2-1992

Conceptual Background for Symbolic Computation

Klaus Berkling

SU-CIS-92-02

Conceptual Background for Symbolic Computation

Klaus Berkling

February, 1992

School of Computer and Information Science
Syracuse University
Suite 4-116, Center for Science and Technology
Syracuse, NY 13244-4100

CONCEPTUAL BACKGROUND FOR SYMBOLIC COMPUTATION

Dr. Klaus Berkling
Professor of Computer and Information Science

ABSTRACT: This paper is a tutorial which examines the three major models of computation—the Turing Machine, Combinators, and Lambda Calculus—with respect to their usefulness to practical engineering of computing machines. While the classical von Neumann architecture can be deduced from the Turing Machine model, and Combinator machines have been built on an experimental basis, no serious attempts have been made to construct a Lambda Calculus machine. This paper gives a basic outline of how to incorporate a Lambda Calculus capability into a von Neumann type architecture, maintaining full backward compatibility and at the same time making optimal use of its advantages and technological maturity for the Lambda Calculus capability.

School of Computer and Information Science
Suite 4-116
Center for Science and Technology
Syracuse University
Syracuse, NY 13244-4100

1. INTRODUCTION

For the past decade, a growing number of computer scientists have advocated the use of functional programming as a means of easing the software crisis. These advocates claim that functional programming languages and techniques increase programmer productivity and enhance the clarity of programs, thereby aiding in their verification and maintenance. A major obstacle to industrial experimentation and acceptance of functional programming languages is that conventional von Neumann computer architectures require considerable compiling efforts and restrictions of the generality of the languages before they can run the problem; consequently, the use of functional languages has been largely confined to small, "academic" applications. Until large, industrial applications are written in a functional language, it will be difficult to objectively evaluate the claims put forth by functional programming advocates. Functional languages are "clean" (or they do not deserve the name!), thus they are limited to equivalence preserving transformations. This is another obstacle to their general adoption, because real applications in data processing require "updating" or persistent state changes, which destroy the clean theoretical base, and therefore destroy clear and easy to comprehend semantics. A pragmatical and operational separation of the different concepts has to be installed

Let us first consider functional languages and then consider their place in the larger picture. What, then, are some of the characteristics of functional languages that advocates claim make them superior to conventional programming languages? The most striking characteristic

of these languages for a conventional programmer is the absence of assignment statements and most control constructs. Functional languages have no notion of a global state or a program counter. The value of a functional expression depends only on its textual context, not on a computational history. The value of an expression is determined only by the values of its constituent expressions. This property is known as referential transparency, and it is the cornerstone of functional programming.

Referential transparency brings programming closer to the world of mathematics -- "functional programs" are compositions of functions in the true mathematical sense. Programming in a functional language is much closer to writing a set of formal rules, either numeric or symbolic, than to conventional programming. Functional programs are primarily concerned with describing what computation is to be done, while conventional programs are more concerned with how a computation is to be done. Another formulation of this is: " ... the underlying concern of a conventional programmer is to guide a single locus of control through a cunningly designed maze of assignment, conditional, and repetitive statements, ... " [KENN84].

Because functional programs behave as mathematical functions, the semantics of functional languages are simple and elegant. This aids in the verification of programs and reasoning about their properties. A small example is in order. We have chosen this example from non-numeric, symbolic programming to emphasize the generality of the approach advocated herein. While the example is given in functional style without

prejudice to a specific language, one must realize, that a functional language as such is usually not equipped to support or even automate such proofs. However, a full and correct implementation of the lambda calculus supports the equivalence preserving transformations needed in this application.

Let us prove that appending lists together is associative. The following example is a private communication by M. Hilton. Here is the definition of append, written as ++, in a representative functional language, where : is the infix list constructor and [] represents the empty list :

```
[] ++ ys = ys                    (1)
(x : xs) ++ ys = x : (xs ++ ys)  (2)
```

We wish to prove that for all lists xs, ys, and zs:

```
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
```

The proof is by structural induction on xs [BURS69].

Base Case: Replace xs by [].

```
([] ++ ys) ++ zs  = ys ++ zs          by rhs of (1)
                  = [] ++ (ys ++ zs)  by lhs of (1)
```

Inductive Case: Replace xs by x : xs

```
((x : xs) ++ ys) ++ zs
                    = (x : (xs ++ ys)) ++ zs  by rhs of (2)
                    = x : ((xs ++ ys) ++ zs)  by rhs of (2)
                    = x : (xs ++ (ys ++ zs))  by induction
                     hypothesis
                    = (x : xs) ++ (ys ++ zs)  by lhs of (2)
```

This completes the proof. The conciseness of this proof demonstrates the semantic "power" of functional languages. While the proof structure is a language property, which is independent of the language implementation, the lambda calculus based machine makes it possible to automate the transformations, substitutions, and rule applications in such proofs while avoiding variable clashes.

Another distinctive characteristic of functional languages is the use of higher-order functions. A higher-order function is one which takes another function as one of its arguments and/or returns a function as its result. For example, the concept of summation -- summing the values for a given function f evaluated at discrete points along the interval between two bounds a and b -- can be expressed by the higher-order function sum:

```
sum f a b next  = 0,                          if a > b
                = (f a) + sum f (next a) b next,   otherwise
```

Next is a function which produces the next point in the interval. Many

mathematical concepts such as integration and Taylor-series expansion, can now be implemented in terms of sum. Higher-order functions make it possible to define very general functions that are useful in a wide variety of applications and not just functions that deal with numbers -- higher-order functions can be used with any data type. This can lead to a substantial reduction in the amount of software necessary to perform significant tasks. Higher order functions are not new, they have been used in LISP with special denotation, and in other functional languages, but there a higher order function can only be returned into a larger context. The production of one function from another one, consisting of nested subfunctions with arbitrary free variables requires the capability to handle free variables correctly with respect to scope and possible name clashes. This is something which only the full, complete and correct implementation of the lambda calculus can provide. This is one case where the proposed lambda calculus machine instruction set provides more functionality than is necessary for the mere implementation of functional languages.

Equipped with capabilities like higher-order functions and verifiable programs how could one not think that functional programming is "the only way to go?" For its advocates there is no other way to go, but for the more pragmatically minded software industry there are several issues which must be resolved if functional programming is to achieve widespread use. Foremost is the problem of execution speed. Functional language implementations have traditionally executed slower on stock hardware than conventional imperative language implementations (we shall explain why in the next section). All of the

wonderful properties of functional languages don't count for much if programs won't execute in an acceptable period of time. Second is the question of whether functional languages are suitable for "real world" applications.

Also, it is unclear if the high productivity attributed to functional programming is due to referential transparency or to other properties such as abstraction, extensibility, higher-order functions or automatic memory management, all of which could be incorporated into more conventional languages. To find out the answer to questions like these it will be necessary to try writing large, "real world" applications in a functional language. But to make this practical, there must exist implementations of functional languages that are of comparable execution efficiency to the conventional languages the software industry is using, which means new implementation technologies will need to be developed for functional languages. In preparation of considering new implementation methods we shall reflect in the next section on the principles of computation. This section is thus conceptual and serves as background material.

2. Models of Computation

As we have mentioned above, the functional approach to computing can only be part of a larger picture. It is therefore necessary to reflect generally on the mechanization of computing. There are in essence three bodies of theoretical knowledge leading to the embodiments of computational machinery.

2.1 The Turing Machine

The Turing Machine, created as an abstract machine by Turing [TURI36] in the nineteen thirties to define computability, may be considered as the conceptual base for what is today known as the von Neumann Architecture. The Turing Machine reads symbols from and stores symbols to a storage medium, while undergoing transitions from state to state. It uses bit strings as symbols. The recognition of a certain state, ensuing transitions, and actions are automated. It is not significant that the storage medium is a tape, but there must be an extendible state space. Conventional computers are generally of the von Neumann type which rely on a programs stored in consecutive memory locations. The program is executed under control of a program counter stepping through these locations. According to the von Neumann principle, data and instructions are stored as words of binary byte data in addressable cells in random access memory. An arithmetic-logic unit performs logical operations (e.g. addition) on the data based on the stored program instructions.

These relate to fetching of the data between the respective memory cells to the arithmetic logic unit, and storing results in other memory cells in the memory whereby state changes are effected.

Computer programming for the von Neumann computers involves keeping track of a multitude of instructions and data and the memory locations in which they are stored, both before they are processed in the arithmetic unit and thereafter. This requires the help of a compiler which translates a user friendly higher level programming language into machine language. A minor error in the details of the program can lead to an inability to identify the specific locations in memory wherein large amounts of data and program instructions are stored. These problems can become quite involved where there are complicated conditional branch structures, recursion, and loops. This inevitably requires extensive debugging with so-called software engineering tools, or even manual debugging on the machine code itself.

2.2 Combinatory Logic

Schoenfinckel [SCHO24] created the Combinator Logic to solve problems centered around the variables in logic (and other computational expressions), their meaning and representation. The representation of a variable (standing for one or more objects) as a string of letters lead to confusion of their meaning because the objects denoted by the string change while transforming the expressions. Schoenfinckel's combinators provided a "variable-free" mechanism to prevent confusion. Because of the close relationship of combinators to functions, Turner [TURN79]

suggested to implement functional languages in terms of a combinator reduction system. Languages based on his system are SASL, KRC, and currently, the latest development, MIRANDA. Computer Architectures emerged, first SKIM [CLAR80], and then NORMA [RICH85] (a Burroughs development). They are based on the combinator reduction system originating from Turner's and Schoenfinckel's work.

Combinator reduction systems have several considerable drawbacks. The applicative source code using variables for the sake of ease of use has to be first compiled into combinator code. The selection of combinators has considerable influence on the size of the resulting object code. The best results known lead to a size increase of A * n * log n, where A is about 10, and n is a measure for the original size. This translates into a roughly ten-fold increase in running time over a machine which could directly execute the source code. Another drawback is the obscure nature of the combinator object code, which has no obvious relationship to the source code which would be discernible by a human being.

Combinator reduction is intrinsically of a weaker nature than reduction with variables, it is "weakly normalizing." This means that the result of a combinator reduction may contain more possibilities of reduction which cannot be done due to the theoretical properties of the reduction system. As a consequence, the power of the language implemented by a combinator reduction system is severely limited to conventional application of functions to arguments, which is already made available by procedural languages. Another drawback of a combinator

reduction system is the replacement of code (forming a redex) in situ by computed code (forming the reductum). Although correct - such replacements do not change the result or meaning of the computation - this method destroys the problem statement. It has to be recompiled or explicitly copied before another run of the reduction process. Also, because of this replacement in situ, the representation needs to be based on a multitude of two cell nodes connected by pointers, and this in turn requires provisions in hardware (marking bits) and software (garbage collection process) to keep track of free and used nodes.

2.3 The Lambda Calculus

In the early nineteen thirties Alonzo Church [CHUR41] created the lambda calculus. This formal system was to be the theoretical foundation for the definition of functions, particularly with respect to the theory of recursive functions and the definition of computability as such. The lambda calculus is about functions with variables, but it goes far beyond the conventional notion of a function, which has a fixed number of formal parameters and expects the same number of actual parameters. Also, conventionally, all formal parameters, and at most all, occur in the body of the function. In contrast, as a matter of fact, the lambda calculus does not "know" about functions in the sense that "function" is a defined entity. The lambda calculus is a simple language with few syntactic constructs. The lambda calculus defines only abstraction, application, and variables.

It has simple semantics, yet it is powerful enough to express all computable functions.

Functional application is, as its name suggests, the capability of applying a function or operator to an argument. In the lambda calculus, application is normally denoted by juxtaposition, with the operator on the left and the argument on the right. As in everyday arithmetic expressions, juxtaposition may be overridden using parenthesis.

Functional abstraction provides the capability of abstracting out particular data from an expression, so that the expression may be used in different contexts with different data. Lambda bindings are the mechanism used to provide abstraction. A lambda binding is signified by the Greek letter, λ, followed by the name of the variable which has been abstracted. This variable is referred to as a bound variable with respect to the following expression from which the bound variable is abstracted. This expression is called the body of the abstraction and is separated from the binding prefix by a period. For example, the expression which adds 3 to a value is (λx. (+ 3 x)). When a lambda binding is applied to an argument the argument is substituted for the bound variable everywhere the bound variable occurs in the body of the abstraction. Continuing with the previous example, applying the expression (λx. (+ 3 x)) to the number 10 yields the expression (+ 3 10), which then yields 13.

Current functional languages employ the lambda calculus' facilities for function application only, while ignoring its powerful abstraction facilities. Because it is strongly normalizing, it can

represent more complicated computations by fewer means.

For all these reasons it is a reasonable objective to implement the full and complete lambda calculus on a system with hard, firm and soft components. We not only get the speed needed to test if functional programming is a viable alternative for the production of "real world" software, we will also have a platform for experimenting with a whole new generation of programming language concepts which exploit the power of abstraction. We therefore propose to construct a small set of new instructions which directly implement the lambda calculus in terms of sequences of such instructions. Thus a significant increase in performance can be obtained, because both the compiling effort as well as the generated object code are substantially decreased in size.

Let us now review some earlier and contemporary work concerning the implementation of functional languages and/or the lambda calculus.

The pioneering work of P. Landin [LAND64] in the sixties introduced first the idea of founding computation on the lambda calculus in terms of the SECD Machine. In the following, implementations of applicative languages based on the SECD Machine turned out to be very slow and not competitive with procedural languages. The inefficiency was so large that the detour using combinators appeared at one point to be more promising than a direct lambda calculus implementation. Following Turner's work, SUPER combinators [HUGH81], the G-Machine [JOHN84], [KIEB84], and TIM [FAIR87] were developed to alleviate the efficiency problems, They did so to some degree by restricting the interpreted code to the conventional use of functions namely, equating the number of

formal parameters to the number of actual parameters and to the set of parameter actual occurring in the function body. The general case, that a function may contain many more parameters, relative free in the body, but bound in various higher, encompassing contexts, is not part of the implementation. Source code which is employing the "general use" needs to be compiled to the conventional function usage, thus the power of general variables is again lost.

A computer based on the lambda calculus has been proposed by Berkling [BERK69]. That computer was intended for use with a new programming language. In the computer, an input channel breaks up the input expression into three cell nodes containing an operation code (e. g. "apply") and pointers to two subtrees. These three cell nodes are stored in a "tablet" which serves as central communication device between a multitude of functional units (e.g. adders) and I/O units. These units have associative access to a subset of nodes, while the tablet is also a shift register shifting nodes cyclically such that all nodes pass by all functional units. If nodes match the input characteristics of functional units, these nodes will be executed concurrently, the results then waiting for passing by target nodes receiving these results.

A computer system employing string reduction based on the lambda calculus was proposed by Berkling [BERK75]. That computer employs a multitude of stack registers holding linearized tree structures in form of sequences of binary encoded node and leaf symbols representing lambda expressions. A program, i.e. a lambda expression, is traversed by shifting these codes up and down the stack registers exposing instances

of reduction rules at the collection of top cells of these stack registers. An instance of a reduction rule (redex) is replaced by its result (reductum) by the reduction processor, which has access only to the top cells of the stack registers and performs state transitions on these top cells, several times if necessary, to accomplish a reduction. Because of its intrinsic structure requiring lots of copying in particular for large data structures, it is too inefficient for present day computing requirements.

Because of its power and simplicity, the lambda calculus is often used as an intermediate language in the implementation of functional languages. Programs in a high-level functional language are translated into lambda expressions which are then compiled into conventional, lower-level machine code. Our approach is similar but requires less compiling effort. The design objective can best be explained by the following metaphor.

3. A Metaphor

The von Neumann computer architecture (in particular modern high-performance, pipelined machines) can be likened to a jet engine, where air intake, compression, combustion, and exhaust follow in sequence continuously, much like instruction fetch, decoding, data fetch, and instruction execution follow in sequence over and over again. In the jet engine, performance peaks when all the parts are in a straight line; in the von Neumann computer performance peaks when all the instructions are in a straight line. It is therefore a main objective to compile all languages to in-line conventional von Neumann computer code. If this is done for a functional language, the result is not much different from a procedural language providing functions and procedures. Conventional procedural languages such as FORTRAN, were designed using the von Neumann architecture as their underlying semantic model; thus they fit von Neumann machines reasonably well and run efficiently. These languages provide a limited ability to structure abstraction and application in terms of expressions and commands, and compilers are needed to convert any "piston" movements to "turbine" movements for efficient execution.

Conventional implementations of genuine functional languages, however, are more like piston engines. Functional languages express computation in terms of expressions which are composed of abstractions and applications. In order to interpret these expressions, conventional

techniques perform up and down movements between the root of an expression and its leaves. An expression is composed of an operator applied to one or more operands. In order to evaluate the expression, a conventional implementation first steps down into the expression and evaluates its operator and its operands. After these have been evaluated, it steps back up to the root of the expression and applies the operator to the operands. Note that the evaluation of the operator and operands is recursive and may require many up and down motions. These up and down motions are more amenable to a "piston engine" computer than a "jet engine" one. These up and down actions are intrinsically less efficient on von Neumann machines because of the continuous testing that must be done in order to determine when and where to reverse direction. These motions are also expensive because each up and down cycle requires the expression be rescanned.

The specific background of the method of dealing with the lambda calculus in this paper is called Head Order Reduction. This method has been especially designed to efficiently embody the lambda calculus. Following the design objective conveyed by the metaphor one would like to represent the lambda expression as "instruction" sequences as long as possible to obtain the jet engine - pipeline effect. The Head Order Reduction method accomplishes just that by recursively building up a lambda expression from linked straight line code. It is therefore necessary to give a short introduction to Head Order Reduction [BERK86].

3.1 Head Order Reduction

An arbitrary lambda expression may be represented in preorder form:

λxn ... λx0 @ ... @ { xj| λy ... } em ... e0 for j,n,m ≥ 0

In general a lambda expression contains a sequence of bindings (λx ...), followed by a sequence of application nodes denoted by the @'s, followed by, what is referred to as the "head". The head can only be a variable (xj) or another lambda expression (λy ...). Following the head are as many lambda expression as there are @'s in the formula. Lambda expressions in these positions are called arguments and are given in the same format.

The operational representation of a lambda expression in a computer must be unique and must protect against the possibility of variable confusion and collision which occur if variables are represented in an inappropriate way, e.g. as character strings. Confusion and collision are standard terms in the theory of the lambda calculus denoting certain fundamental problems. In order to avoid these problems we employ DeBruijn indices [DEBR72], also called binding indices to represent variables in the operational representation of the lambda expressions. This method is a unique, user and machine independent denotation for variables. It avoids confusion and collision of variables.

The binding index method is described as follows: The value of the index standing in for a variable x is the number of bindings (of other variables) located on the path in the expression tree between its

occurrence x and its binding occurrence λx.

For example, the lambda expression:

λx λy λz @ @ (@ (@ x y) z) (λw w)

transforms using binding indices as follows:

λ λ λ @ @ (@ (@ 2 1) 0) (λ 0)

Although different variables (w, z) may be represented by equal indices (0) and the same variable may have for different occurrences different index values, the representation is unique and depends only on the structure of the lambda expression. It lends itself to the implementation technique described herein.

The general form of the lambda expression given above

λxn ... λx0 @ ... @ { xj| λy ... } em ... e0 for j,n,m ≥ 0

corresponds to this binding index form:

H => λ A
A => λn @m { # | H } Am for n,m ≥ 0

Here the distinction between H (for head) and A (for argument) is that the head expression must begin with a lambda. The sharp sign

denotes an index, superscripted indices denote the multiplicity of occurrence.

By using the syntax rules repeatedly we arrive at the general form for every lambda expression where the superscript * means any number $n \geq 0$:

$$\lambda^* @^* \lambda^* @^* \ldots\ldots \lambda^* @^* \# A^*$$

The part up to and including the head variable # is called the spine and plays a major role in the novel method described here. To visualize the spine as graph we represent a sequence of bindings (λ^*) by a 45 degree line from the upper left to the lower right, and a sequence of application nodes ($@^*$) by a 45 degree line from the upper right to the lower left. Thus a terse, graphical representation of a general lambda expression is a zigzag line as shown in Figure 1. The A's are not shown. The complete representation would be a recursive nesting of zigzag lines.

The employment and embodiment of the lambda calculus as a system of computation requires that complex lambda expressions be reduced to simplified equivalent forms which are called normal forms. Two reduction rules have been developed for simplifying complex lambda expressions, they are:

Beta-Conversion Rule B = (λx . M) N ; B reduces to [N/x]M, where N and M are arbitrary lambda expression containing free variables and where [N/x]M means the consistent replacement of N for all occurrences of x in M, whereby means and precautions are taken to avoid confusion

of variables.

Eta Conversion Rule $E = \lambda x . (M\ x)$; E reduces to M, where M is an arbitrary lambda expression containing free variables except the variable x.

A lambda expression is called reducible if it contains at least one instance of either a Beta-Conversion Rule, which is called a beta-redex, or an instance of Eta Conversion Rule, which is called a eta-redex. An expression which does not contain any redices is said to be in Normal Form.

Returning to the graphical representation of an arbitrary lambda expression in Figure. 1, it is observed that the zigzag line structurally has several corners associated with it. In Figure. 1, the corners projecting to the right are called betas-aps and those projecting to the left are called aps-betas. The detailed structure of the corner corners is illustrated in Figure. 2. The corner of an aps-betas is a beta-redex. The corner of the aps-betas in Figure. 2 is the beta-redex $(\lambda a . M)\ Aa$, where M represents the remainder of the zigzag line which is a lambda expression.

No redices are associated with betas-aps. Thus, the zigzag line, or expression that it represents, will be smoothed out or transformed into a single betas-aps graph, or equivalent expression by executing Beta-Conversions (or beta-reductions).

A transformation technique that accomplishes this objective is referred to as beta-reduction-in-the-large, because the set of single beta-reductions associated with a aps-betas corner is considered as one reduction step.

Beta-reduction-in-the-large can be described in terms of the graphical representation of an arbitrary lambda expression. A maximal aps-betas corner is cut from the zigzag line and moved down the zigzag line up to the next sequence of betas. This graph manipulation does not change the meaning (i.e., it is an equivalence preserving transformation) of the expression as long as a copy of the cutout aps-betas corner is inserted, as a prefix, before all arguments pending from the sequence of aps located between the original and final position of the cutout.

Beta-reduction-in-the-large is illustrated in Figure. 3. The cut C-C in Figure. 3 is the maximum possible aps-betas grouping in the upper most part of the zigzag line. The letters a through r represent arbitrary lambda expression (arguments and variables) pending from the zigzag line.

The graph on the right in Figure 3 illustrates that in the transformation, the cut C-C has been moved down the zigzag line to the farthest possible position, namely before the next set of bindings q and r. In addition, the cut C-C has to be inserted as a prefix before all pending argument k to p. The insertion is denoted by underlining in Figure 3, except for argument m where, as an example, the inserted cut is explicitly shown.

In the example of Figure 3, the betas are exhausted before the aps. An example where the aps are exhausted before the betas is shown in Figure 4. In this case, a transformation cannot be executed because the immediately following betas prohibit any downward move. But an extension of the cut C-C, as shown in Fig 4. to the cut D-D allows to capture the betas which are in the way. This technique is called eta-extention-in-the-large and is graphically accomplished by inserting the new betas-aps λjλk @ @ ... j k in the zigzag line such that the aps @ @ ... j k can be taken together with the betas which are in the way to form the new cut D-D. Eta-extention is the application of the Eta-Conversion-Rule in reverse. Eta-extention-in-the-large is a repeated application of the Eta-Conversion-Rule in reverse.

As can be seen from Figure 3 and Figure 4, the application, beginning at the top of the zigzag line, of beta-reduction-in-the-large combined with eta-reduction-in-the-large combined with eta-extention-in-the-large , where appropriate, transforms an arbitrary zigzag line into a single betas-aps-betas-# graph. There is now one more aps-betas cut to make, but it sits just in front of the variable # (assuming DeBruijn index representation for variables). It obviously cannot be moved downward any further.

A special treatment of the head variable #, however, makes the continuation of the computation possible. Considering the betas-# portion of the transformed expression, one can see that it works as a selector on the preceding aps. A selector is a betas-# and has the

detailed structure:

$$\lambda x1 \lambda x2 \ldots \lambda xn \; . \; xm$$

The transformed expression therefore contains an application of a selector to some aps, respective some arguments:

$$(\lambda x1 \lambda x2 \ldots \lambda xn \; . \; xm \;) \; a1 \; a2 \ldots an \Rightarrow am \quad \text{for } 1 \leq m \leq n$$
$$\Rightarrow xm \quad \text{otherwise}$$

The application reduces either to a new argument am, which is simply another zigzag line and the process of headorder reduction continues, or the variable xm.

The selector in DeBruijn index form is very simple, namely $\lambda n \; . \; m$. It can be conveniently implemented as an indexed access of an array of length n of arguments by an index m. The arrangement of arguments in such an array, which is called an environment is part of the implementation as explained later.

Finally, the result is a betas-aps-# corner, which is called the head-normal-form. Except in its arguments, it does not contain any more redices. This resulting skeleton structure has the important property that it will not be altered by later transformations within the arguments, no matter what reduction sequences take place in the arguments pending to the right of the head-normal-form. Moreover, these arguments are independent from one another, i.e. no conversion rule application in the

arguments will cause any two of these arguments to interact. Thus, this independence suggests an implementation whereby the order in which the arguments are reduced is immaterial; moreover, the reduction of these arguments may be performed concurrently. This property deserves further investigation with respect to the availability of parallel computers. The reduction of an argument takes place by recursively applying the method just described.

The reduction method applied herein is termed head-order-reduction and is closely related to normal-order-reduction. In contrast, Head order reduction does not reduce the beta-redices one by one separately, but rather employs an environment. The notion of the "head-normal form" has been introduced by Wadsworth [WADS71].

The prefix portion of the arguments, that is the collection of cuts accumulated in front of the original expressions, can be conveniently represented as a pointer into an environment shared by several arguments. The tuple formed of an environment and an argument expression, respective pointers to them, is generally called closure. The implementation of head-order-reduction preserves this sharing property. Since environments expand and shrink when changing from one argument to another, a naive sharing of environments would lead to corruption. To solve this problem, an implementation approach must include proper control over the shrinking or cutting back, and the restoring of environments.

Because of its encompassing nature, the lambda calculus reduction

system can emulate a combinator reduction system and make it appear strongly normalizing. The reason for this is that combinators are representable by special lambda expressions of a form such that a multitude of bindings is prefixed to an applicative structure containing only application nodes and only variables which occur in the bindings. The example in the appendix shows a lambda expression first compiled into a combinator expression. This expression is then strongly normalized to a lambda expression. (Weakly normalizing would terminate earlier with a more complicated expression). Finally, the original lambda expression is directly reduced to the same small lambda expression, proving the correctness of both approaches.

This demonstrates how very little is accomplished by reducing one combinator. A large number of combinators, however, is needed to represent a computation. Not only is a non-trivial compiling effort required to compose the combinator expression, but the increased size of it alone uses more memory cycles than head order reduction. Thus a combinator reduction system is intrinsically more inefficient, and its implementation is clearly a lengthy and costly enterprise [RICH85].

In a final remark to the conceptual background we observe that the headorder reduction scheme may be considered as another "programming" of a Turing machine, where the problem instance is the input expression and the reduced expression is generated as result instance following the problem instance on the tape, which is of course replaced by a random access memory.

II.

BIBLIOGRAPHY

[APPE87] Appel, A. W . and D. B. MacQueen. "A Standard ML Compiler," *Functional Programming Languages and Computer Architecture,* Lecture Notes in Computer Science 274, Springer-Verlag, 1987, pp. 301-324.

[AUGU84] Augustsson, L. "A Compiler forLazy ML," *Proc. 1984 ACM Symposium on Lisp and Functional Programming,* 1984, pp. 218-227.

[BARE72] Barendregt, H.P. *The Lambda Calculus, its Syntax and Semantics,* North-Holland Publishing Co., Amsterdam, Netherlands, 1981, 1984.

[BERK69] United States Pat. No. 3,646,523 "Computer," 1969.

[BERK75] Berkling, K. J. "Reduction Languages for Reduction Machines," *Proc. IEEE International Symposium on Computer Architecture,* Jan.1975, pp.133-140.

[BERK86] Berkling, K. J. "Head Order Reduction: A Graph Reduction Scheme for the Operational Lambda Calculus," *Proc.of the Santa Fe Graph Reduction Workshop,* Lecture Notes in Computer Science 279, Springer-Verlag, 1986, pp. 26-48.

[BURS69] Burstall, R.M . "Proving Properties of Programs by Structural Induction," *The Computer Journal*, Vol. 12, No.1, Feb 1969.

[CHUR41] Church, A. *The Calculi of Lambda Conversion.* Princeton University Press 1941.

[CLAR80] Clarke, T.J.W., Gladstone, P.J.S., MacLeen, C.D. and Norman, A.C. "SKIM-The S, K, I Reduction Machine". *Record of the 1980 LISP Conference*, Stanford, California.

[DARL81] Darlington,J. and M .Reeve. "ALICE - A Multiprocessor Reduction Machine for Parallel Evaluation of Applicative Languages," *Proc. ACM Conference on Functional Programming, Languages, and Computer Architecture*, 1981, pp. 65-75.

[DEBR72] DeBruijn, N. G. "Lambda-Calculus Notation with Nameless Dummies: A Tool for Automatic Formula Manipulation with Application to the Church-Rosser Theorem," *Indag. Math.*, Vol. 34, 1972, pp. 381-92.

[DENN79] Dennis J.B. "The Varieties of Data Flow Computers," *Proc. IEEE International Conference on Distributed Systems*,1979, pp. 430-439.

[FAIR87] Fairbairn, J. and S. Wray. "TIM: A Simple, Lazy Abstract Machine to Execute Supercombinators," *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science 274, Springer-Verlag 1987, pp. 34-45.

[GREE85] Green, K.J. "A Fully Lazy Higher Order purely Functional Language with Reduction Semantics." Case Center Technical Report No. 8503, Syracuse University, 1985.

[HEND80] Henderson, P. *Functional Programming*. Prentice-Hall, 1980.

[HUGH82] Hughes R. J. M . "Super-Combinators: A New Implementation for Applicative Languages," *Proc. 1982 ACM Symposium on Lisp and Functional Programming*, 1982, pp. 1-10.

[JOHN84] Johnson, T. "Efficient compilation of lazy evaluation." In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pp. 58-69, Montreal, 1984.

[KENN84] Kennaway,J.R. and M .R. Sleep, "The 'Language First' Approach," *Distributed Computing*, Academic Press 1984.

[KIEB84] Kieburtz, R. B. *The G-machine: A Fast Graph-Reduction Processor*. Oregon Graduate Center, Technical Report 84-003, 1984.

[LAND64] Landin,P.J. "The Mechanical Evaluation of Expressions," *The Computer Journal*, Vol. 6, No.4, Jan 1964.

[PEYT871] Peyton Jones,S. L. *The Implementation of Functional Programming Languages*. Prentice-Hall,1987.

[PEYT872] Peyton Jones, S.L., et al. "GRIP- A High-Performance Architecture for Parallel Graph Reduction," *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science 274, Springer-Verlag, 1987, pp. 98-112.

[RICH85] Richards, H. *An Overview of Burroughs NORMA*. Austin Research Center, Burroughs Corp., Austin TX, May 1985.

[SCHO24] Schoenfinckel, M. "On the Building Blocks of Mathematical Logic," 1924 , *From Frege to Godel: A Sourcebook in Mathematical Logic*, van Heijenoort Ed., 1967, pp. 355-366.

[STOY85] Stoye,W . R. *The Implementation of Functional Languages Using Custom Hardware*. Ph.D. Thesis Computer Laboratory, University of Cambridge, May 1985.

[TOYN86] Toyn,I. and Runciman, C. "Adapting Combinator and SECD Machines to Display Snapshots of Functional Computations," *New Generation Computing*, Vol.4, 1986, pp. 339-363.

[TURI36] Turing, A. M. "On computable numbers, with an application to the Entscheidung's problem," *Proc. London Math. Soc.*, Ser. 2-42, 1936, pp. 230-265.

[TURN79] Turner, D.A. "A New Implementation Technique for Applicative Languages," Software Practice and Experience, Vol.9, Sept 1979, pp .31-49.

[VEGD84] Vegdahl, S.R. "A Survey of Proposed Architectures for the Execution of Functional Languages," *IEEE Transactions on Computers*, Vol. C-23, No.12, Dec 1984, pp.1050-1071.

[WADS71] Wadsworth, C. P. *Semantics and Pragmatics of Lambda-Calculus*. Ph.D. Thesis, Oxford University, 1971.

[WATS82] Watson, I. and J. Gurd. "A Practical Data Flow Computer," *IEEE Computer*, Vol.15, Feb.1982, pp.51-57.

[WATS87] Watson, P. and I.Watson, I.. "Evaluating Functional Programs on the FLAGSHIP Machine," *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science 274, Springer-Verlag, 1987, pp. 80-97.

Appendix

********** THIS IS LBTRD-100 **********910430

<expr>
tqw

(-7 (-4 7 (-3 (-4 7 4 (-1 (-4 3 1 (2 9 11)) 5)) (-4 5 1) (-2 3 4)
2 3) 4 (-1 6 5)) (-4 5 1) (-2 3 4) 2 3)
abstraction

---

reds	nodes	maxcln	enmc	sec
0	**75**	0	0	**3**

(k (k (k (w3 (cc (cc c)) (b (w3 (cc c)) (cc (bb (ss c)) (cc (ss c)
(cc (bb (ss (kk (kk (kk c))))) (b c (c b (w3 (cc (cc c)) (b (w3
(cc c)) (cc (ss (ss c)) (cc (cc (bb c)) (c (bb (ss (kk (kk (kk
k3))))) (b (w3 k3) (bb (bb (cc (bb (cc k3)))) i (cc c r)))) (k3
(k3 k3))) (c (kk k3)))))) k3) (k3 (k3 k3))) (c (kk k3))))))))
all reductions

---

reds	nodes	maxcln	enmc	sec
211	**14**	**323**	**1700**	**5**

(-7 3 (-3 6 0 (-4 4 1 (2 4 6))) 0 (-1 2 1))
tqw

(-7 (-4 7 (-3 (-4 7 4 (-1 (-4 3 1 (2 9 11)) 5)) (-4 5 1) (-2 3 4)
2 3) 4 (-1 6 5)) (-4 5 1) (-2 3 4) 2 3)
all reductions

---

reds	nodes	maxcln	enmc	sec
9	**14**	**26**	**177**	**1**

(-7 3 (-3 6 0 (-4 4 1 (2 4 6))) 0 (-1 2 1))

reds: reductions used nodes: nodes generated
maxcln: maximal stack depth sec: actual runtime (MACII)

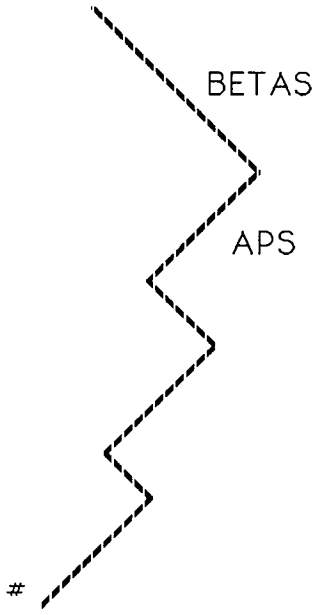
BETAS
APS
#

Fig. 1

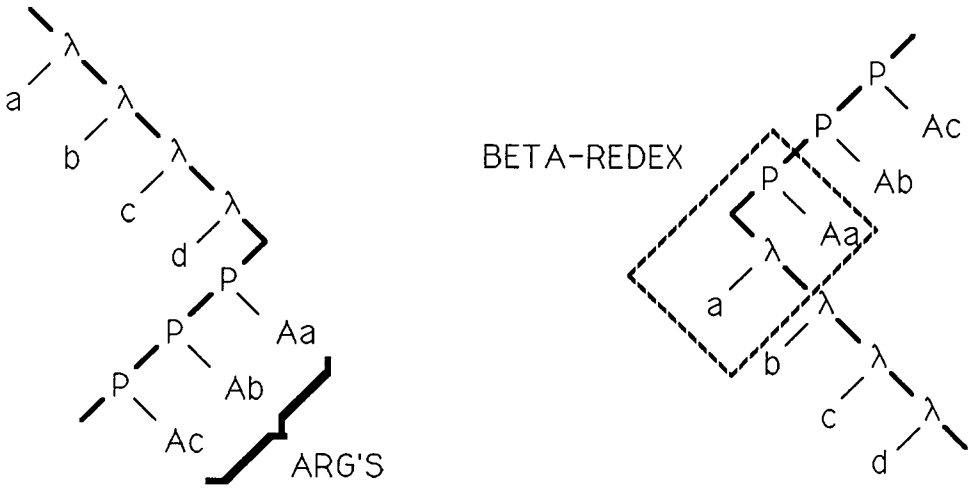
λ
a
λ
b
λ
c
λ
d
P
P
Aa
P
Ab
Ac
ARG'S
BETA-REDEX
P
P
Ac
P
Ab
Aa
λ
a
λ
b
λ
c
λ
d

Fig. 2

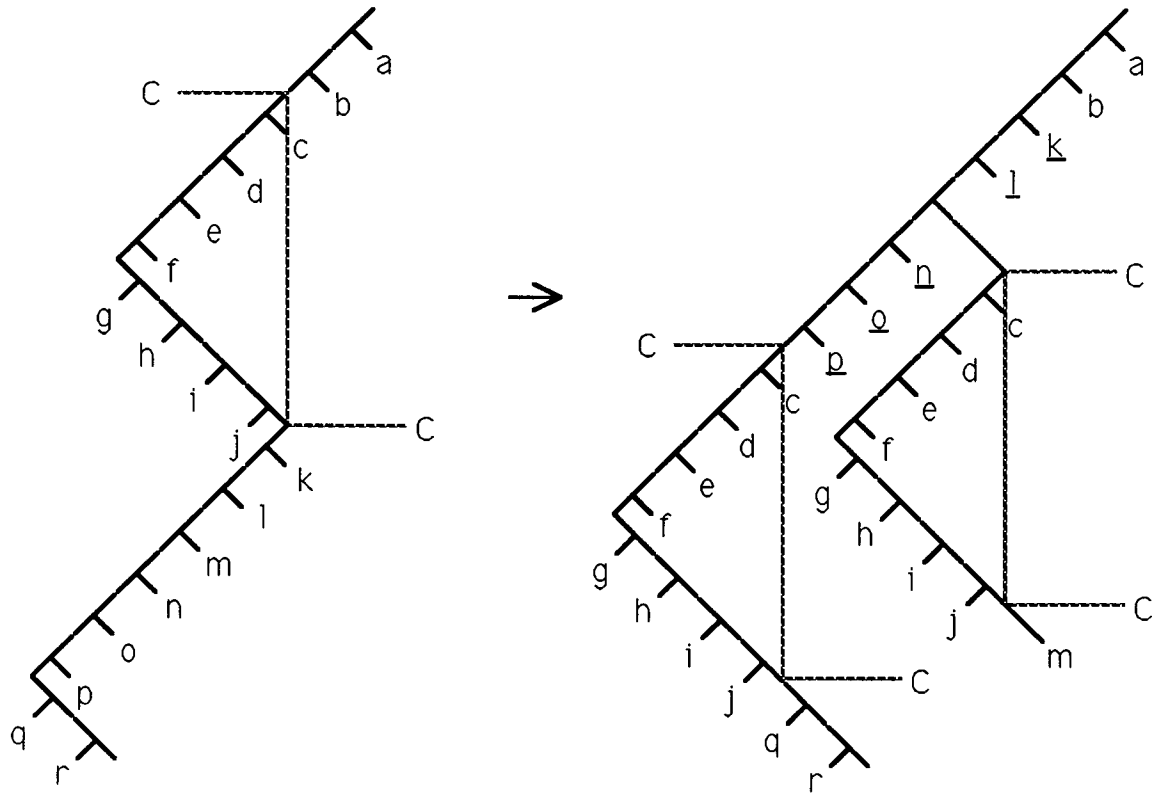
C
a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
q
r
C
C
a
b
k
l
n
C
c
d
e
f
g
h
i
j
C
m
C
o
p
c
d
e
f
g
h
i
C
j
q
r

Fig. 3

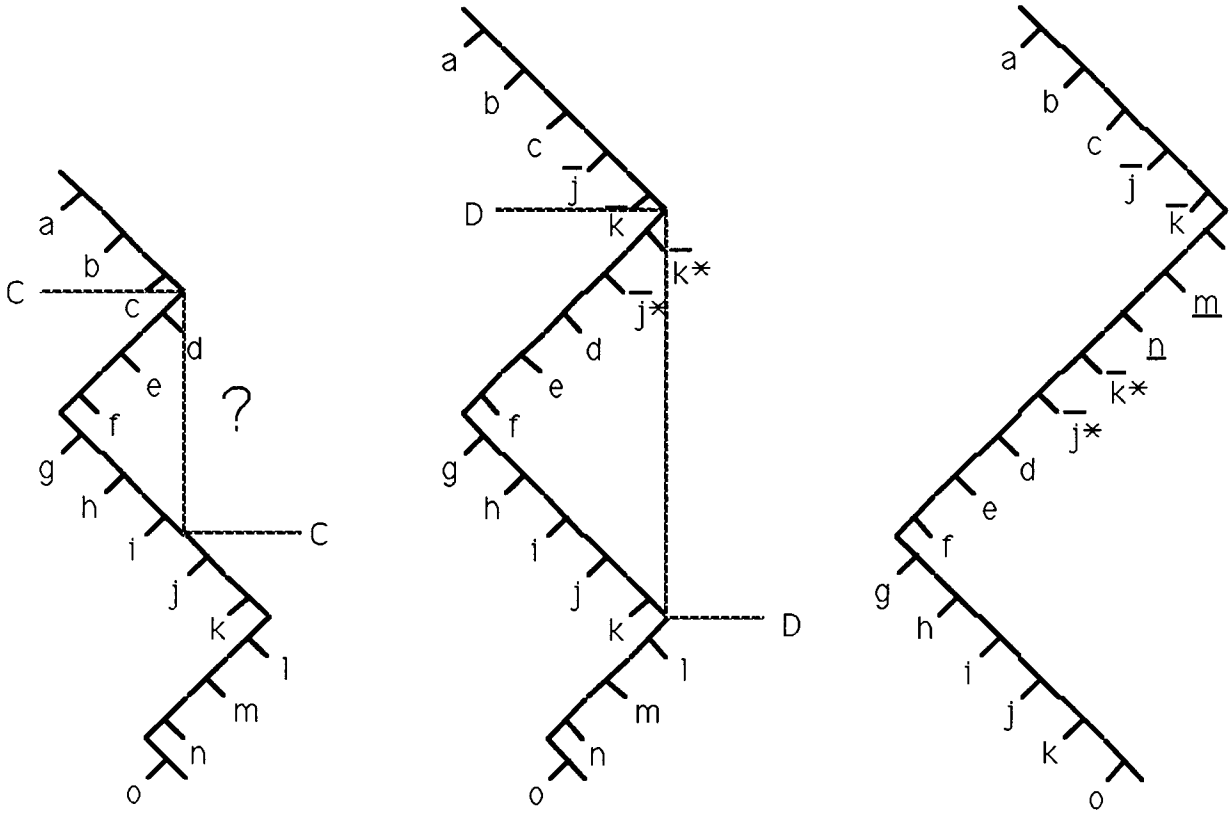
a
b
C
c
d
e
?
f
g
h
C
i
j
k
l
m
n
o
a
b
c
j
D
k
k*
j*
d
e
f
g
h
i
j
k
D
l
m
n
o
a
b
c
j
k
m
n
k*
j*
d
e
f
g
h
i
j
k
o

Fig. 4