5-1990

Arrays and the Lambda Calculus

Klaus Berkling

SU-CIS-90-22

Arrays and the Lambda Calculus

Klaus Berkling

May 1990

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

ARRAYS AND THE LAMBDA CALCULUS

Klaus Berkling
CASE Center
and
School of Computer and Information Science
Syracuse University
Syracuse, NY 13244-1190

May 1990

Keywords: Array processing, functional programming, lambda calculus, reduction machines.

INTRODUCTION

Why do functional languages have more difficulties with arrays than procedural languages? The problems arising in the designing of functional languages with arrays and in their implementations are manifold. They can be classified according to 1) first principles, 2) semantics, 3) pragmatics, and 4) performance. This paper attempts to give an outline of the issues in this area, and their relation to the lambda calculus. The lambda calculus is a formal system and as such seemingly remote from practical applications. However, specific representations and implementations of that system may be utilized to realize arrays such that progress is made towards a compromise of 1) adhering to first principles, 2) clear semantics, 3) obvious pragmatics, and 4) high performance. To be more specific, a form of the lambda calculus which uses a particular representation of variables, namely De Bruijn indices, may be a vehicle to represent arrays and the functions to manipulate them.

PROBLEM

Serious computational problems come almost always in terms of arrays. Nothing seems to be more appropriate to array processing than the von Neumann Computer Architecture. Its linear memory structure, its addressing methods (absolute address + offset + index, base registers, offset registers, index registers), and its instruction set obviously constitute a perfect base to implement arrays. Together with

language features originating from FORTRAN, the system indeed fills basic needs in all four categories stated above. Why would or should one try to improve on it? Because only very basic needs are filled.

For the sake of efficiency, the higher level concepts of programming languages had to be compromised. Real problems are dealing with very large matrices and/or very frequently used matrices, for example for pointwise transformations in image processing. There is no time (or space!) for costly procedure or function entries (prologues) and exits (epilogues), for scoping of variables and blockstructures, but there is a tendency for rather flat programs and code. This becomes manifest in terms of complicated nests of DO-loop constructs and difficult to follow GOTO's. A premium is on the removal of code from more frequently executed innermost DO-loops to less frequently executed outer DO-loops or initialization pieces by program transformations.

The combination of flat code, flat name space, static memory layout, and the assignment statement as basic programming tool causes very unpleasant properties. Since index variables are usually considered first class citizens, occurrences of, for example, $A[I, K]$ may refer to different matrix elements depending on the code executed between these occurrences. Assignments to I and K may destroy the referential transparency. The scope structure is intricately mapped onto sequencing. As a consequence, variables can be easily confused, alterations become a costly, error prone, and time consuming enterprise.

It is particularly difficult to transform such flat programs into "parallel" programs, which make inherent concurrency explicit. Obviously, scoping cannot be expelled without penalty. If it is not part of the language, the compiler must symbolically execute the code to reconstruct dependencies.

Many of the problems to produce efficient and accurate code without the base of higher level concepts arise from the utilization of memory locations. The conventional viewpoint of a memory location is best described as a "box-variable," as opposed to a "place-holder-variable," which is the viewpoint taken by functional languages. This difference makes it difficult for functional languages to adopt easily array features from conventional languages. Four operations are associated to box variables: allocate (a name to refer to the box is created), deallocate, fill the box, read the contents of the box. In

contrast to this, place holder variables are either attached to an expression by function application, or an occurrence of the place holder variable is replaced by the attached expression. After this replacement, neither the place nor the expression is any longer accessible through the variable name. Replacements of several variable occurrences by atomic expressions are not difficult to implement. However, a compound expression has to be shared among several occurrences of the place holder variables, each of which is replaced by a pointer rather than by a copy of the expression. The complexity of copying just cannot be afforded.

The computational problems are rarely such that matrices and vectors can be treated as conceptual units only without ever accessing single elements explicitly. The worst case to consider is the change of one element. While a straight forward functional approach would create a copy of the old matrix with one element changed, the conventional solution simply stores a new value into the appropriate location treating it as a box variable. Conceptually the old place holder variable has been replaced by a new one, although there is still the same identifier and the same pointer to the same memory area. Because of sharing as explained above, however, the change may be more globally effective than planned. The scope of variables has been mapped onto the time scale.

One might take the position to create always new matrices in new memory areas because it is compatible with place holder variables, conceptually clean, and there is enough memory. But the challenge remains to overwrite matrices which are not accessed anymore. It just seems more rewarding to solve problems requiring larger matrices when larger memories become available.

The problems of embedding consistent and elegant array processing into a functional language do not preclude pragmatic solutions. LISP and LISP-like programming languages already incorporate assignment statements and the program feature. One needs only to extend these concepts by functions and "pseudo" functions to make an array, to set a selected element of an array, and to reference a selected element. All arguments may be expressions in accordance with functional language concepts. This is a rather conventional low-level add-on.

The language which really constitutes an advance is APL. Arrays are treated as conceptual entities, functions are modified by operators and extended over the array elements. Thus the power of higher order

functions is obtained and the access to array elements is delegated to the system, if possible.

There is another system originating from physics, which excels in omitting unnecessary detail, like the indication of dimension, summation signs, and bindings. We have the "Einstein" convention that equal indices indicate summation. The "inner" product of two vectors a_k and b_i, where i and k assume all index values $O \ldots n-1$ or $1 \ldots n$, is then simply denoted by $a_j b_j$. There is no free index, thus the result is a scalar. In the same spirit, $a_{ik} b_{kj}$ denotes the matrix product c_{ij}, summation over k produces the inner products of rows i and columns j. The identity matrix is denoted by δ_{ik}, the Kronecker symbol. ($\delta_{ik} = 1$ if $i = k, = O$ otherwise). There are so-called ϵ-tensors, which produce "outer products" $\epsilon_{ik} a_k = b_i$, such that $a_j b_j = O$. In general we have $\epsilon_{ijk} \ldots = +1$ for an even permutation, and $\epsilon_{ijk} \ldots = -I$ for an odd permutation of indices, and $\epsilon_{ijk} \ldots = 0$ for all other cases.

For $n = 2$ we have $a_j b_j = a_j \epsilon_{jk} a_k = a_1 a_2 - a_2 a_1 = 0$. This means b_j is orthogonal to a_i. For $n = 3$ we have $c_i = \epsilon_{ijk} a_j b_k$, which denotes a vector c_i orthogonal to a_j and b_k because the index i is free. Its first component is:

$$c_1 = a_2 b_3 - a_3 b_2$$

Finally, the determinant of a matrix a_{ik} is denoted by :

$$\epsilon_{ijk} a_{1i} a_{2j} a_{3k}$$

Since there is no free index, a scalar is denoted. While the necessary index sets are well defined, very little is given in details how to produce them. Common to APL and the above scheme is the existence of an algebra. The computation of an atomic value from an arithmetic statement can be substantially improved in terms of the number of needed steps and intermediate storage requirements by prior algebraic transformations.

The same approach using matrix algebra will yield savings in terms of intermediate storage, too. But the computation is still expressed in terms of matrices as conceptual units. Whole matrices would have to be stored intermediately, and the challenge to reuse this space by overwriting is also there.

However, by rearranging the sequence of computation in terms of of single elements, it is possible to allocate space only for the input

array and the output array. Then, if the elements of the output array are denoted by an expression in terms of elements of the input array, intermediate storage is needed only for a number of single elements. Moreover, the expression denoting an output element is invariant except for index values. This concept has numerous desirable properties. The input array is only used in read-only mode, the output array is only used in write-only mode. Only the set of intermediate single element cells is used in both modes. This arrangement is very suitable to concurrent processing: Suppose a copy of the input array as well as the skeleton of the expression exists in a multitude of processors. Then each of the processors can concurrently compute a subset of elements of the output array free of interference. What is different from process(or) to process(or) is the set of index values.

The value of L. Mullin's mathematics of arrays [5] is based on the ability to transform the computation of array expressions by formal algebraic methods. The objective of these transformations is the optimal use of resources. The effect of these transformations is that elements of the result array are denoted in terms of elements of the input array. Intermediate structures are not explicitly generated.

Is there any overlap of mathematics of arrays and the lambda calculus? At first sight, the answer might be: none. But, there is certainly one application, namely, the correct implementation of scope structure for array names (and index variables?).

In the following sections we will consider other possibilities which are worth exploring. We assume familiarity with the lambda calculus. The experimental system used in the discussion is based on [3].

FIRST PRINCIPLES

The denotation of an array element like $A[IK]$ has similarity with the denotation of a function value, but only if it appears on the right side of an assignment statement. If it appears on the left side, however, it denotes a location. The denotation of a function value is compatible with the notion of a function represented by a table of ordered tuples. In terms of first principles, the occurrence of such a denotation on the left side of an assignment statement implies a resetting of a table entry, which is a function changing operation, that is, the assignment statement implies a higher order function in this context! This does not seem appropriate. Let us consider it as a datastructure. So, the question is now: What is a data structure?

Datastructures are generally input and output of programs. One may consider datastructures as code which does not require execution anymore, because there is nothing left to be done. This wording has been deliberately chosen to relate it to reduction and the lambda calculus. Reduction is a first principle and means the substitution of symbols by symbols which are equivalent with respect to a set of rules.

The result of reducing a lambda calculus expression consists of nested head normal forms. A head normal form (HNF) is an expression starting with a number of bindings, followed by a number of application nodes, and terminating with a variable. An example of a HNF is shown in tree form:

$$\begin{array}{ll} \lambda a \lambda b \lambda c.- & @ - \cdots \\ & @ - \cdots \\ & @ - \cdots \\ & b \end{array}$$

Redices, which are instances of possible rule applications, are not visible. If there are any, they would be in the argument expressions $(\ldots)$ of the application nodes. Thus, further processing does not alter the HNF once it is constructed. This conclusion applies recursively to all lower level arguments and HNFs until a HNF degenerates to a constant or variable.

We have now recognized the result of a lambda calculus computation, or better reduction, to be a datastructure represented as nested HNFs. Can such a datastructure be the argument of a lambda expression? This does make sense only if all substructures of the datastructure can be selected once it is substituted into this lambda expression. A selector in terms of the lambda calculus is a degenerated HNF without application nodes, for example:

$$\lambda a\ \lambda b\ \lambda c\ \lambda d.\ b$$

The name "selector" is justified, because the expression above selects from 4 arguments the third one, or when counting starts at 0, the second one. Note that we count from left to right, or top down, respectively.

$$
\begin{array}{l}
@ - A_0 \\
@ - A_1 \\
@ - A_2 \\
@ - A_3 \\
\lambda a \lambda b \lambda c \lambda d.\ b
\end{array}
$$

The identifiers denoting variables do not carry any information. De Bruijn indices count the lambdas between variable occurrence and its binding. Thus, the information of the selector is completely given by

$$\lambda\lambda\lambda\lambda.2$$

or even more concisely by

$$\lambda 4.2$$

which is in terms of array accessing a pair of range and index values. In order to make the selector actually "select," it has to be substituted in place of the head variable of a datastructure. The sequence of reductions is shown here:

$$
\begin{array}{lll}
 & @ & -\lambda 4.2 \\
\gg & | & \\
 & \lambda 1. & -@ - A_0 \\
 & & \quad @ - A_1 \\
 & & \quad @ - A_2 \\
 & & \quad @ - A_3 \\
 & & \quad 0
\end{array}
$$

The beta-redex ($\gg$) places the selector first into the head position. Then a sequence of beta-reductions accomplishes the selection of the argument A_2:

$$
\begin{array}{llll}
@ - A_0 & @ - A_0 & @ - A_0 & @ - A_0 \quad A_2 \\
@ - A_1 & @ - A_1 & @ - A_1 & | \\
@ - A_2 & @ - A_2 & | & \lambda 1.A_2 \\
@ - A_3 & | & \lambda 2.A_2 & \\
| & \lambda 3.2 & & \\
\lambda 4.2 & & &
\end{array}
$$

This method of applying a datastructure to a selector was discovered very early by W. Burge [4]. He coined the term "functional"

datastructure. But, one can emulate the application of the selector to the datastructure by using the combinator R and write:

$$R\ \lambda 4.2 < \text{datastructure} >$$

If a datastructure consists of nested HNFs, the need for a compound selector arises. It is, of course, a HNF with simple selectors as arguments, for example:

$$\begin{array}{lll} \lambda 1.- & @ - \lambda 5.4 & \\ & @ - \lambda 6.1 & \\ & @ - \lambda 4.2 & \\ & 0 & \end{array}$$

The application of this compound selector to a nested datastructure causes the latter first to appear in its head position. Then a selection takes place in the toplevel HNF using $\lambda 4.2$. The next lower level HNF now forms a beta-redex with $\lambda 6.1$.

$$\begin{array}{ll} & @ - A\lambda 5.4 \\ & @ - A\lambda 6.1 \\ \gg & | \\ & \lambda 1. - @ - \cdots \\ & \quad \cdots \end{array}$$

This sequence of events repeats itself recursively until the primitive selectors are used up.

The dual nature of what is left subtree (operator) and what is right subtree (operand) of an application node becomes even more evident, if we look now at the construction of compound selectors. The combinator R applied to a simple selector yields an operator on datastructures:

$$\begin{array}{lll} (R\lambda m.n) \Longrightarrow \lambda 1.- & @ - \lambda m.n & \\ & 0 & \end{array}$$

The operation of functional composition makes a compound selector from a set of simple ones. The combinator B does it for two,

$$Bfg \Longrightarrow f \circ g$$

The generalization to $n-1$ functions is the set of combinators

$$\begin{array}{lllll} B_n = \lambda n.- & @ - \cdots & @ - 0 \\ & n-1 & 1 \end{array}$$

such that

$$B_n f_{n-1} \ldots f_1 \text{ arg} = f_{n-1}(\ldots(f_1 \text{ arg}))$$

For more on sets of parameterized combinators see ABDALI [0]. The compound selector given as an example above can be constructed from the three simple ones using B_4:

$$B_4(\lambda 1.0\ \lambda 5.4)\ (\lambda 1.0\ \lambda 6.1)\ (\lambda 1.0\ \lambda 4.2)$$

This expression reduces to:

$$\begin{array}{ll} \lambda 1.- & @ - \lambda 5.4 \\ & @ - \lambda 6.1 \\ & @ - \lambda 4.2 \\ & 0 \end{array}$$

which is what we expected. Dual to the "functional" composition combinators B_n, which arrange a sequence of arguments in successive operator positions, there are also "data" composition combinators T_n, which make HNFs, that is, arguments are arranged in successive operand positions. The general pattern is:

$$\begin{array}{lll} T_n = \lambda n.- & @ - 1 \\ & @ - 2 \\ & \cdots \\ & @ - n - 1 \\ & 0 \end{array}$$

The dual nature of datastructures and function makes

$$B < \text{datastructure1} >< \text{datastructure2} >$$

meaningful, too. However, there is a peculiar twist in the representation of expressions. The tree representation is related to the linear representation by preorder traversal. On the other hand, arguments are numbered top down in trees, because this makes it compatible with the sequence De Bruijn indices referring to them. Thus, elements of a datastructure are numbered from left to right in the linear representation.

Suppose a $n \times n$ matrix is given as a column of rows, which is a datastructure of n elements, where each element is a datastructure of n elements representing a row. Since this is not a very convenient

way of storing a matrix for accessing and a row major order vector representation is preferable, we would like to transform the structure.

$$
\begin{array}{llllll}
m = \lambda 1.- & @ - - - - - - & - - - - & - - \lambda 1.- & @ - a_{11} \\
 & | & & & @ - a_{12} \\
 & @ - - - - & - - \lambda 1.- & @ - a_{21} & @ - a_{13} \\
 & | & & @ - a_{22} & 0 \\
 & @ - \lambda 1.- & @ - a_{31} & @ - a_{23} & \\
 & | & @ - a_{32} & 0 & \\
 & 0 & @ - a_{33} & & \\
 & & 0 & &
\end{array}
$$

Applying this matrix representation to B_4 would place row 3 on top of the tree. To reverse the order we introduce combinators BR_n which reverse the order of their arguments:

$$
\begin{array}{lllll}
BR_n = \lambda_n.- & @- & @- & @ - \cdots - & @ - 0 \\
 & 1 & 2 & 3 & n-1
\end{array}
$$

Now $(m\ BR_n)$ reduces to (in linear representation, application nodes @ suppressed) which has the correct sequence:

$$\lambda 1.0\ a_{33}\ a_{32}\ a_{31}\ a_{23}\ a_{22}\ a_{21}\ a_{13}\ a_{12}\ a_{11}$$

The B_n and BR_n act as concatenators. The sequence of arguments is essential. Mathematicians use $f \circ g$ to indicate that g has to be applied first, and $f \bullet g$ to indicate that f has to be applied first.

The methods of using special lambda expression to represent and manipulate arrays are general and apply also to lists. If datastructures are restricted to two elements we get the "cons" construct, the selectors $\lambda 2.1$ and $\lambda 2.0$ correspond to "car" ("head") and "cdr" ("tail"). Compound selectors correspond to list selectors of the type "caaddar."

As a last point in this section we will demonstrate the use of the lambda calculus as a vehicle to implement the APL type functions of dropping and taking elements from arrays. Although the lambda calculus serves in this context as a very high level machine language, functions become more complicated. One might get the impression that this is due to the use of De Bruijn indices. But not using them would complicate it even more because of the many variable names and the constant worry about confusing them. We already used the

notion of a combinator as an abbreviation for a closed lambda expression to hide unnecessary detail. Combinators are parameterized elements of a set formed according to a common pattern [0]. We introduce the notion of parenthetical combinators, which serve to hide the particular parameter. The value of the parameter is inferred from the input either at run or compile time.

For example, the B, T, and BR combinators are used with the following syntax:

$$B_n = (BN f_{n-1} \ldots f_2\ f_1\ NB)$$

This expression is interpreted as a lambda expression with the appropriate indices filled in which are derived from the number of expressions occurring between "$(BN$" and "$NB)$."

In the next example we use the combinator pair "AT"and "TA"which denote the "apply-to-all" construct [1]. In contrast to J. Backus' version, the function to be applied to all expressions is in our case argument to the construct.

The following expression takes the last two elements (that is, drops the first three elements) from a five element structure: Every element is individually selected and the selectors placed at appropriate positions between "$(AT$"and "$TA)$." The structure enclosed in "$(T$"and "$T)$"corresponding to T_n is argument to the "apply-to-all" construct; the whole expression is abstracted from the elements.

$$\lambda 5.\ (AT(\lambda 5.4)(\lambda 5.3)TA)(T\ 4\ 3\ 2\ 1\ 0\ T) \Longrightarrow \lambda 6.0\ 5\ 4$$

This expression seems involved, but it reduces to $(\lambda 6.0\ 5\ 4)$ which is much simpler and less costly to apply. The next expression takes the first three elements:

$$\lambda 5.(AT(\lambda 5.2)(\lambda 5.1)(\lambda 5.0)TA)(T\ 4\ 3\ 2\ 1\ 0\ T) \Longrightarrow \lambda 6.0\ 3\ 2\ 1$$

These expressions are the base for the implementations of drop-and-take functions which will have an additional parameter to indicate the number of elements dropped or taken. To rotate a five-element structure by two we concatenate both selections from above in reverse order with the combinator B and we obtain:

$$\lambda 5.\ (\lambda 1.B \quad ((AT(\lambda 5.4)\ (\lambda 5.3)\ TA)0)\ ((AT(\lambda 5.2)\ (\lambda 5.1)(\lambda 5.0)\ TA)0))\ (T\ 4\ 3\ 2\ 1\ 0\ T)$$

This expression reduces to $\lambda 6.0\ 3\ 2\ 1\ 5\ 4$. The result is not surprising, but there are more complicated compositions of APL type functions, which after reduction yield a lambda expressions which do not exhibit their operation so clearly. Also, there may be other ways of expressing the drop function. In general, two compound expressions are equivalent, if they reduce to equal normal forms. We can say equal, because the De Bruijn indices yield a unique representation of lambda expressions. An example shows the point. Through experience and intuition one might find the following solution to drop one element from a datastructure:

$$B(\lambda 1.0\ a\ b\ c)K$$

An intermediate step in the reduction sequence is:

$$\lambda 1.(\lambda 1.0\ a\ b\ c)(K\ 0)$$

The next beta-reduction step replaces the innermost variable 0 in the expression by $(K\ 0)$.

$$\lambda 1.K\ 0\ a\ b\ c$$

But $K\ 0\ a$ reduces to 0 and we finally obtain

$$\lambda 1.0\ b\ c$$

The result is again a datastructure and repetition of the $(B \ldots K)$ construct drops a corresponding number of elements. To prove equivalence we reduce

$$\lambda 5.(\lambda 1.(B(B\ 0\ K)K)(T\ 4\ 3\ 2\ 1\ 0\ T)$$

and obtain $\lambda 6.0\ 3\ 2\ 1$. This is the same result as obtained above using the element by element selection for dropping two elements from the structure.

We could now proceed to augment the system by arithmetic. However, adding arithmetic does not give any more insight in the possibilities of the lambda calculus to decompose and compose program- and datastructures.

SEMANTICS

The more difficult aspects of the semantics of the lambda calculus have not been used. Everything is finite and termination problems are not expected.

The use of reduction semantics rather than an evaluation scheme is an important point. The latter would indeed cause a semantics problem. The main reduction rule is beta-reduction. It is not necessary to consider or to refer to an alpha-rule, because there are no identifiers which could confuse variables. However, during a beta-reduction some De Bruijn indices might change in order to maintain the binding structure. Since a lambda expression is uniquely represented if De Bruijn indices are used, there is only one correct answer. Beta-reduction could be done by hand, but the mechanical assistance of an operational lambda calculus system facilitates it. As a matter of fact, all examples shown above have been verified using such a system [3].

To qualify for the task, a system has to be a full and complete, strongly normalizing implementation of the lambda calculus. A functional language implementation will generally not suffice. The system has to handle relative free variables correctly. The necessity for it becomes evident from the following argument.

A nest of DO-loops would appear in a functional language or lambda calculus representation as a nest of recursive functions. The transformations which move code from high frequency areas to low frequency areas leave free variables, which have to be used and implemented as such. The innermost, high frequency areas cannot be artificially closed. This would lead to more beta-reductions and the expected gain in efficiency is lost.

Reduction semantics has to be implemented in such a way that partial application and higher order functions are naturally available in the system. The concept of a function either insisting on the availability of all its arguments, or else generating error messages, is not very useful in this context. Partial application and higher order functions are the norm, not the exception, since input and output are represented as functional datastructures. The system has to be strongly normalizing, because functions (datastructures) have to be brought in normal form as output. When parameters are set to customize certain functions, some simplifying reductions within the function should be possible.

Equivalence of different procedures to achieve the same effect on the same input can be established by reducing both to the equal normal forms. The normal form is at the same time the most efficient, minimal representation of the procedure. Thus, the starting point can be as declarative as necessary to convince the user of its correctness. Also, the process of obtaining the normal form corresponds to a conventional compilation process.

PRAGMATICS

When comparing a bulk of assembly code with a huge lambda expression allegedly accomplishing the same task the question arises: What has been gained here? The answer is easy: The lambda expression is based on a very good mathematical theory. But, this answer barely satisfies. The deeply nested lambda expression, with no functions to discern and bindings as well as unintelligible De Bruijn indices distributed all over the expression, reveals its meaning in no way faster or easier than a bulk of assembly code. And above all, in either case the problem is, how to obtain it in the first place. A higher level programming language is a way for assembly code, without really solving the problem, however. But what is the higher level language with respect to the lambda calculus, which is in itself a higher level language? Syntactic sugaring may raise the level of appearance, but does not necessarily raise the conceptual level.

Some researchers consider combinators to be an alternate system to implement functional systems. In terms of the lambda calculus they correspond to very simple, closed lambda expressions, so simple that there is no need to implement the lambda calculus at all. (There is, of course, the need to compile functional expressions into combinator expressions.) Combinators as abbreviations of simple lambda expressions, however, fill a need to conceptualize general patterns of usage. They hide De Bruijn indices and other detail, one does not want to know about. The parenthetical combinators hide explicit parameters, which are inferred. They are more than an abbreviation, they represent a process (not necessarily part of the lambda calculus!) to create a combinator of a certain class or type.

There is an obvious need to name lambda expressions once they are created and their internal details are not of interest anymore. But don't we admit assignment statements again through the backdoor? This would certainly violate first principles. First of all, we do not

fill a box variable, we associate a name to an expression so they become equivalent. (By the way, names are expanded only in head positions, there is no point to do it everywhere.) Secondly, first principles may be bent to accommodate pragmatic needs. In this case, the pragmatic need consists of defining and redefining names. An easy way to accomplish this is to make environment entries from definitions. These associations remain valid and accessible as long as the environment is not cut back. Redefinitions are shadowed, but not overwritten. This method can be characterized as uncompleted (may be over the life of the system) beta-reductions.

The interactive capabilities of modern computers offer another pragmatic advantage. The conventional, historic method first creates a large program structure, the source code, then compiles it into object code, and then "runs" it. This method is due to hardware limitations which have been overcome and is not adequate anymore. The best way to construct large, nested structures is interactive and piecemeal.

The system can guide the user, indicate at each position or state what are permitted moves, what is correct input, what are allowed structure changes. Trivial errors are prohibited at the spot. The problem of parsing can be made to disappear if only atomic tokens are permitted as input while the expression structure is system driven.

There should be means to traverse the structure once it is created to make changes where necessary. To adhere to first principles in this pragmatic context means to maintain full "recursiveness." With other words, what is available and possible at the top of the expression should be possible at every subexpression. Desirable meta-operations on expressions are:

1. Perform a predetermined number of reductions on it. (√)
2. Display it according to a selected format. (√)
3. Transform it into a combinator expression (abstraction). (√)
4. Replace it by another expression. (√)
5. Name it. (√)
6. Save it on secondary storage. ( )
7. Expand it. (√)

All these amenities do not belong to the lambda calculus, but they make it a working system. The checkmarks indicate availability in our experimental system.

PERFORMANCE

If the lambda calculus is to be used for real computing the question must be asked, does it provide real performance? The key to the possibility to reach satisfactory performance is the particular representation which can be chosen if one uses De Bruijn indices. The sequence of application nodes in the representation of vectors has to be implemented in consecutive memory locations. This is as good as any conventional method. If the sequence of application nodes becomes arguments of a function, they are now in consecutive environment locations, too. The De Bruijn index is the same as the array index, and the same as the offset when addressing the memory with the current environment as base address. Thus, the basics of hardware use are essential the same as in the conventional method and no penalty in performance has to be paid.

The issue is not as clear with respect the nests of recursive functions into which nested DO-loops get transformed. Although recursion benefits from the implementation choices, too, their automatic transformation into loops is difficult.

Some special non-lambda calculus measures are needed to permit environment changes. The environment is a shared, read-only tree structure containing arrays. Adhering to first principles permits overwriting environment entries by equivalent values only. To create a new array with only one element changed, access to the array in the environment has to be filtered for a selected index by a new environment. All other accesses are referred to the old array. This method allows the existence of different arrays, which share common elements as much as possible.

CONCLUSION

This paper investigates the suitability of the lambda calculus as a representation for arrays and the functions to manipulate them. The results show that only a full and complete, strongly normalizing implementation of the lambda calculus will suffice. Reduction semantics, De Bruijn indices, performance oriented representation and

implementation techniques based on proven hardware concepts, and special interactive tools have to be combined. Much further work is necessary to realize the potential power of the lambda calculus as a machine language for actual use. Particular attention has to be given to the scaling-up properties of the methods described.

BIBLIOGRAPHY

[1] Abdali S.K., "An Abstraction Algorithm for Combinatory Logic." *The Journal of Symbolic Logic*, Vol. 41, Number 1, March 1976, pp. 222-224.

[2] Backus, J. "Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs." CACM, V21, N8, pp613-641, (1978).

[3] Berkling, K.J., "Headorder Reduction: A Graph Reduction Scheme for the Operational Lambda Calculus," *Proceedings of the Los Alamos Graph Reduction Workshop*, Springer Lecture Notes in Computer Science, Vol 279, (1986).

[4] Burge, W.H. "Recursive Programming Techniques." Addison-Wesley, Reading Massachusetts, (1975).

[5] Mullin, L.M.R. "A Mathematics of Arrays." Ph.D. Dissertation, School of CIS, Syracuse University, (1988).