AN EXTENSION OF LAMBDA-CALCULUS FOR FUNCTIONAL PROGRAMMING*

GYORGY REVESZ

▷ An implementation oriented modification of lambda-calculus is presented together with some additional conversion rules for list manipulations. The resulting set of axioms is very simple and provides for a theoretical foundation of the semantics of functional programming. At the same time it can be used directly for the implementation of a powerful graph-reduction algorithm. ◁

INTRODUCTION

Here we assume familiarity with the basics of lambda-calculus and with the main ideas of functional programming as promulgated by Backus. [2]

The efforts to implement functional languages have come up with various ideas and approaches. There are basically two different groups of efforts:

The first group is trying to solve the problem by using lambda-calculus reduction or combinatory reduction for function evaluation. (See, for instance, [4] and [7].)

The second group is trying to use more specialized techniques tailored to the given functional language. (See, for instance, [5].)

In both cases the function to be evaluated is usually represented as a graph (sometimes a tree) and thus, the function evaluation process can be seen as a graph manipulation. Each step of the function evaluation will be reflected by a corresponding change in the graph and the entire graph should eventually collapse to a single node if the function evaluates to a constant.

Now, the question is how best to represent a function via a graph and then how to make each step in a reasonably efficient manner. A major problem in this regard is

*This research has been partially supported by NSF Grant No. MCS-8304819.

Address correspondence to Gyorgy Revesz, Department of Computer Science, Tulane University, New Orleans, LA 70118.

how to deal with common subexpressions. Sharing is desirable in order to avoid a duplication of the effort spent on the evaluation of common subexpressions. But sharing of subexpressions makes the graph representation and its manipulation more complex. A good discussion of various methods can be found in [1].

In the present paper we shall describe a new graph reduction technique which seems to combine the advantages of both the lambda-calculus and the combinators. Indeed, it avoids the translation from lambda expressions to combinators so it does not have to deal with bracket abstraction [8]. (Such a translation is also burdensome for program debugging because combinatory terms are hardly readable by human beings.) At the same time, our conversion rules are simpler than the usual β-rule for the standard lambda-calculus. This way, the substitution operation is performed in several steps each producing a valid lambda expression for an intermediate result. Similar axioms have been studied in a recent paper [6].

Furthermore, we have to add only three specific axioms for lists and we get a quite powerful functional language. This language will be described in the next section, together with the graph representation of the expressions in the language.

A SIMPLE REDUCTION LANGUAGE

Our language is a direct extension of the lambda notation to include arbitrarily nested lists as expressions. The only deviation from the usual lambda-notation is the fact that we always parenthesize the operator of an application rather than its operand. So we would write $(f)x$ instead of $f(x)$, etc. The syntax of our Simple Reduction Language (SRL for short) is the following:

$$\begin{aligned}
\langle\text{expression}\rangle ::= {} & \langle\text{variable}\rangle|\langle\text{constant}\rangle|\langle\text{list}\rangle| \\
& (\langle\text{expression}\rangle)\langle\text{expression}\rangle| \\
& \lambda\langle\text{variable}\rangle.\langle\text{expression}\rangle \\
\langle\text{list}\rangle ::= {} & [\,]|[\langle\text{expression}\rangle\langle\text{list-tail}\rangle \\
\langle\text{list-tail}\rangle ::= {} &]|,\langle\text{expression}\rangle\langle\text{list-tail}\rangle \\
\langle\text{constant}\rangle ::= {} & \langle\text{integer}\rangle|+|-|*|/|\hat{\ }|\tilde{\ }|\&|?
\end{aligned}$$

Hence we can form functional expressions like

$\lambda x.\lambda y.(x)y$
$\lambda x.\lambda y.\lambda z.((x)z)(y)z$
$(\lambda x.[(x)y,(y)x])M$
$(\lambda q.[p,q,r])Q$
$(\lambda x.(\lambda y.[x,y])a)[b,c,d]$

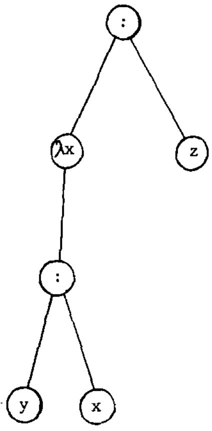
and so on. With this, of course, we expect that the third example in the above list will reduce to $[(M)y,(y)M]$, the fourth to $[p,Q,r]$, and the fifth to $[[b,c,d],a]$. We have included three constant symbols for list manipulating functions. These are denoted by $\hat{\ }$, $\tilde{\ }$, and &, and represent the operations of head, tail, and cons, respectively. Hence

$(\hat{\ })[a,b,c]$ yields a
$(\tilde{\ })[a,b,c]$ yields $[b,c]$
$((\&)a)[x,y,z]$ yields $[a,x,y,z]$.

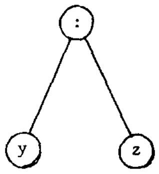
Similarly, we have included the arithmetic operations as constant symbols. Variables are character strings starting with a letter followed by letters or digits as usual.

The question mark represents the fixed point combinator such that $(?)E$ reduces to $(E)(?)E$ for any expression E.

The internal representation of an expression is a directed acyclic graph reflecting the syntactic structure of the expression. The nodes of the graph correspond to the syntactic operations making up the expression. There are four basic types of nodes: variable, constant, abstraction, and application. An application node (denoted by: in the graph) has two children, an abstraction node has one, and a variable or constant node has none. The abstraction node accommodates the bound variable in itself. For instance, the expression $(\lambda x.(y)x)z$ is represented by the following graph:

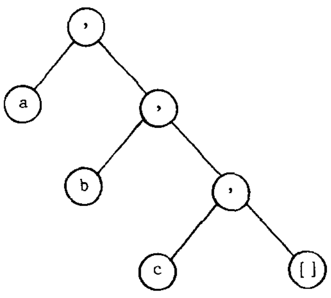


This would be reduced to



in one step according to the standard β-rule. But in our system, as we shall see later, this reduction takes place in three steps. The gain of this extra effort is the total elimination of the substitution operation which is far too complex anyway for being considered an elementary operation.

For the representation of lists we need only two more node types. One corresponds to the cons operation, the other is the nil node corresponding to an empty list. So the list $[a, b, c]$ is represented by the graph.



The elements of a list can, of course, be arbitrary expressions involving other lists, etc. A list-forming operation, *list*, can be defined easily as follows:

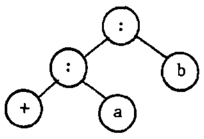
let list $= \lambda x.[x]$

which makes the application

(list)A

reducible to $[A]$ for any expression A.

All functions are curried in our notation, that is the addition of a and b is represented by $((+)a)b$ and has the graph



If someone likes a pair better for an argument, he can define the function *plus* this way

let plus $= \lambda x.((+)(\hat{\ })x)(\hat{\ })(\tilde{\ })x$

which gives

(plus)$[a, b] = ((+)a)b$

as a result. Conversely, one can define the pairing function like this

let pair $= \lambda x.\lambda y.[x, y]$

which, when applied to two arguments in a row, gives the requested pair namely,

$((\text{pair})A)B = [A, B]$

for any A and B. This definition is obviously simpler than the following:

let pair $= \lambda x.\lambda y.((\&)x)(\text{list})y$

but both are correct.

For more complex operations on lists we need the built-in function *nil* to check, if a list is empty. Therefore, we have

$$(\text{nil})A = \begin{cases} \lambda u.\lambda v.u, \text{ if } A = [\,] \\ \lambda u.\lambda v.v, \text{ if } A \text{ is a nonempty list} \\ \text{undefined otherwise} \end{cases}$$

The *if p then q else r* construct can be defined in our lambda notation as a three-argument function as follows:

let cond $= \lambda p.\lambda q.\lambda r.((p)q)r$

provided that the truth values are represented by the appropriate lambda expressions (as with the nil above).

Hence, the *append* function, which puts together two lists, can be defined recursively as follows:

let append

$$= \lambda x.\lambda y.(((\text{cond})(\text{nil})x)y)((\&)(\hat{\ })x)((\text{append})(\tilde{\ })x)y$$

Here the recursion can be resolved with the aid of the fixed-point combinator.

let append

$$=(?)\lambda f.\lambda x.\lambda y.(((\text{cond})(\text{nil})x)y)((\&)(\hat{\ })x)((f)(\tilde{\ })x)y$$

For better readability, redundant parentheses can be omitted if we assume that application associates to the right. But then, as a trade-off, we need an infix operator like ':' to prevent variables from running together. (Blank as an operator is not a good idea.) In this case, the above definition would read

let append

$$=?\!:\lambda f.\lambda x.\lambda y.((\text{cond}:\text{nil}:x):y):(\&:\hat{\ }:x):(f:\tilde{\ }:x):y$$

Nevertheless, this syntactic convenience will not alter the fact that here we have curried functions whose application to an argument returns usually another function.

Composition of functions can be defined as in lambda-calculus:

let comp $=\lambda x.\lambda y.\lambda z.(x)(y)z$

Indeed, having the full power of lambda-calculus made available we can easily define other functional forms producing new functions from given ones.

A QUASIOPERATIONAL SEMANTICS FOR SRL

In the previous section we have already informally discussed certain semantic features of the language.

For a complete formal treatment of the semantics we offer a set of axioms which allow for *meaning-preserving* transformations, called reductions, on the expressions. These transformations are related to the process of the evaluation of the expressions and thus, they can be seen as an *operational semantics* of the language. Due to the Church-Rosser property, we do not have to be very specific about the exact strategy of the evaluation procedure, we only have to make sure that it succeeds whenever the expression does have a value (i.e., there exists at least one sequence of elementary transformations leading to an irreducible form).

Before describing the axioms we have to define the set of variables occurring free in an expression.

Definition. The set of variables occurring free in some expression E, denoted by $\varphi(E)$, is defined recursively as follows:

(0) $\varphi(c)=\{\ \}$ for every constant c.
(1) $\varphi(x)=\{x\}$ for any variable x,
(2) $\varphi(\lambda x.E)=\varphi(E)-\{x\}$ for any expression E and variable x,
(3) $\varphi((E_1)E_2)=\varphi(E_1)\cup\varphi(E_2)$ for any two expressions E_1, E_2.
(4) $\varphi([\,])=\{\ \}$
(5) $\varphi([E_1,\ldots,E_n])=\bigcup_{i=1}^{n}\varphi(E_i)$ for any sequence of expressions $E_1,\ldots,E_n$.

The reduction axioms will be arranged in three groups. The first group consists of the renaming rules, referred to as α-rules. The notation $\{z/x\}E$, where x and z are arbitrary variables and E is an arbitrary expression, represents the operation of renaming, i.e., replacing the free occurrences of x by z in E. This is defined inductively by the following α-rules:

(α1) $\{z/x\}E \to z$ if $E = x$
(α2) $\{z/x\}E \to E$ if $x \notin \varphi(E)$
(α3) $\{z/x\}\lambda y.E \to \lambda y.\{z/x\}E$ if $x \neq y \neq z$ and z is not bound in E
(α4) $\{z/x\}(E_1)E_2 \to (\{z/x\}E_1)\{z/x\}E_2$
(α5) $\{z/x\}[E_1,\ldots,E_n] \to [\{z/x\}E_1,\ldots,\{z/x\}E_n]$

The next group corresponds to the β-rule of the standard lambda-calculus. But we have here four rules and no substitution operation as such.

(β1) $(\lambda x.x)E \to E$
(β2) $(\lambda x.E_1)E_2 \to E_1$ if $x \notin \varphi(E_1)$
(β3) $(\lambda x.\lambda y.E_1)E_2 \to \lambda z.(\lambda x.\{z/y\}E_1)E_2$ for any $z \notin \{x\} \cup \varphi(E_1) \cup \varphi(E_2)$ that is not bound in E_1
(β4) $(\lambda x.(E_1)E_2)E_3 \to ((\lambda x.E_1)E_3)(\lambda x.E_2)E_3$

The last group contains two specific axioms dealing with lists.

(γ1) $([E_1,\ldots,E_n])F \to [(E_1)F,\ldots,(E_n)F]$
(γ2) $\lambda x.[E_1,\ldots,E_n] \to [\lambda x.E_1,\ldots,\lambda x.E_n]$

Interestingly enough, if the last two axioms are added to the α-rules and β-rules above, we can deduce any of the valid properties of lists. For instance, the algebraic law of composition

$$[f,g]\circ h = [f\circ h, g\circ h],$$

treated as an axiom by Backus, can be deduced as follows. By the definition of composition we have

$$[f,g]\circ h = ((\lambda x.\lambda y.\lambda z.(x)(y)z)[f,g])h$$

The right-hand side β-reduces (in several steps) to $\lambda z.([f,g])(h)z$. Then by using the γ-rules we get

$$\lambda z.([f,g])(h)z \underset{\gamma_1}{\to} \lambda z.[(f)(h)z,(g)(h)z] \underset{\gamma_2}{\to}$$

$$[\lambda z.(f)(h)z, \lambda z.(g)(h)z] = [f\circ h, g\circ h]$$

For algebraic laws involving the $p \to f; g$ conditional form by Backus, it is necessary to note that the equivalent lambda-expression has this form

$$\lambda p.\lambda f.\lambda g.\lambda x.((p:x)f:x)g:x$$

which takes four arguments rather than just three.

The lack of the substitution operation is not a problem for us, because its power is simply shared by the given set of axioms. In fact, this is an advantage for the implementation, because these elementary steps are easier to deal with.

From a theoretical point of view, it is worth mentioning that our axiomatic method has certain interesting properties.

First of all, we did not have to give up the intuitive simplicity of the lambda-notation in order to eliminate the complexity of the substitution from our axiom system. This is an obvious advantage over the use of combinators. This also means that we do not have to translate our functional expressions into combinators which saves us the trouble of the so called bracket abstraction [8].

Concerning lists, it is well known that they can be represented in pure lambda-calculus and thus, at least theoretically, no extra notation and no additional rules are necessary to deal with them. But the same is true for the integers, boolean values, etc. So it is a question of practicality where one would like to draw the line. In our opinion, list structures are useful and they are worth the effort to treat them directly as they are. This is especially true for our axiom system where it only takes two extra rules (besides the $\alpha 5$-rule) to deal with them. For a combinatory treatment of lists, see also [3].

Conventional programming languages like Pascal can be translated relatively easily into SRL, if the program is well structured. There is no big difference between the lambda-notation and the usual representation of formal parameters. If, for instance, we have an expression E to compute the value of a two-argument function $f(x, y)$, then the equation $f(x, y) = E$ is not much different from $f = \lambda x.\lambda y.E$. The translation of some other language constructs is less obvious but manageable. Hence, an interesting justification of structured programming can be obtained based on the simplicity of the translation of such programs into a functional language.

A NEW GRAPH-REDUCTION TECHNIQUE

As we have mentioned in the previous section we represent any SRL expression internally as a directed acyclic graph. Initially this graph has a tree form which is essentially the parse tree of the expression and is built by the parser during the input phase.

Thereafter, the evaluation will reduce the graph into its simplest form, possibly a single node holding a constant value. If the reduced form is not a single node then we have a functional expression as a result which is quite normal for function valued expressions. (We can have all kinds of partial evaluations or function valued functions with no difficulty at all.)

The current version of our implementation has the following node types:

[]	list terminator (empty list)
:	application
λx	abstraction
var	variable
@	indirection
?	fixed point combinator
,	internal list constructor
^	head operator
~	tail operator
#	numeric value
&	list constructor as a binary operator
$\{z/x\}$	renaming operator

The arithmetic operators $+, -, *, /$ are implemented as special variables just like some other built-in functions.

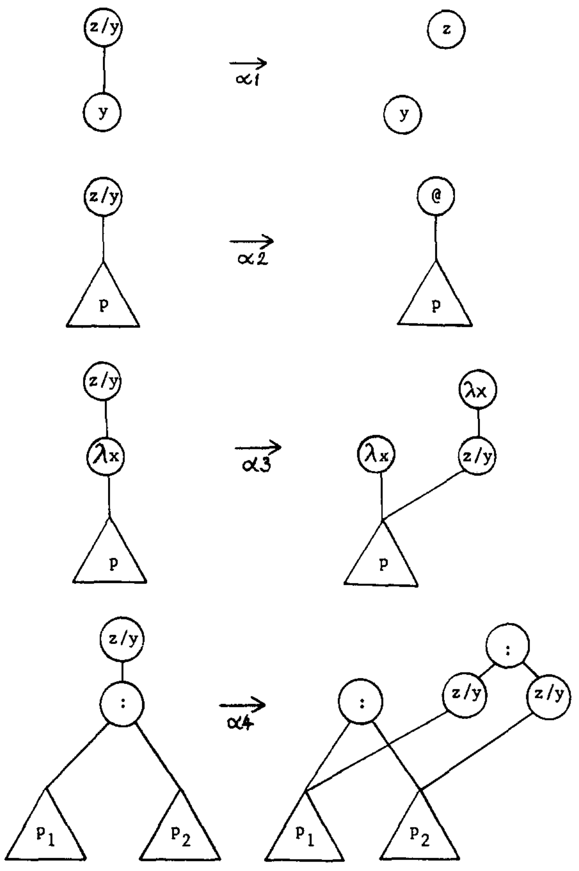


FIGURE 1. Renaming rules.

One of the interesting features of our implementation is our handling of the renaming operation. For each renaming request we create a new node holding the renaming prefix. This node will act upon its dependent expression according to the α-rules as shown in Figure 1.

The initial graph as constructed by the parser does not contain indirection or renaming nodes. These are created only when necessary during the reduction process.

The reduction procedure follows the so-called normal order (also called leftmost, or outside-in) reduction strategy starting from the root. Whenever it locates a β-redex, i.e., a subexpression of the form $(\lambda x.P)Q$, then it will choose one of the β-rules to apply to that redex. It always tries to apply $\beta 2$ first and if this fails then one of the other three possibilities must occur unless the top-node of the subgraph P is an internal list constructor.

The application of a β-rule will modify the graph as shown in Figure 2. The only node that is changed in its own place is the top-node of the redex. Otherwise we have to create new nodes in order to make the sharing of subexpressions possible.

A new renaming node can occur only as a result of the application of a $\beta 3$-rule. But it will also be eliminated before the next β-reduction occurs.

For the implementation of the $\alpha 5$-rule and the two γ-rules we make one more step in their internal decomposition. Namely, the internal form of the $\alpha 5$-rule will be

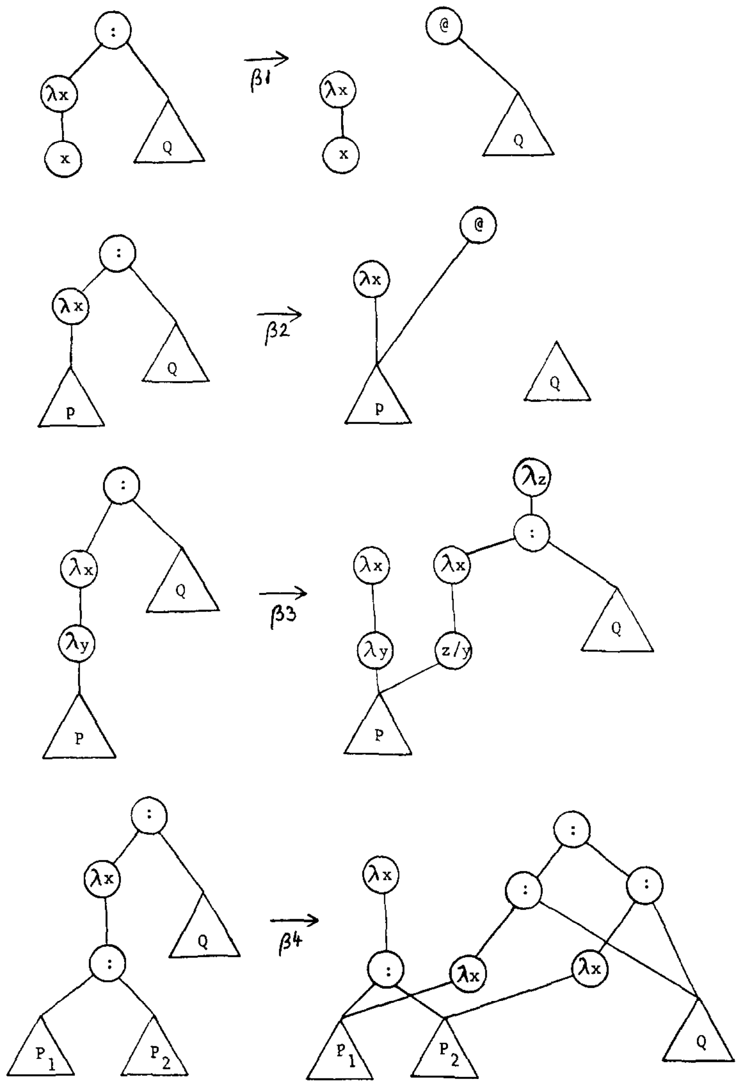
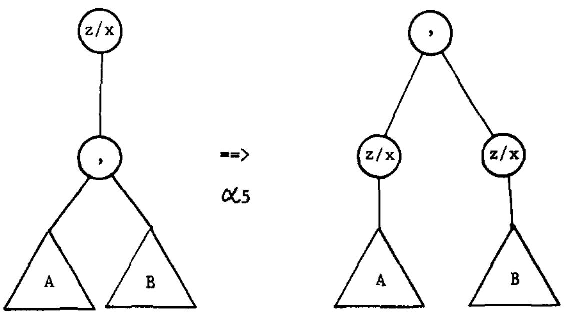
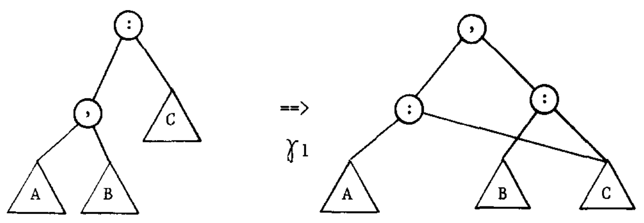


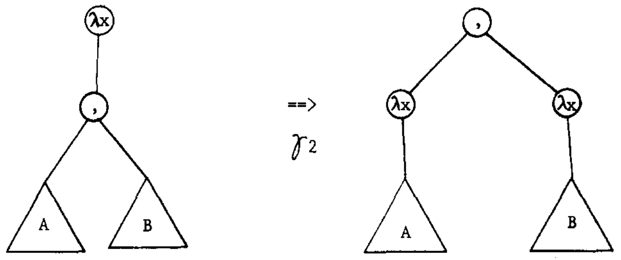
FIGURE 2. Reduction rules.

the following:

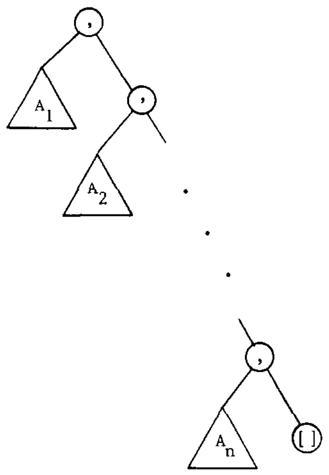


This means that the renaming prefix will simply be distributed between the first member and the rest of a list. Similarly the two gamma rules will be decomposed as follows:





Note, that the internal representation of an arbitrary list has the form



Now in order for the above rules to work, we have to add the following internal gamma rules

$([\,])E \to [\,]$ and $\lambda x.[\,] \to [\,]$

which will take care of the correct termination of the above decompositions.

The entire system is very simple and easy to implement. But it is quite efficient, too. To support this last claim we may add that a single stack is used to control the process. We have no funarg problem with environments and closures, etc. This is so because we strictly follow the normal order in a fully lazy manner. At the same time, the expression-sharing mechanism gives us automatically all its possible benefits.

Special thanks is due to Peter Revesz for his contribution to the programming of the graph-reducer.

REFERENCES

1. Arvind, Kathail, V. and Pingali, K., Sharing of Computation in Functional Language Implementations, *Proc. Internat. Workshop on High-Level Computer Architecture*, May 21–25, 1984, Los Angeles, pp. 5.1–5.12.
2. Backus, J., The Algebra of Functional Programs: Function Level Reasoning, Linear Equations, and Extended Definitions, in: J. Diaz (ed.), *Formalization of Programming Concepts*, Lecture Notes in Comp. Sci., Vol. 107, Springer, New York, 1981, pp. 1–43.
3. Bohm, C., An Abstract Approach to (hereditary) Finite Sequences of Combinators, in: J. P. Seldin and J. R. Hindley (eds.), *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, Academic Press, New York, 1980, pp. 231–242.
4. Henderson, P. and Morris, J. M., A Lazy Evaluator, *Conf. Record of the Third ACM Symposium on Principles of Program Languages*, 1976, pp. 95–103.
5. Magó, G. and Middleton, D., The FFP Machine—A Progress Report, *Proc. of the Internat. Workshop on High-Level Computer Architecture*, May 21–25, 1984, Los Angeles, pp. 5.13–5.25.
6. Revesz, G., Axioms for the Theory of Lambda-Conversion, *SIAM J. Computing*, to appear.
7. Turner, D. A., A New Implementation Technique for Applicative Languages, *Software, Practice and Experience* 9:31–44 (1979).
8. Turner, D. A., Another Algorithm for Bracket Abstraction, *J. Symbolic Logic* 44:267–270 (1979).