Algebraic Transformation Techniques for Functional Languages

P. G. Harrison and H. Khoshnevisan

Department of Computing, Imperial College of Science and Technology, 180 Queen's Gate, London SW7 2BZ

The often conflicting needs to make software both efficient and correct have made a large contribution to the present so-called software crisis, and transformation-based support environments for functional languages offer a major step towards solving this conflict in requirements. In such an environment programs are initially developed by concentrating only on the understandability, correctness, clarity, reliability and maintenance aspects. This initial specification is then transformed through a series of meaning-preserving transformations to achieve an efficient implementation. We argue that the abstraction level of the majority of commonly used functional languages is too low-level for the results of the analysis to be automatically implemented or very generally applicable. However, by compiling such programs into a higher-level, more structured, variable-free representation, we show how the analysis can achieve more powerful, mechanised, and generally applicable transformations. This is due in part to the elimination of the concern for the domain of objects, since function definitions are now expressed purely in terms of functions and so become simpler.

Many recursive functions are linear in the sense that the number of recursive function calls they generate is bounded by a number which is proportional to the magnitude of their argument. The performance of functional languages can therefore be improved by a more efficient implementation of linear functions, and we derive equivalent imperative language loops for a large class of linear recursive functions. Moreover, such linear functions can be detected automatically in the parsing phase of a compiler and their loop implementations generated. Other recursive functions are non-linear, generating a number of function calls that grows in a non-linear manner with respect to the magnitude of the arguments to which they are applied, for example quadratically or exponentially. Although non-linear functions tend to be fewer, their run-time performance tends to be relatively much poorer, and so their efficient implementation too is of considerable importance to functional languages. We illustrate how certain non-linear function definitions can be transformed into linear ones, and how they can therefore subsequently be implemented as loops. An alternative, more automatic approach for the treatment of non-linear functions uses memo-functions, which are functions that 'remember' all the arguments to which they have been applied, together with the corresponding results computed from them. We define a class of non-linear functions for which memoisation linearises the time-cost of calls of a non-linear function to itself whilst executing in bounded space. The technique for generating such memo-functions is widely applicable, easily mechanised and achieves improvements in efficiency that are comparable with existing program transformation schemes. Furthermore, the sizes of the tables for these memo-functions are guaranteed not to exceed a compile-time constant found by a simple static analysis of the definition of the non-linear function.

Received May 1987

1. INTRODUCTION

The difficulties encountered in the production of reliable and maintainable software are largely due to the conflicting needs to make software both efficient and correct. When using conventional high-level languages the programmer is forced to consider efficiency aspects or use of machine resources which are totally unrelated to the correctness of his program. The transformation approach is based on separating the concerns of correctness and efficiency. Programs are initially developed by concentrating only on the understandability, correctness, clarity, reliability and maintenance aspects, and this initial specification is then modified through a series of meaning-preserving transformations into an efficient implementation. The present paper considers the transformation of functional languages, such as Hope, KRC, and Miranda, for which the syntax is essentially a sugared form of the lambda calculus.

The work of Burstall and Darlington, which presented a set of correctness-preserving transformation operations, laid down the pattern for later work in this area, but their approach requires a considerable amount of ingenuity in choosing some of the basic transformation steps. Another drawback is that formally based transformation systems based on this approach tend to be too low-level, in the sense that the power of their primitive operations is very restricted. Thus transformations expressed solely at this level are typically very long, complex, difficult to comprehend and so normally have to be organised by the user. Any program-transformation technology relies on an analysis of the representation of programs. These representations have typically been at the abstraction level of the language in which the program was written, or at lower levels succeeding one or more compilation phases. We argue that the abstraction level of the majority of commonly used functional languages is too low for the results of the analysis to be generally applicable or suitable for mechanisation, and make the case for a higher-level, variable-free representation.

In the lambda-calculus-based functional languages a function is defined by stating the result values of applications of that function in terms of the particular objects to which the function is applied. In this framework a new program, i.e. function, is built by applying given functions to variables or objects which are in general the results of other such applications, until the result object is built. The new program is then obtained by abstracting the object variables as formal parameters. For example, the following Hope definition of a function which
computes a list of factorials, defined in terms of a function for factorial which is assumed to exist already, is:

\[
\text{factlist}(x) = \begin{cases} 0 & \text{if } x = 0, \\ \text{nil} & \text{else} \\ \text{factorial}(x) \cdot \text{factlist}(x - 1) & \end{cases}
\]

The problem is that the principal tool for building programs in these languages is lambda abstraction, which is not a combining form for programs, but merely combines a variable and an expression which contains free variables to form a program. In other words it combines two entities, neither of which is a program, to form a program. Thus lambda abstraction does not provide an algebra of programs on which a theory of programming could be based.

One of the main reasons behind the failure to develop a formal theory for imperative programs is that they lack referential transparency. Although this situation has improved in the area of functional programming, we are still a long way from having general theorems which are immediately useful to a programmer for transforming his program or proving it correct. As a result of the explicit role of objects in the lambda-calculus-based functional languages, the transformations which have been developed are of a rather specific nature, for example, the work of Burstall and Darlington, in which a set of rules is presented for transforming recursively defined functions at the object level. Many examples and strategies may be generated, but as in the structured programming approach, there are only a few generally applicable equivalences. This kind of approach can be very powerful in that many equivalences between programs can be proved, and thus it may be helpful when a new problem is encountered, for example if one were the first person to have encountered the need for recursion removal. But it is very difficult to make the results obtained more general and re-usable. This is because a lot of ingenuity is often needed in choosing the right steps—"Eureka" steps—in the transformation process; the "definitions", "where-abstractions" and "folds" tend to be particularly obscure. For example, it is necessary to select for folding not only the appropriate functions but also the right formal parameters.

The derivation of generally applicable theorems about programs is greatly assisted if programs are built only by the application of certain operations to existing programs, and if these operations or combining forms, also called functional or program forming operations (PFOs), have attractive algebraic properties. Indeed, this is a fundamental characteristic of many mathematical systems, and in Section 2 we outline our function-level approach, based on Backus's "functional algebra".

Many recursive functions are linear in the sense that the number of recursive function calls they generate is bounded by a number which is proportional to the magnitude of their argument. The performance of functional languages can therefore be improved by a more efficient implementation of linear functions. Section 3 derives equivalent imperative language loops (equivalently tail-recursion in a source-to-source transformation) for a large class of linear-recursive functions which includes the tail-recursive functions as a small subset. Moreover, such linear functions can be detected automatically in the parsing phase of a compiler and their loop implementation generated. Other recursive functions are non-linear, generating a number of function calls that grows in a non-linear manner with respect to the magnitude of the arguments to which they are applied, for example quadratically or exponentially. Although non-linear functions tend to be slower, their run-time performance tends to be relatively much poorer, and so their efficient implementation too is of considerable importance to functional languages. Section 3 also illustrates how certain non-linear function definitions can be transformed into linear ones, and can therefore subsequently be implemented as loops.

Section 4 discusses memoisation, an alternative, more automatic approach for the treatment of non-linear functions that uses memo-functions. These are functions that remember all the arguments to which they have been applied, together with the corresponding results computed from them, and a variant of memo-functions called self-garbage-collecting memo-functions is presented which can be used to linearise the time-cost calls of a non-linear function to itself whilst executing in bounded space. The technique for generating such memo-functions is easily mechanised and achieves improvements in efficiency that are comparable with existing program-transformation schemes. In addition to releasing the storage taken up by the memo-tables after the useful lifetime of the memo-function, self-garbage-collecting memo-functions garbage-collect (or re-use) entries when they are guaranteed no longer to be useful. In this way they ensure that the size of the table is kept to a minimum. This is achieved at run time by a synthesised function (a 'table-manager' function) which, upon inserting new results, deletes (or re-uses) obsolete entries, and for a certain class of non-linear functions we show that the memo-table sizes are bounded by constant values. A summary and conclusions of the paper are laid out in Section 5.

2. THE FUNCTIONAL-LEVEL APPROACH

The functional-level view of programs is one in which programs are constructed by applying operations to existing programs, where these operations (combining forms or PFOs) have strong algebraic properties as well as being constructs with good expressive power. In other words, the fundamental goal of the function-level view is to emphasise the mathematical structure of programs rather than the textual one. What is meant by good expressive power is that given a program forming operation \( P \), if \( f \) and \( g \) are existing programs with known behaviour then one should know in a simple and direct way how the new program \( P(f, g) \) behaves. PFOs should have a set of algebraic laws that relates them to one another by equating an expression in which one PFO is the main connective with another expression in which a different PFO is the main connective. Programs that are built using the PFOs have a hierarchical structure, as each sub-program itself is built using PFOs, and this gives programs a tree structure that contains program-forming operations at the nodes and primitive functions at the leaves. The advantages and importance of structure have long been recognised in conventional programs, but more importantly for the present work, this simple structure facilitates many of the mechanical transformation techniques considered in later sections.

The function-level approach brings a mathematical viewpoint to programming in much the same way as the
development of abstract data types has to the subject of types, or algebra has for arithmetic. It achieves this by emphasizing the operations on programs and their algebraic structure as opposed to the representation of the programs themselves. With properties derived from the function-level view, it is possible to develop a technology for reasoning about and transforming programs in which general theorems can be established for significant classes of functions and be applied easily in practice.

2.1 A function-level formalism

We adopt most of the notation and primitives of the FP system of Backus. Although FP function definitions are free of objects, we must consider objects, since they are the results of applications of programs. The set of objects contains the undefined object \( \bot \) ("bottom"), a set of atoms, the empty sequence \(< >\), and the sequence \(< x_1, \ldots, x_n >\) of non-bottom objects \( x_i \). Typically the set of objects include numbers, symbols or identifiers, and the truth values \( T \) and \( F \). An object expression is either an object, a sequence of object expressions or an application \( F/E \), where \( F \) is a function expression and \( E \) is an object expression.

A function expression is either

1. a primitive function or
2. \( G(F_1, \ldots, F_n) \), where \( G \) is one of a fixed set of primitive combining forms (e.g., composition, construction, condition defined in the next section), the \( F_i \) are function expressions, and \( n \) is the arity of \( G \), or
3. \( x \), for some object \( x \), where \( x \) is a constant function, i.e., a function that is everywhere \( x \) except at \( \bot \) where it is also \( \bot \), or
4. a user-defined function, for which a unique definition exists.

A function definition has the form

\[
\text{def } f = E
\]

where \( f \) is a function symbol (not previously used) and \( E \) is a function expression that may involve \( f \). Williams shows in Ref. 15 that the strict, bottom-up semantics of FP constitutes a safe computation rule to compute the least fixed point of this recursion equation.

All FP functions are strict (i.e., for all functions \( f, f; \bot = \bot \)) and map a single object into a single object. We write \( f \cdot x \) for the result of the application of the function \( f \) to the object \( x \).

In the following section examples of primitive FP functions and combining forms are given. They are defined informally by presenting an object-level description of them. However, their formal semantics described by function-level equations can be found in Backus.

<table>
<thead>
<tr>
<th>Table 1. Definition of notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T )</td>
</tr>
<tr>
<td>( F )</td>
</tr>
<tr>
<td>( \bot )</td>
</tr>
<tr>
<td>( : )</td>
</tr>
<tr>
<td>(+, -,*)</td>
</tr>
<tr>
<td>( id )</td>
</tr>
<tr>
<td>( null )</td>
</tr>
<tr>
<td>( eq )</td>
</tr>
<tr>
<td>( hd, tl )</td>
</tr>
<tr>
<td>( al, ar )</td>
</tr>
<tr>
<td>( cons )</td>
</tr>
<tr>
<td>( consr )</td>
</tr>
<tr>
<td>( i )</td>
</tr>
<tr>
<td>( lei )</td>
</tr>
<tr>
<td>( subi )</td>
</tr>
<tr>
<td>( o )</td>
</tr>
<tr>
<td>( a )</td>
</tr>
<tr>
<td>( \bot )</td>
</tr>
<tr>
<td>( q )</td>
</tr>
<tr>
<td>( r )</td>
</tr>
<tr>
<td>( ID )</td>
</tr>
</tbody>
</table>

2.2 Notation

In Table 1 we give a summary of the notation of some FP primitives and PFOs which is used throughout the paper together with other abbreviations and symbols used. The meaning of each primitive function is given by specifying the result of its application to various kinds of objects. If they are applied to any other kind of object not mentioned in their meaning, the result of the application will be \( \bot \). For example, an attempt to subtract the integer 1 from

the character 'a' will yield \( \bot \) since 'a' is not of the correct type for the primitive sub1. Note that the three PFOs composition, construction and conditional, are sufficient to represent any first-order function definition in our formalism.

2.3 Axioms of the FP algebra and function-level equations

The functional algebra of FP is based upon a set of axioms which derive from the properties of the primitive functions and functionals (combining forms). Each primitive will induce a number of axioms corresponding to its semantics, giving the appearance of the axioms associated with an abstract data type. The axioms in Table 2 are presented as equations in variable-free form, giving the functional algebra complete independence from the object domain, and so any set which does not give rise to a contradiction would define an algebra. However, to be useful, when the two sides of an axiom equation are applied to the same arbitrary object, the resulting equation must be known to hold at the object

THE COMPUTER JOURNAL, VOL. 31, NO. 3, 1988 231
level. For example, an axiom corresponding to the head selector function for sequences might be \( hdo \ cons \ f, g = f \), for functions \( f, g \). When applied to an object, \( x \), this yields the equation \( hdo:cons: x, g, x = f, x \) which we know to be true. The same argument applies to all of the axioms we list in Table 2. Any further primitive data types, such as trees, could also be assumed to exist, each contributing to its own set of axioms, but we just give the sequence axioms. The addition of the axioms for a new type is a simple exercise in the specification of abstract data types.

<table>
<thead>
<tr>
<th>Table 2. FP Axioms</th>
</tr>
</thead>
<tbody>
<tr>
<td>( id \circ x = x ) where ( id ) is the identity function</td>
</tr>
<tr>
<td>( x \circ y = x ) for constant function ( x ) in the domain that ( y ) is defined†</td>
</tr>
<tr>
<td>( cons \circ { x, y, \ldots, y_n } = { x, y, \ldots, y_n } ) (( n \geq 0 ))</td>
</tr>
<tr>
<td>( hdo \circ { x, y, \ldots, y_n } = x ) in the domain that ( x, y, \ldots, y_n ) is defined†</td>
</tr>
<tr>
<td>( tlo \circ { x, y, \ldots, y_n } = { x, y, \ldots, y_n } ) (( n \geq 1 )) in the domain that ( x, y, \ldots, y_n ) is defined†</td>
</tr>
<tr>
<td>( m \circ { x, y, \ldots, y_n } = { x_m, \ldots, y_n } ) where ( 1 \leq m \leq n ) in the domain that ( x, y, \ldots, y_n ) is defined†</td>
</tr>
<tr>
<td>[Thus the selector function, ( m = hdo \circ t ) for ( n \geq 1 ).]</td>
</tr>
<tr>
<td>( null \circ { } = T )</td>
</tr>
<tr>
<td>( null \circ (cons \circ { x, y }) = F ) in the domain that ( a \circ { x, y } ) is defined</td>
</tr>
<tr>
<td>( f \circ (g \circ h) = (f \circ g) \circ h ) (associativity of composition)</td>
</tr>
<tr>
<td>( f \circ (p \circ q) = p \circ (f \circ q) ) for ( q \neq f \circ q )</td>
</tr>
<tr>
<td>( p \circ q = p \circ r \rightarrow s = p \circ q \circ s )</td>
</tr>
<tr>
<td>( [f_1, \ldots, f_n] \circ g = [f_1 \circ g, \ldots, f_n \circ g] )</td>
</tr>
<tr>
<td>( (p \circ f; q; \ldots) = p \circ (f \circ q; \ldots) )</td>
</tr>
<tr>
<td>† A function ( f ) is defined in the domain ( D ) if ( f : x + 1 ) for ( x + 1 \in D ).</td>
</tr>
</tbody>
</table>

In our study of algebraic transformation techniques, we will consider functions defined by equations of the form \( f = p \rightarrow q \); \( Hf \) for some functional \( H \), where the functions \( p, q \) are fixed. Properties of such functions are then determined by the structure of \( H \), which is frequently expressed in terms of functional compositions. The composition of the functionals \( H_1 \) and \( H_2 \) is written as a juxtaposition, and \( H = H_1 \circ H_2 \) is defined by \( Hf = H_1(f \circ H_2) \) for function variable \( f \). We also denote the identity functional by \( ID \), defined by \( ID(f) = f \), so that \( ID \circ H = H \) for \( H \) functional. The \( n \)-fold functional composition of \( H \) with itself is written as \( H^n \), and is defined by \( H^n = ID \circ H \circ ID \circ H \circ \ldots \circ ID \circ H \) for \( n \geq 1 \).

Section 3 considers the special case in which the functional \( H \) is linear, giving a linear function \( f \) and leading to a loop representation for \( f \). Non-linear functionals \( H \) are considered in Sections 3 and 4.

3. Recursion Removal

3.1 Linear functions and the linear expansion theorem

Intuitively, a linear function is one that generates a number of function calls to itself which grows linearly in the magnitude of the argument to which it is applied. Thus, certainly, tail-recursive functions are linear, and these are defined, in the terminology of the previous section, by equations of the form \( f = p \rightarrow q; \) \( Hf \) for some fixed function \( k \), corresponding to a functional \( H \) defined by \( Hf = f \circ k \). However, the tail-recursive functions form a small subset of the linear class, and in particular, any function with a 'comb-shaped' reduction tree, i.e. one that grows unidirectionally down its left spine, must be linear – the factorial function is an example.

Given the definition \( f = p \rightarrow q; \) \( Hf \), we can derive an expanded version by substituting for the occurrence of \( f \) in the right-hand side by the right-hand side itself. This gives the equation

\[ f = p \rightarrow q; H(p \rightarrow q; Hf) \]

and repeating the operation gives

\[ f = p \rightarrow q; H(p \rightarrow q; H(p \rightarrow q; Hf)) \]

and so on. In order to obtain a number of terms that grows linearly, we need the functional \( H \) to distribute through a conditional expression, and a sufficient condition for this is that we have the property that for all functions \( a, b, c \),

\[ H(a \rightarrow b; c) = H(a \rightarrow Hb; Hc) \]

for some functional \( H \). \( H \) is a linear functional if this property holds together with the additional condition that if for any object \( x, H \perp x \perp \perp \), then, for all functions \( a, H_a; x = T \). The latter condition is necessary to ensure convergence in the expansion theorem given below, and it clearly holds if the functional \( H \) is strict, i.e. \( H \perp = \perp \).

For a linear functional \( H \), the above expansion then simplifies to

\[ f = p \rightarrow q; H(g \rightarrow Hq; \ldots; H_n p \rightarrow H^n q; H^n f) \]

for all \( n \geq 0 \). Thus we have the Linear Expansion Theorem (LET).

If \( H \) is a linear functional, then the function \( f \) defined by \( f = p \rightarrow q; Hf \) for fixed functions \( p, q \), is given by \( f = p \rightarrow q; H_1 p \rightarrow H_1 q; \ldots; H_n p \rightarrow H_n q; H^n f \).

The linearity condition, \( H(a \rightarrow b; c) = H(a \rightarrow Hb; Hc) \), is sufficient but not necessary for \( f \) to have a non-recursively expandable, and the property that \( H(a \rightarrow b; c) = H(a \rightarrow Hb; Hc) \) for some functional \( H \) would also be sufficient. Indeed, this is still not necessary, and expansion theorems for certain classes of non-linear functions are considered by Williams and Harrison.6,15

Now we can easily identify a basic set of elementary linear forms by specifically testing for the linearity conditions, but more importantly, the set of linear forms is closed, so that a great many linear forms can be generated. The Functional Composition Theorem of Backus states that if the functionals \( H \) and \( G \) are linear, then their composition, \( HG \), is also linear and has a predicate transformer \( H \circ G \), the composition of the predicate transformers of \( H \) and \( G \) respectively.3

The basic collection of linear functionals called simple linear functionals, or SLFs for short – that we consider are defined together with their predicate transformers in Table 3.

The letters \( f \) and \( a \) are used to denote function variables, all other letters denoting arbitrary fixed functions, and the sequence of three dots ... denotes a finite sequence of fixed functions, say \( r_1, \ldots, r_n \) for \( n \geq 0 \).
<table>
<thead>
<tr>
<th>Linear form, ( Hf )</th>
<th>Predicate transformer form, ( H_a )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r )</td>
<td>( T )</td>
</tr>
<tr>
<td>( f \circ r )</td>
<td>( a \circ r (H_a = H) )</td>
</tr>
<tr>
<td>( r \circ f )</td>
<td>( a (H_a = ID) )</td>
</tr>
<tr>
<td>( [\ldots, f, \ldots] )</td>
<td>( a (H_a = ID) )</td>
</tr>
<tr>
<td>( (\text{for example } Hf = [f, r] \text{ and } Hf = [r, f] \text{ are common}) )</td>
<td></td>
</tr>
<tr>
<td>( p \circ q \circ f )</td>
<td>( p \circ T \circ a )</td>
</tr>
<tr>
<td>( p \circ r \circ f )</td>
<td>( p \circ a \circ T )</td>
</tr>
<tr>
<td>( f \circ q \circ r )</td>
<td>( a (H_a = ID) )</td>
</tr>
</tbody>
</table>

Each of the forms listed in Table 3 can be shown to be linear very easily. For example, in the case of the first, for arbitrary functions \( a, b, c \), \( H(a \circ b \circ c) = r = T \rightarrow r = H_a \circ H_b \circ H_c \) if \( H_a = T \). Similarly, in the second case, \( H(a \circ b \circ c) = (a \circ b \circ c) \circ r = a \circ r \circ b \circ r \circ c \circ r \) (by the conditional axioms) = \( H_a \circ H_b \circ H_c \) so that \( H_i = H \). The proofs for the other cases are equally straightforward. Moreover, the second condition for linearity, which we have largely ignored, is also easily verified; recall that all FP functions and sequence constructions are strict.

By applying the Composition Theorem to these SLFs we can identify a very rich class of linear forms and their predicate transformers. It is the corresponding linear functions that we will transform into iterative form – loops or tail recursive functions – in the next section. Note that all forms \( Hf \) which are built from the three primitive functionals of function composition, construction, and conditional, and contain only a single occurrence of the function variable \( f \), are linear. Thus for example the form \( Hf = h \circ o \circ f \circ o \) is linear and has predicate transformer defined by \( H_a = a \circ o \). This follows since \( H = H_1 \circ H_2 \circ H_3 \) where \( H_1 \circ f = h \circ o \), \( H_2 \circ f = [i, f] \), \( H_3 \circ f = f \circ o \). Thus \( H_1 = ID \), \( H_2 = ID \), \( H_3 = a \circ o \), and \( H_i = ID \). The significance of this example is that it defines the factorial class of linear functions. If \( f = p \circ q \circ Hf \), then assigning the functions \( eq \circ 0 \) to \( p \), \( i \) to \( q \), \( * \) to \( h \), \( id \) to \( i \) and \( sub1 \) to \( j \) gives the definition of factorial, whereas the assignments \( null \) to \( p \), \( [] \) to \( q \), \( cons \) to \( h \), \( hd \) to \( i \) and \( tl \) to \( j \) yield the function ‘reverse’ that reverses sequences.

The decomposition of a functional expression into SLFs is precisely what must be performed by the parser of such expressions, and minimal extra effort is needed to obtain the predicate transformers. The same will apply in the next section when we build the loops corresponding to linear functions during parsing.

### 3.2 Iterative forms of linear functions

Using the LET of the previous subsection, we may form an iterative implementation corresponding to a linear function \( f \) with definition of the form \( f = p \circ q \circ Hf \). Such an implementation may take the form of either a loop (or pair of loops) defined on objects, or an equivalent tail-recursive version of \( f \). We consider the first alternative, which is slightly the simpler and requires no further transformation for an imperative implementation; the second is given in Ref. 7.

Now, given an object \( x \) as argument, if \( f \circ x \) is defined, then by the LET, \( f \circ x = (H^p q) \circ x \), where \( n \) is the smallest integer such that \( (H^p q) \circ x = T \). Thus, for the application of \( f \) to \( x \), \( f \) can in principle be ‘computed’ iteratively in a loop, starting with the known function \( q \) in the ‘accumulator’ and applying \( H \) to the accumulator \( n \) times. The result of the application is then obtained by applying the final contents of the accumulator to \( x \). Of course, in general the increasing complexity of the functions in the sequence \( q, Hq, H^2q, \ldots \) renders this approach impractical, and further transformation is necessary to obtain a loop at the object level. The idea behind the forthcoming analysis is that if a loop implementation does exist for the expression \( f \circ x \), the loop should comprise the assignment to an accumulator of an expression which depends on two variables: the current accumulator and some loop input variable, \( x_n \) given by the loop count \( i \) on the \( i \)th iteration.

Let \( r_i = (H^p q) \circ x \) for \( 0 \leq i \leq n \) and some set \( \{x_i | 0 \leq i \leq n\} \) with \( x_n = x \), where \( n \) is the integer defined above. Then \( r_n = f \circ x \) by the LET, and if we can find an object-level expression \( E_u \) corresponding to \( H \) with the property that \( r_i = E_u(r_{i-1}, x) \) for \( 1 \leq i \leq n \), the set of objects \( \{r_i | 1 \leq i \leq n\} \) can be computed iteratively in a loop using destructive assignment to an accumulator variable, \( r = E_u(r, x) \), giving the result \( r = r_n \) in the accumulator as required after the final iteration. If the functional \( H \) is built entirely from SLFs, we will see that \( r \) and \( \{x_i\} \) can be computed for any object \( x \) from \( p \) and \( H \), which can be obtained as described in the previous section. Since we also know that the initial value of the accumulator is \( r = r_0 = q \circ x_0 \), it remains only to determine the expressions \( E_u(v, u) \) for object variables \( v, u \).

Again, we first determine the expression \( E_u \) when \( H \) is an SLF, and then define it for the case in which \( H \) is a composition of SLFs in terms of the corresponding expressions of the constituents of the composition, which are known. That is, if \( H = H_1 \circ \ldots \circ H_n \) where each \( H_i \) is an SLF (\( 1 \leq i \leq n \)), \( E_u \) is defined in terms of \( E_{H_1}, \ldots, E_{H_n} \) and the predicate transformers, \( H_{\text{tr}} , \ldots, H_{\text{tr}} \). Here we consider only functionals \( C \) which are compositions of the following three of the SLFs, \( H \), introduced in the previous section: \( Hf = f \circ r \), \( Hf = r \circ f \) and \( Hf = \ldots \), \( Hf \), where \( r \) is an arbitrary fixed function, and ... represents a finite sequence of fixed functions. We call functionals \( C \) of this type composite linear functionals, or CLFs. A substantially more general definition of CLFs and corresponding loop syntheses is given in Ref. 7.

The SLF case that \( Hf = r \) for all functions \( f \) is trivial, the function defined by the equation \( f = p \circ q \circ r \) being non-recursive; the same also applies if \( H \) is a composition of SLFs that includes one of this type.

We will need the following lemma in our derivation of the expression \( E \) for CLFs:

**Lemma 1**

Given CLF \( H \), \( Hf = f \circ o \circ H, id \) for all functions \( f \).

We denote the function \( H, id \) by \( j \), and in fact it is equal to \( j \) in the case of the factorial-like form \( Hf = h \circ o \circ f \). The proof of the lemma is by induction on the number of SLFs in the composition \( H \).

**Base case**

If \( Hf = f \circ o \), then \( H, id = id \circ r \) and so \( j = r \). Thus \( Hf = f = f \circ o \).

If \( Hf = r \circ o \), then \( H, id = ID \) and so \( j = id \). Thus, \( Hf = f = f \circ o \).
Inductive case
Suppose that $H = H_1 \ldots H_m$ for $m > 2$, and let $L = H_1 \ldots H_{m-1}$. Then we may assume inductively that $L_i f = f \circ L_i id$ for all functions $f$. But,

$H f = L(H_m f)$ by the Functional Composition Theorem,

$= H_m f \circ L id$ by the inductive hypothesis,

$= f \circ H_m id \circ f \circ L id$ by the base case,

$= f \circ L(H_m id)$ by the inductive hypothesis,

$= f \circ L id$ by the Functional Composition Theorem,

which completes the proof.

Now consider the function $f$ defined by the equation $f = p \to q \to H f$, and the terminating application $f : x$. The first consequence of the above lemma is that the least integer $n$ such that $H^n f : x = T$ is the least $n$ such that $p : f^n : x = T$, which can be computed easily in a simple while loop, by successively applying $f$ to $x$ and then $p$ to the result. In fact we make further use of this same while loop by defining the set $\{x_i | 0 \leq i \leq n\}$ by $x_0 = x$ (which we are given), and $x_{i+1} = f : x_i$ for $1 \leq i \leq n$.

The key property that we require of the expression $E$ is that $(H f) : x = E(H f) : x$ for all functions $f$ and objects $x$. For then we have the following theorem.

Theorem 1
Given CLF $H$, integer $n \geq 0$ and object $r_0$, let the set $\{r_i | 1 \leq i \leq n\}$ be defined by $r_0 = (H f) : x_0$, where the set $\{x_i | 0 \leq i \leq n\}$ is defined as above. Then $r_i = E(r_{i-1}, x_i)$ for $1 \leq i \leq n$.

Proof
$r_1 = (H f) : x_1 = (H(H f)) : x_1$ for $1 \leq i \leq n$

$= E(H(H f)) : x_1$ by hypothesis, substituting $H f$ for $f$, and abbreviating $E$ by $E$

$= E(H f) : x_1$ by Lemma 1, substituting $H = H$

$= E(r_{i-1}, x_i)$ by definition of $x_{i-1}$ for $i \geq 1$

$= E(r_{i-1}, x_i)$ by definition of $r_{i-1}$ for $i \geq 1$

From this, the iterative implementation of the expression $f : x$ follows immediately as a pair of loops: the while loop referred to above which computes $n$ and the set $\{x_i | 0 \leq i \leq n\}$, and the for loop which is entered $n$ times to compute $f : x$ as $r$, by starting with $r = q : x_0$ and successively assigning $(n$ times) $r = E(r, x)$.

It therefore remains to define appropriate expressions $E$ for arbitrary CLFs $H$, which satisfy our hypothesis. The required definitions are given next, and their validity is established in the lemma that follows them.

If $H f = f \circ r$, then $E(u, v) = u$ for object variables $u, v$.
If $H f = r \circ f$, then $E(u, v) = r : u$ for object variables $u, v$.
If $H f = r_1 \ldots r_n$, then $E(u, v) = r_1 \ldots r_n$ for object variables $u, v$.

If $H = BC$ where $B$ is one of the three SLFs considered in this analysis and $C$ is a CLF, then $E(u, v) = E_B E_C (u, B id : v, v)$

The theory is therefore completed with the following.

Lemma 2
The above definition of the expression $E_H$ associated with a CLF $H$ satisfies the property that $(H f) : x = E_H ((H f) : x, x)$ for all functions $f$ and objects $x$.

Proof
We again use induction on the number of SLFs in the composition $H$.

Base case
If $H f = f \circ r$, then $E_A (H f, x, x) = H f : x = (f \circ r) : x = H f : x$.

If $H f = r \circ f$, then $E_A (H f, x, x) = r : (H f, x) = r : (f : x) = (r \circ f) : x = H f : x$.

If $H f = r_1 \ldots r_n$, then $E_A (H f, x, x) = r_1 \ldots r_n : (H f, x) = r_1 \ldots r_n : (f : x)$.

Inductive step
Suppose that $H = AB$ where $A$ is an SLF and $B$ is a CLF. Then,

$E_B ((H f) : x, x) = E_B (E_A (H f, x, A id : x), x)$ by definition of $E_B$

$= E_A (E_B (B f, A id : x), x)$ by Lemma 1.

Thus,

$E_B ((H f) : x, x) = E_B (B f, A id : x)$ by the inductive hypothesis

$= E_A (A(B f) : x)$ by Lemma 1

$= A(B f) : x$ since $A$ is an SLF of the base case

$= H f : x$

completing the proof.

We can see how the transformation works by considering as an example the generalised factorial function of Section 3.1, defined by the equation $f = p \to q \to H f$. Now, for the SLFs $H_1, H_2, H_3$ we have the respective expressions $E(u, v) = h : u, E(u, v) = i : u, v, E(u, v) = u$. Thus by Theorem 1 the assignment statement in the for loop is

$r = E_H (r, x)$

i.e. $r = E_B (E_A (r, H_1 id : x_1), H_2 id : x_1)$

I.e. $r = E_A (E_B (E_A (r, x_1), x_1), x_1)$

I.e. $r = h : i : x_1, r$

In the case of the factorial function, therefore, where $p = eq 0, q = 1, h = \ast, i = id$ and $j = sub 1$, we obtain the loop assignment $r = \ast : i : + : x_1, r$, and $H id = j = sub 1$.

For reverse, where $p = eq, q = \{\}, h = cons \ast, i = \ldots$ and $j = t l$, we get $r = cons r, h : d : x_1, r$, and $H id = j = t l$.

These yield the familiar imperative loops. For example, reverse $: x$ would be implemented by the following instantiated template,

if $null : x$

then $<=$

else

begin

Loop Inputs: = PUSH($x$, EMPTY_STACK); $n = 1$;

end.
while null:tl: TOS(Loop_Inputs) ≠ T do
begin
n := n + 1;
Loop_Inputs := PUSH(tl: TOS(Loop_Inputs), Loop_Inputs);
end
r := << > ;
for i := 1 to n do
begin
r := consr: < hd: TOS(Loop_Inputs), r > ; (* since x is at top of the input stack *)
Loop_Inputs := POP(Loop_Inputs);
end

In fact, the transformation given so far provides only a minimal improvement in performance, since it is only the calls to the recursive function that have been removed. However, this is not surprising in view of the generality of the scheme which applies to all linear recursive functions defined by a CLF, some of which cannot be optimised any further, for example reverse. Such a function would first generate a sequence of recursive calls to itself until its argument reached the base case. It would then complete the corresponding function applications using the arguments which must have been saved in the initial calling sequence. If it is impossible to reconstruct these arguments by starting with the argument of the base case and repeatedly applying some fixed function, there is no alternative to implementing the function's application in two phases: one to precompute and stack all of the arguments and reach the base case, the other to successively complete the evaluations of the function's defining expression instantiated with the stacked argument values. In the case of reverse, each argument stacked after the first (that of the top-level application) is the tail of the previous one, which cannot therefore be reconstructed because of the destructive nature of the tail function which discards the head of its argument. However, in the case of factorial, the next argument stacked is the integer one less than the current one, and so in this case the previous argument can be reconstructed by adding one. Thus there is no need to stack the arguments in an implementation of the factorial function.

In general, the 'next' argument is obtained by applying the known function j = Hj id to the 'current one'. Thus we can avoid building a stack if we know that j has an inverse, and what that inverse, j⁻¹, is. In this way we would achieve some space saving, but we cannot dispense with the while loop entirely since we need to find the number of iterations required in the for loop, n, and also the value of the argument, x₀, in the base case. For certain functions, these two values can be determined without recourse to executing a while loop. For example, if the predicate function p = eq0 and j = sub1, then n = x, the top-level argument, and x₀ = 0.

In the case of factorial, we therefore obtain
begin
n := A; (* A is the number whose factorial
x := 0; (is needed *)
r := 1;
for i := 1 to n
do
x := add1 : x;
r := * : < r, x > ;
end

This is the normal imperative implementation of factorial, working from the base case upwards. However, an equally common loop implementation of this function works 'downwards' from the top-level argument, i.e. in reverse. Such loop reversal depends here on the property that the primitive function multiply, i.e. *, is associative. In general, we may dispense with the for loop by reversing the while loop if a similar, less restrictive, associativity condition holds which fits in very naturally into the above transformation scheme. This condition relates to the function Eµ, and we do not need j = Hj id to possess an inverse since the while loop uses this function as it stands to drive its iteration. We state the result from Ref. 7, using the notation we have defined above.

Let a₀ = y₀ = x, and aᵢ = Eµ(yᵢ, aᵢ₋₁), yᵢ = j(yᵢ₋₁) (i ≥ 1). Then Hf: x = Eµ(q : yᵢ, aᵢ₋₁), where n = min{j : p: j' : x = T} if Eµ(u, Eµ(v, w)) = Eµ(Eµ(u, v), w) for all objects u, v, w.

Under this condition, it can be shown similarly that the equivalent tail-recursive form of the function f is

\[ f = p \rightarrow q; E0[q \circ j0, 1] \circ g_0[i, id] \]

where

\[ g = p \circ 0 \rightarrow id; g_0[j0, 1] \]

and the FP function E is defined by E: < u, v > = Eµ(u, v).

Thus in the case of factorial we obtain Eµ(u, v) = *; < v, u > (as above) so that E = *o[2, 1] = * (since * is commutative), giving tail-recursive form

\[ f = p \rightarrow q; E0[q \circ j0, 1] \circ g_0[i, id] \]

where

\[ g = p \circ 0 \rightarrow id; g_0[j0, 1] \]

or the following loop
begin
r := x;
while eq0: (sub1 : x) ≠ T do
begin
x := sub1 : x;
r := * : < r, x > ;
end

3.3 Non-linear functions

The appeal of the function-level algebraic approach to transformation is that it applies to substantial classes of functions – we have only considered part of the linear class – and the transformations are mechanisable. Thus it makes automatic optimisation of many functional programs possible. All of the information required for the translation of the linear class of functions considered in this section into iterative form, for example, can be obtained during the parsing of the functions' defining equations, which would still be necessary in any implementation. However, the approach is certainly not limited to linear functions, many non-linear functions having equivalent linear versions, from which iterative implementations follow from the above results. We call such a transformation linearisation. The non-linear functions concerned are called degenerate multilinear, the defining equation of such a function, f, having the form f = p \rightarrow q; Hf where Hc = M(e, ..., v) for function variable v, and M is a multilinear functional of n

THE COMPUTER JOURNAL, VOL. 31, NO. 3, 1988 235
parameters, which is defined by the following. An $n$-multilinear form $M(f_1, \ldots, f_n)$ is independently linear in each of its arguments considered separately. i.e. for $1 \leq i \leq n$, $M(f_1, \ldots, f_i, \ldots, f_n) = M(a \rightarrow M(f_1, \ldots, f_i, b, f_{i+1}, \ldots, f_n))$ for some fixed functionals $M_i$. (In fact, as in the definition of a linear form, we also require that if there exists an object $x$ such that $M(f_1, \ldots, f_i, x, f_{i+1}, \ldots, f_n) = x_\perp$ then for all functions $a, M(a \cdot x) = T$, which always follows if $M$ is strict.)

For example, the Fibonacci function is degenerate bilinear:

$$\text{fib} \ n = \begin{array}{ll} \text{if} & n < 1 \\
\text{then} & \text{fib}(x-1) + \text{fib}(x-2) \\
\text{else} & \text{fib}(x-1) \end{array}$$

which has the following variable free form

$$\text{fib} = l e l \rightarrow 1 + o [\text{fib} \circ \text{sub}1, \text{fib} \circ \text{sub}2]$$

where $lel$ is defined by $lel \cdot x = T$ if $x \leq 1$, and $F$ if $x > 1$ for integer $x$. This can be written as

$$\text{fib} = lel \rightarrow 1; G(f, f)$$

where

$$G(u, v) = + o [u \circ \text{sub}1, v \circ \text{sub}2]$$

The following is the simplified result of Ref. 6, which can be used for the linearisation of fib.

**Theorem 2**

Let $f = p \rightarrow q; M(f, f)$, where $M$ is bilinear with predicate transformers $M_a$ and $M_b$. If

(a) There exists a strict linear form $M_0$ with predicate transformer $M_0$ such that

$$M_0 = M_0^a \circ M_0^b \quad \text{for integers} \quad 1 < h^a < h^b, \ k = 1, 2$$

(b) There exists a bilinear form $M'$ with all its predicate transformers equal to $M_0$ such that

$$M'(u_1, u_2) = M'(M_0^{-1} u_1, M_0^{-1} u_2).$$

If $M_0$ has an inverse, $M'(u_1, u_2) = M(M_0 M_0^{-1} u_1, M_0 M_0^{-1} u_2))$

(c) For all objects $x$, $p \cdot x = T$ implies $(M(p) \cdot x) = T$ for integers $j > 0$

then

(1) There exists a function $g$ such that $f = 1 \| g$ and

$$g = p \rightarrow q; Hg \text{ where } Hw = [M'(1 \circ w, 2 \circ w), M_0(1 \circ w)]$$

and $q = [q, M_0^{-1} q]$.

(2) $H$ is linear with predicate transformer $H_1 = M_0$, so that the function $g$ is linear. (In fact we can also say that $f$ is the composition of the selector $1$ with the least fixed point of the equation for $g$.)

It is easy to see that the $G$ in the definition of fib is bilinear, with predicate transformers $G_1, G_2$ respectively given by $G_1 a = a \circ \text{sub}1$ and $G_2 a = a \circ \text{sub}2$ for function variable $a$. It is easily shown that $G$ satisfies the conditions of the above linearising theorem – in particular because $G_1$ is a power of $G_2$, i.e. $G_1 = G_2^3$. Thus, in the case of fib, we have $G(u, v) = + o [u \circ \text{sub}1, v \circ \text{sub}2]$. $p = lel$ and $q = 1$, so that its linear form is given by

$$\text{fib} = 1 \| g \text{ where } g = lel \rightarrow [1, 1]; [+ o [1 \| \circ \text{sub}1, 2 \| \circ \text{sub}1]]$$

upon direct substitution. The generalised scheme for transforming linear functions into loops considered in the previous section could now generate the usual loop implementation of the Fibonacci series, but some very straightforward simplification (using the FP axioms) yields

$$g = lel \rightarrow [1, 1]; [+ o [1 \| 2, l \| o \| o \| s \| b1]$$

Thus we obtain $g = lel \rightarrow [1, 1]; [+ l \| o \| o \| s \| b1] \| \text{using the fact that} \ 1, 2 = id \ \text{for correctly typed expressions. i.e. when each side is applied to a sequence of two elements. Applying the Linear Expansion Theorem to this function, we get}$

$$fib = 1 ; (g ; x)$$

$$= 1 ; (\text{H} ; [1, 1] ; x) \text{ where } n = \text{min}[\text{lel} \circ \text{sub} \perp, x = T)$$

$$= 1 ; ([+ l \| l ; 1, 1] ; 1) ; x = x \text{ where } n = \text{min}[x - i \leq 1 = T]$$

$$= 1 ; ([+ l \| l ; 1, 1] ; 1 ; 2, 1, 3)$$

This reflects precisely the iterative implementation of the Fibonacci function, which uses a pair of accumulators, repeatedly swapping and adding them.

Applying the techniques of Harrison and Koshnevisan directly, we obtain the following assignment in the loop corresponding to the linearised version of the Fibonacci function:

$$r_n = < + ; < 1 \circ r_{n-1}, 2 \circ r_{n-1} > , 1 \circ r_{n-1} >$$

This unsimplified version is in fact only marginally less efficient than the above, explicitly separating the components of $r_{n-1}$ before applying $+$.  

### 4. Memoisation

Memoisation is an alternative route to the efficient implementation of non-linear functions. A memo-function, originally introduced by Michie, is like an ordinary function except that it remembers all the arguments it has been applied to, together with the corresponding results computed from them. If a memo-function is ever re-applied to an argument it does not re-compute the result, but just re-uses the result computed earlier. Therefore one can see that memoisation can be used to replace a potentially expensive computation by a simple table look-up. The classic example is again the Fibonacci function. Since each call of fib generates two recursive calls, the cost of computing fib(n) is exponential in $n$. However, memoised fib will execute in linear time, since for each value $n$, fib(n) is computed only once.

By far the most serious drawback associated with memoisation is that memo-functions can interfere with garbage collection. This is because new entries continue to be added to the table even if particular arguments will never be passed to the function again, and it is difficult to know when it is safe to delete such entries. A continuously growing memo-table may ultimately bring the execution to a halt once it uses up the entire storage available, and the cost of memo-table lookup will also be increasing with its size. In this section we introduce a variant of memo-functions, called self-garbage-collecting (s.g.c.) memo-functions which can be used to linearise the cost of calls of a non-linear function to itself. 

#### 4.1 An implementation of self-garbage-collecting memo-functions

S.g.c. memo-functions can be used to linearise certain functions that have execution time which is quadratic/
exponential in the size of their argument due to the calls from within the function’s defining expression to the function itself. For such specific use of memo-isation it is sufficient to create a local memo-table for each ‘top-level’ application of the non-linear memo-function, but this does slightly restrict its use. For example, two top-level applications will not be a memo-table lookup of the first. Although this implies that our variant is less powerful than full memoisation (where cost of some top-level applications may be constant), it is as powerful as program transformation, since the transformed function will also execute every ‘top-level’ application in the same way. Thus we replace the function definition of the non-linear function $f$ by a call to the function $f'$ (where $f'$ is the s.g.c. memoised version of the function $f$) followed by the garbage collection of the memo-table of $f'$. Whenever inserting a new entry, the s.g.c. memoised version of the function attempts to delete (or re-use) the entries that will no longer be useful, and we next describe an implementation of s.g.c. memoisation.

### 4.1.1 A garbage-collecting INSERT operation

Associated with each memo-function we have a function called the **table-manager**, which is generated by the compiler (see Section 3.4). The table-manager functions are of type

\[
\text{type_of_argument_of}_{f} = \text{list}\left(\text{type_of_argument_of}_{f}\right)
\]

This function is executed by the INSERT operation of memo-tables at run-time in order to find the particular arguments whose entries can be safely deleted (or overwritten) as a result of the insertion that is about to follow. Intuitively one would expect that such a function for \text{fib} may be

\[
\text{lambda } x \Rightarrow [x - 2]
\]

Note that we are using a Hope-like syntax for the definition of our table-manager functions, where ‘[ ]’ will denote a list of objects. Thus in the above we are specifying a list which has just one element in it. Similarly for a non-linear tree-using function the table-manager function may be

\[
\text{lambda } \text{Leaf}(x) \Rightarrow \text{nil} | \text{Node}(l, r) \Rightarrow [l, r]
\]

Note that the meaning of the latter is as follows: given a pointer to a Node, delete the two entries that correspond to the left and right sub-trees.

Note that the insert operation does not need to scan the entire table. It can simply do an ordinary LOOKUP operation on the table to locate the entry for deletion, hence making the most of hashing or a tree organisation of the table. The space gains are achieved by increasing the cost of INSERT by a small amount, since INSERT now has to execute the table-manager each time. However, note that no extra apparatus is needed to execute the table-managers since they are simply expressed in the functional language that we are compiling, and we will see that the table-managers are non-recursive; built from the ‘predicate transformers’ in the case of a degenerate multilinear function. The s.g.c. memoised version of a function will not insert entries for argument values where the result of the application is the base-case value of the function. This is primarily for efficiency reasons, since clearly in such cases it is possible to avoid a LOOKUP, and we also cannot identify any previously created memo-table entry to delete. Therefore we synthesise the definition of our s.g.c. memoised version so that base-case conditions are tested for prior to doing a LOOKUP. Should the base-case condition be true then the memo-function returns the base-case result and will not alter the memo-table. Otherwise the memo-function will proceed by doing a LOOKUP to see if the result is already known. Hence for a non-linear tree-using function the table-manager function could be

\[
\text{lambda } \text{Node}(l, r) \Rightarrow [l, r] \quad \text{instead of } \text{lambda } \text{Leaf}(x) \Rightarrow \text{nil} | \text{Node}(l, r) \Rightarrow [l, r]
\]

since we know that base-case results are never inserted and hence the table-manager will never be applied to a leaf. This is illustrated by the source-to-source transformations of Section 4.5, which fully explain the operational semantics of our memoisation.

### 4.2 The underlying theory for s.g.c. memo-isation

In this section we define a large and automatically detectable class of recursive functions which we call **composite multilinear**. We first give some further notation:

\[(1) f = (f_1, \ldots, f_n) \text{ for some } n > 0\]
\[(2) \text{ if } h = (f_1, \ldots, f_n, h_1, \ldots, h_m) \text{ where } f = (f_1, \ldots, f_n) \text{ and } h = (h_1, \ldots, h_m) \]

**Definition 3.2**

$M$ is a **Composite m-Multilinear Form (m-CMF)** if, letting $i$ range over $1 \leq i \leq m$, either (a) $M_f = f$ (i.e. $M = ID$)

or (b) $M_f = [M_1(f_1^{(1)}), M_2(f_2^{(2)}), \ldots, M_n(f^n)]$ where $f = f^{(1)} || f^{(2)} || \ldots || f^{(n)}$, and for $i \leq i \leq n, M_i$ is an $m_i$-CMF and $m_i > 0$ for at least one $i$. Then $M$ is $(\Sigma_{m_i}, m_i)$-multilinear with

\[M_j = M_{k_j} \quad \text{s.t.} \quad (\Sigma_{j=1}^{k_j} m_j) < j \quad \text{and} \quad (\Sigma_{j=1}^{k_j} m_j) \geq j\]

and $j = (\Sigma_{j=1}^{k_j} m_j)$, for integer $k$ and all $j, 1 \leq j \leq (\Sigma_{j=1}^{k_j} m_j)$.

or (c) $M_f = P_{f^{(1)}}^{(1)} A_{f^{(2)}}^{(2)} B_{f^{(3)}}^{(3)}$ where $f = f^{(1)} || f^{(2)} || f^{(3)}$, and $P, A, B$ are $m_i$-CMF, $m_2$-CMF and $m_3$-CMF, respectively, and $m = m_1 + m_2 + m_3$. Then $M$ is an m-CMF with p.t.s. $M_j = P_{j=1}^{(1)} m_j$ $j \leq m_1$, $M_j = A_{j=1}^{(2)} m_j$ $j \leq (m_1 + m_2)$ and $M_j = B_{j=1}^{(3)} m_j$ $j \leq (m_1 + m_2)$. Then $M$ is an m-CMF. Then $M$ is an m-CMF with p.t.s. $M_j = P_{j=1}^{(1)} m_j$ $j \leq m$.

or (d) $M_f = M_{f^{(1)}}^{(1)} M_{f^{(2)}}^{(2)}$ where $f = f^{(1)} || f^{(2)}$ and $M' = M_{f^{(1)}}^{(1)} M_{f^{(2)}}^{(2)}$. Then $M$ is an m-CMF with p.t.s. $M_j = P_{j=1}^{(1)} m_j$ $j \leq (m_1 + m_2)$.

The proofs that each of the above cases is a multilinear form, and that the expressions given for their p.t.s. are correct, can be found in Ref. 10. In fact certain technical conditions must hold for part (c) to be correct, but these are a little obscure and we will not be concerned with them.

### 4.2.1 Detection of CMFs

Detection of CMFs is automatic and requires only a slightly modified parser; one that detects composite multilinear forms and identifies their degree of multilinearly and predicate transformers. The detection of
CMFs is straightforward, since the parser in any case needs to establish the hierarchical structure of any function definition. The base case is recognised when the function, say \( f \), is parsed. The degree of multilinearity is then simply 1 and the p.t.s is ID. CMFs are detected recursively by recognising simpler CMFs used in conditionals, compositions or constructions; their degree of multilinearity and p.t.s are found using their definition given in the previous section.

4.2.2 Outline of s.g.c. memo-isation

Recursive functions that lead to redundant computations can be expressed in the form

\[
f = p \rightarrow q; \quad H(f, \ldots, f)
\]

where \( p \) and \( q \) are fixed functions (i.e. they do not involve \( f \)) and \( H(f, f, \ldots, f) \) is a functional expression involving two or more occurrences of \( f \). We consider recursive function definitions of the form \( f = p \rightarrow q; \quad H(f) \) where \( H \) is a degenerate multilinear form, i.e. where \( H(f) = M(f, \ldots, f) \) for some multilinear form \( M \), defined in Section 3.3. Our approach to an efficient implementation of s.g.c. memoisation is to analyse the predicate transformers of the degenerate multilinear forms in order to establish when a particular result of a memo-function (i.e. corresponding to some particular argument) can be used in the computation of another. More precisely we need to establish the particular redundancy relationship between the various predicate transformers \( M_n, \ldots, M_1, \ldots, M_i \). In this paper we will consider only one such redundancy relationship, namely when there is a highest common generator \( M_0 \) for all the p.t.s. \( M_i \) (\( 1 \leq i \leq n \)). \( M_0 \) is defined by \( M_i = M_{i-1} \) for the minimum set of positive integers \( i \), such if a set exists (\( 1 \leq i \leq n \)). \( M_0 \) is therefore the highest rather than just any common generator.

For example in the case of \( fib \), \( M_1 = M_2 \) and \( M_2 = M_3 \) where \( M_3 = \alpha \cdot sub \cdot 1 \). Such information not only tells us when particular results of the memo-function are re-usable, but also shows whether s.g.c. memo-isation will be worthwhile.

Note that if the p.t.s. have a common generator \( M_0 \) then \( M_i = M_{i-1} = M_{i-2} = \alpha \cdot a \cdot sub \cdot 1 \), and this implies that the p.t.s. all commute with one another since \( M_i \cdot M_j = M_j \cdot M_i = M_0 \cdot M_0 \cdot a \) for function variable \( a \).

4.2.3 Synthesis of the table-manager functions

We can synthesise the table-manager function (which will be used by the INSERT operation) by analysing the predicate transformers and finding their lowest common composition.

First we define the lowest common sum of the set of integers \( b_1, \ldots, b_k \) to be the lowest integer \( k \) s.t. for all \( j \leq k \), \( b_j = \min_{i=1,\ldots,k} b_i \) for some non-negative integers \( s_1, \ldots, s_k \). Now let \( lcs \) be the lowest common sum of the integers (depending on \( j \)) involved in the definition of the highest common generator given in the previous section. Then the lowest common composition of the p.t.s. is defined by \( LCC = M_{lcs} \cdot id \). We now have the following theorem, the proof being given in Ref. 10.

Theorem 3

Let \( f \) be defined by \( f = p \rightarrow q; \quad M(f, \ldots, f) \), where \( M \) is a degenerate \( n \)-multilinear form with p.t.s. having a highest common generator and lowest common composition \( LCC \). Then the s.g.c. memo-isation deletion strategy: ‘When inserting the result \( f(x) \), garbage collect the memo entry for the argument \( LCC \cdot x \)’ has the following properties

(i) For any depth-first evaluation mechanism the deletion strategy is safe in the sense that in the evaluation of any top-level application of the function \( f \), all results corresponding to identical arguments are only evaluated once.

(ii) For any application of \( f \), the total number of memo table entries at run time will never exceed \( lcs \).

The table-manager functions can now be simply generated as

\[
\text{lambda } x \Rightarrow [LCC \cdot x]
\]

4.3 Examples of table-managers

4.3.1 Fibonacci

The function \( fib \) defined in variable-free form by \( fib = \lambda x.\cdot \lambda y.\cdot (y \cdot (x \cdot y) \cdot +) \) is degenerate bilinear and has highest common generator \( M_n = M_0 \) as noted in Section 4.2.2. The lowest common composition of the p.t.s is \( LCC = sub \cdot 2 \), which implies that the function is s.g.c. memoisable and that its table manager is given by

\[
\text{lambda } x \Rightarrow [x \cdot 2]
\]

Theorem 3 guarantees that the s.g.c. memo-ised version of the function \( fib \) will execute in such a way that no identical call to \( fib \) will be evaluated more than once, and that the maximum size of the memo-table at run time for any top-level application of \( fib \) will be 2.

4.3.2 Comb

Consider the following function definitions

\[
\text{data } \text{combination} = \text{twos} \cdot 3 \cdot \text{s} \cdot 5 \cdot \text{s} \cdot (\text{num} \times \text{num} \times \text{num})
\]

\[
\text{dec } \text{add} : \text{num} \times \text{list} \cdot \text{combination} \rightarrow \text{list} \cdot \text{combination}
\]

\[
\text{add}(a, \text{nil}) = \text{nil}
\]

\[
\text{add}(2, \text{twos} \cdot 3 \cdot \text{s} \cdot 5 \cdot \text{s}(x, y, z) \cdot W) = \text{twos} \cdot 3 \cdot \text{s} \cdot 5 \cdot \text{s}(x + 1, y, z) \cdot \text{add}(2, W)
\]

\[
\text{add}(3, \text{twos} \cdot 3 \cdot \text{s} \cdot 5 \cdot \text{s}(x, y, z) \cdot W) = \text{twos} \cdot 3 \cdot \text{s} \cdot 5 \cdot \text{s}(x, y + 1, z) \cdot \text{add}(3, W)
\]

\[
\text{add}(5, \text{twos} \cdot 3 \cdot \text{s} \cdot 5 \cdot \text{s}(x, y, z) \cdot W) = \text{twos} \cdot 3 \cdot \text{s} \cdot 5 \cdot \text{s}(x, y, z + 1) \cdot \text{add}(5, W)
\]

\[
\text{dec } \text{g} : \text{num} \rightarrow \text{list} \cdot \text{combination}
\]

\[
g2 \rightarrow \text{twos} \cdot 3 \cdot \text{s} \cdot 5 \cdot \text{s}(1, 0, 0)
\]

\[
go3 \rightarrow \text{twos} \cdot 3 \cdot \text{s} \cdot 5 \cdot \text{s}(0, 1, 0)
\]

\[
go4 \rightarrow \text{twos} \cdot 3 \cdot \text{s} \cdot 5 \cdot \text{s}(2, 0, 0)
\]

\[
go5 \rightarrow \text{twos} \cdot 3 \cdot \text{s} \cdot 5 \cdot \text{s}(0, 0, 1, 2) \cdot \text{twos} \cdot 3 \cdot \text{s} \cdot 5 \cdot \text{s}(1, 0, 0)
\]

\[
\text{dec } \text{comb} : \text{num} \rightarrow \text{list} \cdot \text{combination}
\]

\[
\text{comb} \cdot n = \text{if } n \leq 5 \text{ then } \text{g} \cdot n
\]

\[
\text{else } \text{add}(5, \text{comb}(n - 5)) \cdot < * > \cdot \text{add}(3, \text{comb}(n - 3)) \cdot < * > \cdot \text{add}(2, \text{comb}(n - 2))
\]

where the infix function \(< * >\) appends any two lists together in such a way that no duplicate element will exist.
in the result. The function \( \text{comb}(n) \) returns all the combinations of numbers 2, 3 and 5 that when added will make up \( n \).

The function \( \text{comb} \) has a corresponding variable-free form of

\[
\text{comb} = l e 5 \rightarrow g ; < * > o \{ \text{addc} o \{ 5, \text{comb} o \text{sub} 5 \}, < * > o \{ \text{addc} o \{ 3, \text{comb} o \text{sub} 3 \}, \text{addc} o \{ 2, \text{comb} o \text{sub} 2 \} \}
\]

This definition is degenerate 3-multilinear and has the predicate transformers \( M_{a} a = a o \text{sub} 5 \), \( M_{a} a = a o \text{sub} 3 \) and \( M_{a} a = a o \text{sub} 2 \). The highest common generator of the p.t.s is \( M_{a} a = a o \text{sub} 5 \) and the lowest common composition of the p.t.s is \( \text{LCC} = \text{sub} 5 \), which implies that the function is s.g.c. memoisable and that its table manager is given by

\[
\text{lambda} \ x \Rightarrow [x - 5]
\]

The s.g.c. memoised version of the function \( \text{comb} \) will execute in such a way that no identical call to \( \text{comb} \) will be evaluated more than once, and the size of the memo-table at run time for any top-level application of \( \text{comb} \) will not exceed 5.

4.4 Generalisation for user-defined data types

S.g.c. memoisation is applicable to a class of multilinear functions whose predicate transformers have a common generator. However, one would expect that a large proportion of user-defined functions would operate on user-defined data types such as trees. Due to the arity of such data types the predicate transformers of a multilinear function will in general be incompatible, for example including the selectors RightTree and LeftTree. Thus s.g.c. memoisation will not be possible since the p.t.s involved fall into sets, and each p.t. is only compatible with those in its own set, and not with the ones in any other set.

Memo-isation is still possible if the function definition is degenerate multilinear, but instead of a single compatible set of p.t.s it has several such sets. This memo-isation is achieved by simply forming a bigger table-manager from each of the smaller ones corresponding to each set of compatible p.t.s. For example, \( n \) sets of compatible p.t.s within a degenerate multilinear function definition, and ensuring that the corresponding table-managers of \( \text{lambda} \ x \Rightarrow b_{1}, \ldots, \lambda \text{m a} \ x \Rightarrow b_{n} \), defined by Theorem 3. Then we will simply memoise the function with the following table-manager,

\[
\text{lambda} \ x \Rightarrow b_{1} < \ldots < b_{n}
\]

where \( < \ldots < \) denotes the infix operator \( \text{append} \). An example of this is given in Section 4.6. Note that although in general one would not expect to come across many such recursive user-defined functions, this generalisation plays an important role in the efficient implementation of functions that are synthesised by combining two or more user-defined functions into one single definition, also considered in Section 4.6. In order to handle user-defined data types in our variable-free framework, we need to translate HOPE-like functions into FP. This may be done as follows.

Constructors of a HOPE data type are given a unique code. An \( n \)-ary constructor object is then represented by a sequence of size \( n + 1 \) whose first element is the constructor code. For example, consider the HOPE data declaration statement

\[
\text{data} \ \text{Tree}(x) = = \text{Leaf}(x) + \text{Node}(\text{Tree}(x) \times \text{Tree}(x))
\]

where \( x \) is a type variable. \text{Leaf} and \text{Node} will be given the codes \( 0 \) and \( 1 \) respectively and an object of type \( \text{Tree}(\text{num}) \) such as \( \text{Node}(\text{Leaf}(8), \text{Leaf}(9)) \) will be represented by \( < 1, < 0, > , < 0, 9 > , > . \) In this way we can write equivalent FP function definitions by replacing the pattern-matching of HOPE by a conditional tree and then carrying out object abstraction on the entire definition. For example consider the function \( \text{depth} \),

\[
\begin{align*}
\text{depth}(\text{Leaf}(x)) &= 0 \\
\text{depth}(\text{Node}(1, r)) &= 1 + \text{max}(\text{depth}(1), \text{depth}(r))
\end{align*}
\]

which will have the equivalent FP form of

\[
\text{depth} = e q 0 \circ \text{hd} \rightarrow 0 ; + \circ [1, \text{max} \circ [\text{depth} o 2, \text{depth} o 3]]
\]

4.5 Source-to-source transformation for s.g.c. memo-functions

The implementation of memoisation will require primitives in the underlying machine such as \( \text{LOOKUP, INSERT and DELETE ENTRY} \). If it is not possible (or desired) to incorporate these primitives, we can generate a new function which is equivalent to the unmemoised one except that it takes and returns an extra argument which is the memo-table. For example, the function definition for s.g.c. memoisable function \( f \) of type \( \alpha \rightarrow \beta \), will be replaced by a new function \( f' \) of type \( (\text{MemTabType} \times \alpha) \rightarrow (\text{MemTabType} \times \beta) \), where \( \text{MemTabType} \) represents a memo-table of arguments and results for the function \( f \) for instance \( \text{list} \times (\alpha \times \beta) \). We now need to change the definition of the function \( f' \) to \( 2 o f' o [\text{EmptyTab}, \text{id}] \), where \( \text{EmptyTab} \) is a constant function which returns the empty memo-table; for example in case of a list implementation of memo-tables, \( \text{EmptyTab} \) will be \( \{ \} \). The generation of the new function and the change of the old definition to a qualified call of the new function can be carried out automatically using the information already obtained for the memoisation as follows.

Given an s.g.c. memoisable function \( f = p \rightarrow q; Hf \) where \( H \) is a degenerate \( n \)-multilinear form, we show the structure of the transformed function \( f' \) in HOPE to aid readability. The HOPE function definition for \( f(x) \) will be of the form

\[
\begin{align*}
\text{if BaseCondition}(x) \\
\text{then BaseValue}(x) \\
\text{else RecursivePart}(f(a_{1}), f(a_{2}), \ldots, f(a_{n}))
\end{align*}
\]

where \( a_{i} (1 \leq i \leq n) \) are HOPE expressions which form the arguments to the \( n \) function calls of \( f \) in the else part of the definition.

The generated definition for \( f' \) will then be:

\[
\begin{align*}
f'(\text{tab}, x) &= \\
\text{if BaseCondition}(x) \\
\text{then (tab, BaseValue(x))} \\
\text{else let (KnowIt, res) } &= = \text{LOOKUP(tab, x) in} \\
\text{if KnowIt} \\
\text{then (tab, res) } & \\
\text{else ReconstructedElsePart} \\
\text{where ReconstructedElsePart is the nested qualified expression}
\end{align*}
\]

\[
\text{let (tab}_{1}, A_{1}) = = f'(\text{tab}, a_{1}) \ \text{in}
\]

THE COMPUTER JOURNAL, VOL. 31, NO. 3, 1988 239
let \( \text{tab}_2, \text{A}_2 \) = \( f'(\text{tab}_1, \text{a}_2) \) in

: : : : : :

let \( \text{tab}_2, \text{A}_2 \) = \( f'(\text{tab}_{x-1}, \text{a}_2) \) in

let TheAnswer = RecursivePart(\( \text{A}_1, \text{A}_2, \ldots, \text{A}_n \) in

\( \text{INSERT}(\text{tab}_n, (x, \text{TheAnswer}), \text{Table-Manager-Function}), \text{TheAnswer} \);

and Table-Manager-Function is the function generated by the memoisation technique.

Similarly the HOPE definition of \( f \) will be replaced by

\[
(\text{let} \ (\text{FinalTab}, \text{Answer}) = = f'(\text{nil}, x) \text{ in} \text{Answer})
\]

which is equivalent to the FP expression \( 2 \sigma f' \sigma [.], \text{id] } \). As well as the templates above and the synthesised table-manager function, we require the following HOPE definitions to be pre-defined.

\[
\begin{align*}
\text{type} \ & \text{arg\_type} = = \alpha \\
\text{type} \ & \text{res\_type} = = \beta \\
\text{type} \ & \text{table} = = \text{list(\text{arg\_type} \times \text{res\_type})} \\
\text{type} \ & \text{manager} = = \text{arg\_type} \rightarrow \text{list(\text{arg\_type})}
\end{align*}
\]

\[
\text{dec LOOKUP:} \ \text{table} \times \text{arg\_type} \rightarrow \text{trual} \times \text{res\_type} \quad (*\text{returns true} \ \text{and the result if it finds an entry for the given argument} \ ; \text{returns false otherwise} *)
\]

\[
\text{dec DELETE\_ENTRY:} \ \text{table} \times \text{list(\text{arg\_type})} \rightarrow \text{table} \quad (*\text{finds entries of the arguments supplied, if they exist, and removes them from the table}*)
\]

\[
\text{dec INSERT:} \ \text{table} \times \text{list(\text{arg\_type})} \times \text{manager} \rightarrow \text{table}
\]

\[
\text{INSERT}(\text{tab}, (\text{arg}, \text{res}), \text{man}) = (\text{arg}, \text{res}) :: \text{DELETE\_ENTRY}(\text{tab}, \text{man}(\text{arg}))
\]

Note that the entire operation is automatic. The HOPE function definitions are initially analysed by looking at their equivalent FP definitions (as described previously), and if a function is found to be memoisable its associated table manager is synthesised. A simple HOPE reparser will then construct the new definition using the template given above and replaces all of the top-level applications of \( f \) by a HOPE qualified expression.

### 4.5.1 Source-to-source transformation in the fib example

The reconstructed HOPE definition for \( \text{fib} \) will be as follows,

\[
\text{fib}(x) = (\text{let} \ (\text{FinalTab}, \text{Answer}) = = f'(\text{nil}, x) \text{ in} \text{Answer})
\]

where the s.g.c. memo-ised version of \( \text{fib}, \text{fib}' \) is defined by

\[
\text{fib}'(\text{tab}, x) = \begin{cases} 
\text{if} \ x \leq 1 & \text{then} \ (1, \text{tab}) \\
\text{else let} \ (\text{KnowIt}, \text{res}) = = \text{LOOKUP}(\text{tab}, x) \text{ in} \\
& \text{if} \ \text{KnowIt} \ \text{then} \ (\text{tab}, \text{res}) \\
& \text{else let} \ (\text{tab}, \text{A}_1) = = \text{fib}'(\text{tab}, \ x-1) \text{ in} \\
& \text{let} \ (\text{tab}, \text{A}_2) = = \text{fib}'(\text{tab}, \ x-2) \text{ in} \\
& \text{let} \ \text{TheAnswer} = = \text{A}_1 + \text{A}_2 \text{ in} \\
& \text{INSERT}(\text{tab}_n, (x, \text{TheAnswer}), \lambda y \rightarrow [y-2]), \text{TheAnswer} \\
\end{cases}
\]

#### 4.6 Other uses of s.g.c. memo-functions

In general, a functional program is written as a set of mutually recursive function definitions rather than a single one of the type we have considered so far in this paper. To illustrate how we can apply memo-isation to optimise mutual recursion, consider the function deepest which, given a tree, returns the list of values stored in the leaves at the deepest part of the tree. This is defined in terms of the function depth as follows:

\[
\begin{align*}
\text{deepest}(\text{Leaf}(x)) &= [x] \\
\text{deepest}(\text{Node}(s, r)) &= \begin{cases} 
\text{if} \ \text{depth}(s) > \text{depth}(r) & \text{then} \ \text{deepest}(s) \\
\text{else if} \ \text{depth}(s) < \text{depth}(r) & \text{then} \ \text{deepest}(r) \\
\text{else} & \text{deepest}(s) < > \text{deepest}(r)
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{depth}(\text{Leaf}(x)) &= 0 \\
\text{depth}(\text{Node}(s, r)) &= 1 + \max(\text{depth}(s), \text{depth}(r))
\end{align*}
\]

More generally, the function pairs \( f \) and \( g \) can be defined as follows:

\[
\begin{align*}
f &= p_o \rightarrow q_o \ ; \ F(f, \ldots, f, g, \ldots, g) \\
g &= p_o \rightarrow q_o \ ; \ G(f, \ldots, f, g, \ldots, g)
\end{align*}
\]

where in general \( F \) and \( G \) may have any number of \( f \) variables and \( g \) variables including 0.

We cannot simply s.g.c. memo-ise \( f(\text{depth}) \) because although \( \text{depth} \) would then use at most 2 memo-table entries the actual pattern of re-using does not lie within calls of \( f(\text{depth}) \), but rather within separate top-level applications of \( f(\text{depth}) \) within \( g(\text{deepest}) \). We may use another variant of memoisation which fully memoises \( f(\text{depth}) \) only for the specific usage of calls to it from \( g \) (deepest). The memo-table of \( f(\text{depth}) \) is garbage-collected after the completion of the top-level application of \( g \) (deepest).

However, as pointed out in Ref. 10, such functions can often be further optimised by combining them into a single function definition, \( h \), where \( h \) is the construction of the two definitions. In other words \( h \) is defined by \( h = [f', g'] \), where \( f' \) and \( g' \) are the same as definitions of \( f \) and \( g \) in which all occurrences of \( f \) and \( g \) have been replaced by \( 1 \\text{o} \text{h} \) and \( 2 \\text{o} \text{h} \) respectively. In this way the new definitions for \( f \) and \( g \) are \( f = 1 \\text{o} \text{h} \) and \( g = 2 \\text{o} \text{h} \), where \( g \) is degenerate multilinear. Hence s.g.c. memoisation can be used to implement the function \( f \) efficiently. For example, for the combined definition of depth and deepest we will have a maximum table size of the depth of the tree to which it is applied, and the table manager, \( \lambda \ 
\]

\[
\begin{align*}
\lambda = [2; : x, 3; : x] \\
\text{The selector functions} 2 \text{ and } 3 \text{ will select the left and right trees respectively, and alternatively, in HOPE, we get}
\end{align*}
\]

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

Thus when the s.g.c. memo-ised version of the combined depth and deepest function is applied to a tree, the depth and deepest of branches are found, starting from the leaves. The values required for any internal node are computed from the LOOKUP results (i.e. for both depth and deepest) of its immediate children, achieving an efficient implementation by maintaining a table with an upper bound of the depth of the tree to which it is applied.
5. CONCLUSIONS

We have developed a number of program-transformation techniques based on a variable-free function-level analysis. First we presented a method that generates an object-level loop implementation for any function defined by a composite linear form which can be detected by only a slightly modified parser; one that detects simple linear forms, identifies their predicate transformers, $H$, and then constructs $E_n$ for a CLF $H$. The transformation system described would clearly be easy to implement in an FP compiler, but equally, all of the results apply to any functional programming language such as LISP, HOPE or ML; it is a relatively simple task to first abstract object variables.

We then illustrated a technique for linearisation of a class of non-linear functions through the use of a variant of memo-functions, called self-garbage-collecting memo-functions. As well as giving up the storage taken up by the memo-tables after the useful lifetime of the memo-function, self-garbage-collecting memo-functions garbage-collect (or re-use) entries when these are guaranteed no longer to be useful. In this way they ensure that the average size of the table is kept to a minimum. This is achieved by a synthesized function (a ‘table-manager’ function) which, upon inserting new results, deletes (or re-uses) entries that are obsolete. Furthermore, the size of the tables for self-garbage-collecting memo-functions are guaranteed not to exceed a compile-time constant found by a simple static analysis of the definition of the non-linear function. A generalised form of the technique can also handle functions which operate on user-defined data types.

Whilst providing a highly expressive formalism, functional programming languages lack the ability to use relations in several modes, as in the logic languages. This feature can be incorporated into a purely functional system to a great extent if the inverses of functions are available, and it is not difficult to implement this in the same way as in logic-based systems, i.e. by unification. However, unification tends to produce inefficient solutions to problems, including many for which there do exist solutions in the form of conventional recursive functions. Harrison and Khoshnevian use a similar function-level approach, introduced in this paper, to synthesise recursive inverse function definitions for certain first-order recursive functions. A second important application for inverse functions is in the transformation of abstract data types into concrete types which can be implemented more efficiently. This is based on the observation that an abstract function and its concrete version form a commutative square with the abstraction and concretisation functions mapping between types; the concretisation function is the inverse of the abstraction function.

We have seen that the algebraic approach to program transformation provides mechanisable optimisation for a number of classes of functional programs, all of the information required being obtainable during parsing. By way of comparison, the unfold/fold methodology too could derive, as a result of an appropriate sequence of the elementary steps, any functional equivalence obtained by the algebraic approach. However, how to choose the right ‘appropriate step’ without the benefit of hindsight is a substantial problem, and the choice would appear difficult, if not impossible, to automate with the generality required. For the present, the most effective answer would appear to be a compromise in which a type of transformation meta-system could be extended to include new higher-level steps, or tactics, to apply each algebraic equivalence between functions (or functions and loops). In this way, the increased automation of the algebraic approach could be acquired whilst still preserving the near-completeness of the unfold/fold methodology. Indeed, such an approach might even provide a means by which new tactics, and possible new functional equivalences, could be derived for use by the meta-system.

REFERENCES


Announcements

11–13 JULY 1988

BNCOD-6: Sixth British National Conference on Databases, University College Cardiff, in association with the British Computer Society. This is the sixth in the series of BNCOD conferences, which are primarily designed to provide a forum for British research work in the database field. The 1988 conference is being organised by University College Cardiff. Papers will be presented on any aspect of databases and database systems, and in particular new developments. Particular topics are Data models; Advanced user interfaces – use of 4GL; Knowledge bases and object-oriented-models; Expert database systems; Integrating vision and databases; Information retrieval and databases, Distributed databases – including gateways; Time representation in databases; Databases in office automation. Research papers and those based on user experience are welcome.

If you hope to attend the conference, contact: W. A. Gray, BNCOD-6, Department of Computing Mathematics, University College Cardiff, Cardiff CF2 4AG, Wales.

1–5 AUGUST 1988

ECAI-88, München

The European Conference on Artificial Intelligence 1988 is organised in cooperation by the European Coordinating Committee for Artificial Intelligence (ECAI), the Gesellschaft für Informatik e. V. (GI), and the Technische Universität München. The scientific meeting will take place on 2–4 August, the tutorials on 1, 2 and 5 August.

Submitted Papers

Papers will be presented on any of:

- Agent architecture and multi-agent interaction
- Architectures and languages
- Cognition (including knowledge acquisition)
- Demonstrations of academic AI-software
- Epistemology and social issues of AI
- Industrial applications
- Knowledge representation
- Logic programming
- Machine learning
- Natural language understanding
- Non-standard approaches (e.g. Model logics)
- Reasoning, theorem proving and their applications
- Robotics
- Vision

Invited speakers

Jerry De Jong (Urbana-Champaign), Chris Hogg (Imperial College), Tom Mitchell (CMU), Jean Rohner (Bull), Erik Sandewall (Linköping).

Tutorials

First day, introduction; second and third days, state of the art
- Expert systems: Jim Hunter and Jean-Gabriel Ganascia
- Deductive databases: Herve Gallaire and Jean-Marie Nicolas
- Intelligent tutoring: Ben du Boulay and Peter Ross
- Logic programming: David Warren and Keith Clark
- Machine learning: Yves Kodratoff and Katharine Morik
- Modal logics: Allan Ramsay and Ray Turner
- Theorem proving: Jean-Pierre Jouannaud and Jieh Hsiang
- Architectures of machines for AI: Jean-Paul Sansonnet and Jean-Paul Syre

For further information:

Professor Dr Bernd Radig, Technische Universität München (ECAI-88), Institut für Informatik, Postfach 20 24 20, D-8000 München 2, Federal Republic of Germany.

3–7 OCTOBER 1988

EUUG Autumn ’88 Conference, Lisbon, Portugal

New Directions for UNIX

The Autumn ’88 European UNIX systems User Group Technical Conference will be held in Lisbon, Portugal. Technical tutorials will be held on Monday 3 and Tuesday 4 October, followed by the three-day conference, ending on Friday 7 October. A pre-conference registration pack containing detailed information will be issued in early June 1988.

Subject areas will include:

- Real time
- Security issues
- Distributed processing
- Multi-processors and parallelism
- Supercomputing
- Internationalisation
- Fault tolerance
- Transaction processing
- Virtual memory
- Object-oriented approaches
- Videotext applications
- Standards and conformance tests

If you wish to receive a personal copy of any further information about this, and future EUUG events, please write, or send electronic mail, to the Secretariat.

Places in the main conference hotel are limited. In line with the suggestion made at Dublin, it will be possible to book for conferences well in advance. Therefore, there will be a special bookings desk for this conference available during the London meeting.

For further information please contact:

The Secretariat, EUUG, Owles Hall, Owles Lane, Buntingford, Herts, SG9 9PL, UK. Tel: (+44) 763 73039. Fax: (+44) 763 73255. Email: euug@inset.uuep.