", San Jose, California, IBM Research Report RJ 1010, April 7, 1972.

[5] J. Backus
"Programming Language Semantics and Closed Applicative Languages", San Jose, California, IBM Research Report RJ 1245, July 5, 1973.

[6] D.E. Knuth
"The Art of Computer Programming, Fundamental Algorithsms", Vol. 1, Reading, Mass.: Addisom-Wesley, 1968, pp. 318-319.

[7] A. Church
"The calculi of lambda-conversion", Annals of Math. studies, No.6, Princeton Univ. Press, 1941.

[8] K.J. Berkling
"A Method to Rename Variables Using an Unbinding Operator",
(to be published)

[9] P.J. Landin
"The mechanical evaluation of expressions",
The Computer Journal, Vol. 6, No.4 (January 1964), pp. 308-320.

[10] K.J. Berkling
"A Computing Machine Based on Tree Structures"
IFEE Transactions on Computers, Vol. C-20, No.4 (April 1971), pp. 404-418.

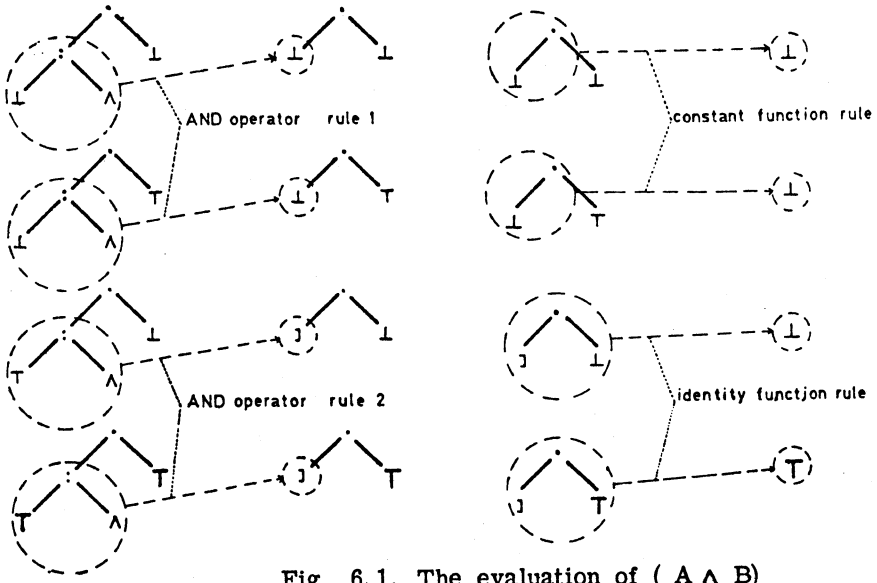

Fig. 6.1. The evaluation of (A ∧ B) = .: A ∧ B employing four reduction rules.

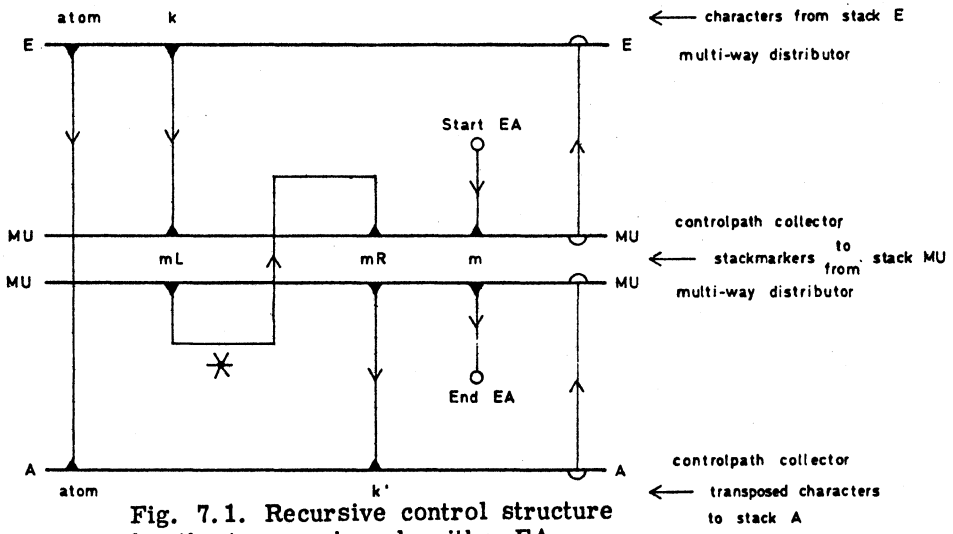

Fig. 7.1. Recursive control structure for the transposing algorithm EA using a system stack MU.

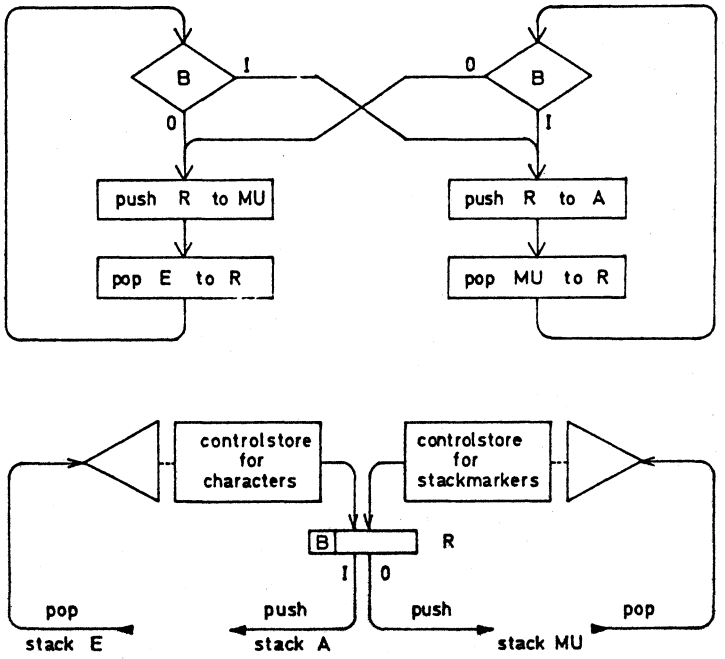

Fig. 7.2. State chart and dataflow graph for algorithm EA.

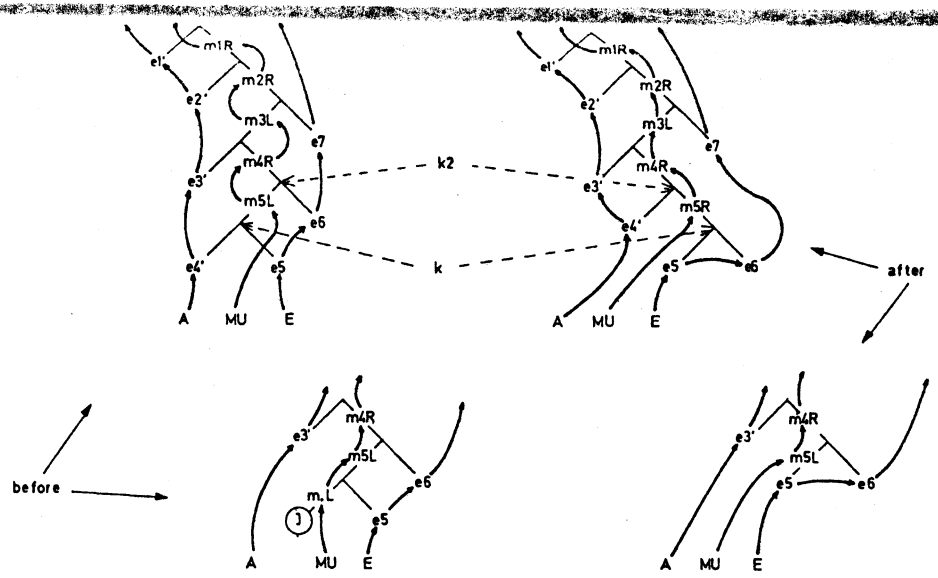

Fig. 7.3. Rearranging of tree structures by manipulating stack tops.

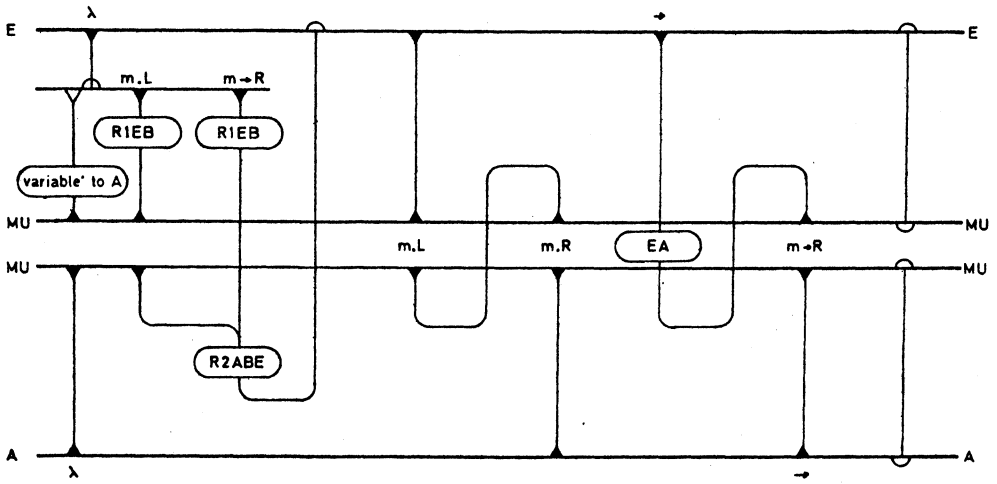

Fig. 7.4. Part of the reduction processor control structure for the constructors λ , ., and →.