Programming with Equations

CHRISTOPH M. HOFFMANN and MICHAEL J. O'DONNELL
Purdue University

Equations provide a convenient notation for defining many computations, for example, for programming language interpreters. This paper illustrates the usefulness of equational programs, describes the problems involved in implementing equational programs, and investigates practical solutions to those problems. The goal of the study is a system to automatically transform a set of equations into an efficient program which exactly implements the logical meaning of the equations. This logical meaning may be defined in terms of the traditional mathematical interpretation of equations, without using advanced computing concepts.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*nonprocedural languages*; D.3.4 [**Programming Languages**]: Processors—*interpreters*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Languages

Additional Key Words and Phrases: equations, term-rewriting systems

1. INTRODUCTION

Computer scientists have spent a large amount of research effort developing the semantics of programming languages. Although we understand how to implement ALGOL-style programming languages efficiently, it seems to be very difficult to say what the programs mean. The problem may come from choosing an implementation of a language before giving the semantics which define correctness of the implementation. We wish to reverse the process by taking clean, simple, intuitive semantics and then looking for correct, efficient implementations.

We suggest the following scenario as a good setting for the intuitive semantics of computation. Our scenario covers many but not all applications of computing (e.g., real-time applications are not included).

> A person is communicating with a machine. The person gives a sequence of assertions followed by a question. The machine responds with an answer or by never answering.

The problem of semantics is to define in a rigorous and understandable way what it means for the machine's response to be correct. A natural informal definition of correctness is that any answer which the machine gives must be a logical consequence of the person's assertions, and that failure to give an answer must mean that there is no answer which follows logically from the assertions. If the

This research was supported in part by National Science Foundation grant MCS 78-01812.
Authors' address: Department of Computer Sciences, Purdue University, West Lafayette, IN 47907.

language for giving assertions is capable of describing all the computable functions, the undecidability of the halting problem prevents the machine from always detecting those cases where there is no answer. In such cases the machine never halts. The style of semantics based on logical consequences seems most appropriate to descriptive or applicative languages such as LISP, Lucid, PROLOG, and the Red languages, although ALGOL-like languages may also be defined in such a way.

Semantics based on logical consequence is much simpler than many other styles of programming language semantics. For example, the popular style of denotational semantics based on work by Scott and Strachey requires that each program be associated with a unique function from input values to output values which is deemed to be the meaning of the program. But programs typically give only partial information about the objects in the programmer's mind which inspired him to write the program. As a result, the mathematical objects used as the meanings of programs in Scott–Strachey-style semantics are built within a fairly sophisticated kind of lattice theory and may bear little resemblance to the intuition. If a program is given as a set of assertions, then the logical consequences of the program are all those additional assertions which must be true whenever the assertions of the program are true. To be more precise, an equation A = B is a logical consequence of a set E of equations iff, in *every* algebraic interpretation for which every equation in E is true, A = B is also true (see [30, chap. 2] for a more technical treatment). There is no need to determine which one of the many models of the program assertions was really intended by the programmer; we simply compute for him all the information we possibly can from what we are given.

Theoretically, any logical language with a complete proof system could be used to describe programs. The PROLOG language [26] uses the first-order predicate calculus as a programming language. The implementation of PROLOG is based on backtracking, a potentially costly strategy. The implementations which have been published so far [9, 31, 36] violate the predicate calculus semantics in order to gain efficiency in some cases. We use the more restricted language of equational logic in order to achieve higher efficiency and avoid backtracking while satisfying exactly the logical semantics. Goguen [16, p. 500] claims that "any reasonable computational process can be specified purely equationally." Whether or not this thesis holds, equations are sufficiently powerful to describe a large number of important computations. Theoretically, equations may be used to program all of the computable functions. Section 3 gives several examples of practically useful equational programs.

We specialize our computing scenario to equational languages:

> The person gives a sequence of equations followed by a question, "What is E?" for some expression E. The machine responds with an equation "E = F," where F is a simple expression.

Section 2 discusses the correctness of equational computations and gives a technical definition of "simple expression."

Our goal is a useful system which accepts equations as input and automatically produces a program to perform the computations described by the equations. This goal is analogous to the goal of formal parsing research, which tries to

produce parsers automatically from context-free grammars. In order to achieve reasonable efficiency, we must accept some restrictions on the form of equations that may be given. Section 2 describes these restrictions.

Once such an equation processor is achieved, several natural applications follow:

(1) We may write programs as sets of equations.
(2) We may define programming languages by equations, and the equation processor will produce interpreters. For instance, the equations from [27], with some corrections, define LISP. An equation processor would automatically produce a LISP interpreter which exactly obeys those equations (existing LISP interpreters usually do not). References [15, 19] discuss some advantages of being faithful to the LISP equations.
(3) Equations describing data types may be used to automatically produce correct implementations as suggested by [18, 35].
(4) Theorems of the form A = B may sometimes be proved by receiving the same answer to the questions "What is A?" and "What is B?" Reference [25] discusses such theorem provers.

2. TECHNICAL PRELIMINARIES

The theoretical foundations for computing with equations are treated in detail in [30]. This section summarizes the crucial results. Equations containing variables are intended as an abbreviation for the infinite set of equations obtained by all possible substitutions of terms for variables. In some cases the substitutions are restricted to terms satisfying some syntactic condition, for example,

$$\text{mult(div(X, Y), Y)} = \text{X } \textbf{where } \text{Y is a numeral other than 0.}$$

Given an equational program D and a question "What is E_0?", a computer could merely enumerate proofs using D as a set of postulates until a theorem $E_0 = E_f$ appeared with E_f sufficiently simple. Such a strategy is unacceptably inefficient. In order to do better, we assume that each equation in D is ordered as

$$\text{left-hand side} = \text{right-hand side}$$

in such a way that the simple expressions which may be output are exactly those containing no instances of left-hand sides of equations.

In such cases, a better interpreter might produce a sequence $E_0, E_1, E_2, \ldots$ of expressions by repeatedly replacing instances of left-hand sides by the corresponding right-hand sides until an E_f containing no left-hand sides is found. Sequences produced by replacing left-hand sides of equations by right-hand sides are called reduction sequences and are studied in [7, 10, 13, 23, 30, 32–34]. An instance of a left-hand side is called a redex, and an expression with no redices is in normal form. The crucial theoretical problem in computing reduction sequences is to choose the next redex to replace. Bad choices might lead to an infinite sequence even though a normal form exists, causing a failure to produce a logically entailed answer. Worse, for some equation systems, there may be normal forms which cannot be obtained by a sequence of reductions. If multiple normal forms exist for the same input expression, we might worry about which one to choose.

Reduction sequences are not the only way to approach computation from equations. Nelson and Oppen [28, 29] use a congruence closure algorithm to find the consequences of a set of equations without actually performing replacements. Congruence closure was designed to test two given expressions for equality, rather than to reduce a single expression to a simpler form. Also, congruence closure handles only a finite set of equations without variables. In order to apply congruence closure ideas to programming, we need a way to identify normal forms within the algorithm and also a way to choose from a set of equations with variables the finite set of instances of those equations which are actually needed. Chew [12] has recently solved these problems at a theoretical level. The practical impact of the solutions has not yet been considered; so we only discuss reduction methods in this paper.

In order to achieve well-behaved reduction sequences, we need some reasonable additional restrictions on equations. These restrictions are significantly more liberal than those of [7, 10, 13] and are essentially the same as those of [23, 32].

1. No variable may be repeated on the left-hand side of an equation. For instance,

$$\textbf{if } X \textbf{ then } Y \textbf{ else } Y = Y$$

is prohibited.

2. If two different left-hand sides match the same expression, then the corresponding right-hand sides must be the same. So the pair of equations

$$g(0, X) = 0, \qquad g(X, 1) = 1$$

is prohibited, since g(0, 1) could be replaced by either 0 or 1.

3. When two (not necessarily different) left-hand sides match two different parts of the same expression, the two parts must not overlap. For example, the pair of equations

$$\text{first}(\text{pred}(X)) = \text{pred}, \qquad \text{pred}(\text{succ}(X)) = X$$

is prohibited, since the left-hand sides overlap in first(pred(succ(0))).

Reference [30] shows that, with these three restrictions,

(1) For each expression E_0 there is at most one normal form E_f, and E_f may be obtained by reducing E_0.
(2) Any strategy for choosing which redex to reduce that guarantees that each outermost redex in an expression is eventually reduced produces a normal form E_f whenever such a normal form exists.

We can use (2) above to prove that a machine is computing correctly from a set of equations.

In all cases satisfying restrictions 1–3, we can generate correct computations by reducing at each step all of the outermost redices in an expression. Such a strategy may be wasteful. For example, in an expression of the form

$$\textbf{if } A \textbf{ then } B \textbf{ else } C$$

(with the usual equations for conditional), we should reduce A to either T or F before investing any effort in B or C, one of which must be irrelevant. In many

cases the sequential nature of sets of equations allows a single redex to be chosen at each step so that no work is wasted. References [22, 23, 30, 34] treat the theoretical issues involved in sequentiality. Reference [30] claims an optimality result for some cases, but the proof has an error. Reference [23] has a slightly weaker, but correct, result that no unnecessary reductions are performed. Reference [22, sec. 6] presents, without proof, an algorithm for detecting and exploiting sequentiality. Certain systems of equations, such as those involving the parallel OR equations

$$\mathrm{OR}(\mathrm{T}, \mathrm{X}) = \mathrm{T}, \qquad \mathrm{OR}(\mathrm{X}, \mathrm{T}) = \mathrm{T}$$

cannot be made sequential in any reasonable sense, and optimality is practically impossible to achieve.

3. EXAMPLES OF EQUATIONAL PROGRAMS

With the examples developed in this section, we wish to give an intuitive feeling for the flavor of equational programming and illustrate the computational process entailed by equations. In Section 2 we claimed that a set of equations may specify a computation. There may be many different strategies for computing with the same set of equations. Using the approach of repeatedly replacing left-hand sides of equations by corresponding right-hand sides, the step-by-step behavior of a computation, as well as the result, is apparent from the equations. This transparency of the computation allows a user to think abstractly about the meanings of equations or more concretely about the derivations which they imply.

Our examples are drawn from several areas: list processing, abstract data types, search tree insertion, and interpreters of programming languages. In Section 4 we fix a precise notation for communicating an equational program to the machine.

3.1 List Processing

As a first example, consider list processing. Lists are objects defined as follows:

Definition 3.1

1. *nil* is the empty list.
2. If X is an atomic object or a list and Y is a list, then *cons*(X, Y) is also a list that contains X followed by everything in Y.

Lists are built using the single constructor cons and the special object nil over some fixed domain of atomic objects.

It is customary to abbreviate lists. For example, the list

$$\mathrm{cons}(\mathrm{A}, \mathrm{cons}(\mathrm{B}, \mathrm{cons}(\mathrm{C}, \mathrm{nil})))$$

is often written (A, B, C) [27]. For our purposes it is useful to visualize lists as trees of a certain shape. The list (A, B, C) is shown as a tree in Figure 1.

The *head* of a list is its first element; the *tail* is the list obtained by dropping the head from the list. The function *car* returns the head of a list, and the function *cdr* its tail. Head and tail are undefined for the empty list. The equational program is

$$\mathrm{car}(\mathrm{cons}(\mathrm{X}, \mathrm{Y})) = \mathrm{X}; \qquad \text{(L1)}$$
$$\mathrm{cdr}(\mathrm{cons}(\mathrm{X}, \mathrm{Y})) = \mathrm{Y}. \qquad \text{(L2)}$$

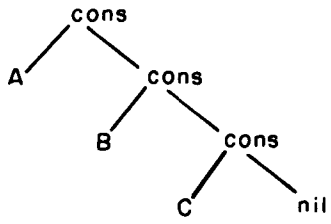

Figure 1

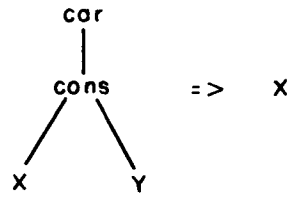

Figure 2

Recall that, by 2 of Definition 3.1, a nonempty list is of the form cons(X, Y). So, for example, by eq. (L1),

$$\text{car}(\text{cons}(\text{A}, \text{cons}(\text{B}, \text{cons}(\text{C}, \text{nil})))) = \text{A},$$

the head of the list (A, B, C). The computation entailed by eq. (L1) may be visualized as a tree transformation, as shown in Figure 2.

Two lists can be concatenated. Concatenating the empty list with any list Z should result in the list Z:

$$\text{concat}(\text{nil}, \text{Z}) = \text{Z}. \tag{L3}$$

Also, concatenating a nonempty list cons(X, Y), with head X and tail Y, with a list Z should result in a new list whose head is X and whose tail is the concatenation of Y and Z:

$$\text{concat}(\text{cons}(\text{X}, \text{Y}), \text{Z}) = \text{cons}(\text{X}, \text{concat}(\text{Y}, \text{Z})). \tag{L4}$$

In our scenario, a user wishing to concatenate a list (A, B, C) and a list (D, E) would first type, as assertions, eqs. (L3) and (L4) and then type the expression

$$\text{concat}(\text{cons}(\text{A}, \text{cons}(\text{B}, \text{cons}(\text{C}, \text{nil}))), \text{cons}(\text{D}, \text{cons}(\text{E}, \text{nil}))).$$

Note that the equational program can be used directly to specify the evaluation of the defined function. For example, by using eqs. (L3) and (L4), the machine would transform the expression as follows:

$$\begin{aligned}
&\text{concat}(\text{cons}(\text{A}, \text{cons}(\text{B}, \text{cons}(\text{C}, \text{nil}))), \text{cons}(\text{D}, \text{cons}(\text{E}, \text{nil})))\\
&\quad = \text{cons}(\text{A}, \text{concat}(\text{cons}(\text{B}, \text{cons}(\text{C}, \text{nil})), \text{cons}(\text{D}, \text{cons}(\text{E}, \text{nil}))))\\
&\quad = \text{cons}(\text{A}, \text{cons}(\text{B}, \text{concat}(\text{cons}(\text{C}, \text{nil}), \text{cons}(\text{D}, \text{cons}(\text{E}, \text{nil})))))\\
&\quad = \text{cons}(\text{A}, \text{cons}(\text{B}, \text{cons}(\text{C}, \text{concat}(\text{nil}, \text{cons}(\text{D}, \text{cons}(\text{E}, \text{nil}))))))\\
&\quad = \text{cons}(\text{A}, \text{cons}(\text{B}, \text{cons}(\text{C}, \text{cons}(\text{D}, \text{cons}(\text{E}, \text{nil}))))).
\end{aligned}$$

The first expression is input by the user. The subsequent expressions, except the last, are derived by applying eq. (L4), and the last line follows from the previous one using eq. (L3). Note that the last expression would be output as the final answer, since no further transformations are entailed by eqs. (L3) and (L4).

We observe that concatenation can be programmed by equations in a very natural and clear manner, and that the computation unfolds through a recursive use of the equations as term transformations.

Finally, consider the operation of reversing a list. The reversal of the empty list is again the empty list. Reversal of a list with head X and tail Y is a list resulting from concatenating the reversal of Y with the single-element list (X):

$$\text{reverse(nil)} = \text{nil}; \tag{L5}$$

$$\text{reverse(cons(X, Y))} = \text{concat(reverse(Y), cons(X, nil))}. \tag{L6}$$

It should now be apparent that equations express list processing operations in a very intuitive and clear way. The programming language LISP [27], which permits a very restricted form of equational programs, derives its popularity from this fact.

3.2 Abstract Data Types

Many authors have proposed the use of equations to specify abstract data types and operations on them [16, 18]. They argue that equations are an especially clear form for the presentation of abstract data types and allow for analysis by rigorous mathematical techniques.

For an example of such an equational specification, consider a type *set*, with *union*, *member*, and *set* operations. The operation "set" constructs a singleton set containing its argument; "member" tests set membership; and "union" is as expected. We assume further that set elements can be compared by a function *eq* for which we do not give equations here, since further assumptions about the nature of set elements would be required. T and F denote truth values, and the function *OR* of truth values is needed. The equations are now

$$\text{OR(T, X)} = \text{T};$$

$$\text{OR(X, T)} = \text{T};$$

$$\text{OR(F, F)} = \text{F};$$

$$\text{member(X, empty)} = \text{F};$$

$$\text{member(X, set(Y))} = \text{eq(X, Y)};$$

$$\text{member(X, union(Y, Z))} = \text{OR(member(X, Y), member(X, Z))}.$$

Note that the functions set and union are not specified by any equation. Thus they become constructors for abstractly representing sets. Sets thus could be visualized as binary trees whose leaves, the set elements, descend from the function symbol set.

It is straightforward to give equations for further operations such as intersection and set difference.

3.3 Search Tree Insertion

Our next example is search tree insertion. Specifically, we develop a program for insertion into *2–3 trees*, a special case of *B-trees*. We include this example to illustrate two important points:

1. Fairly large equational programs can be developed and understood with relative ease.
2. The 2–3 tree example illuminates the need for the nonoverlapping restriction on equations. Overlapping equations for 2–3 tree insertion cause interference

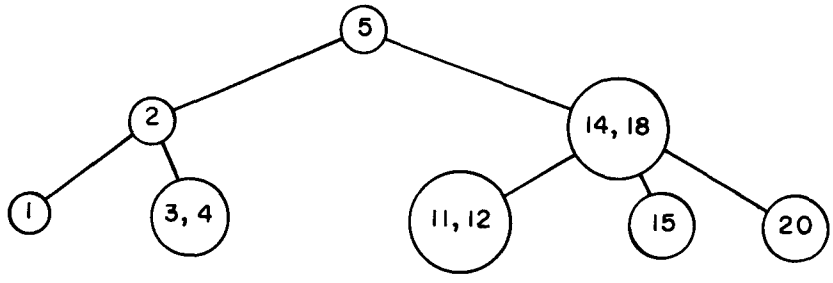

Figure 3

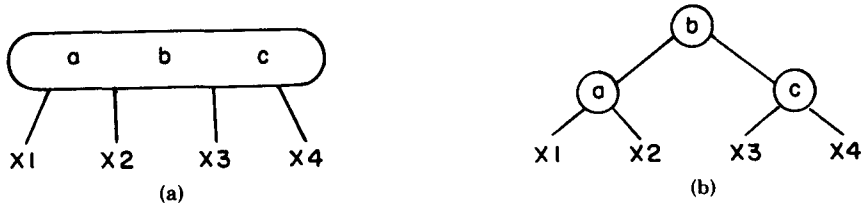

Figure 4

when several insertions are attempted in parallel. Nonoverlapping equations may be used freely in parallel. Thus, our syntactic restrictions on equations are not arbitrary but are connected with practical problems of parallel computation.

We believe that this example is an important application of equational programming and not just a toy problem.

Informally, a 2–3 tree is a data structure with the following properties. In a 2–3 tree, there are *2-nodes*, with two descending subtrees, and *3-nodes*, with three descending subtrees. A 2-node is labeled with a single key "a," such that the keys labeling nodes in the left subtree are all smaller than a and the keys labeling nodes in the right subtree are larger than a. A 3-node is labeled with two keys a and b, where $a < b$, such that the keys in the left subtree are all smaller than a, the keys in the middle subtree are greater than a and smaller than b, and the keys in the right subtree are larger than b. A *leaf* is a node whose subtrees are empty. A 2–3 tree is perfectly balanced; that is, the path lengths from the root to the leaves of the tree are all equal. Figure 3 shows an example of a 2–3 tree.

Because the tree is maintained in balanced shape, a key lookup operation requires at most $O(\log N)$ comparisons, where N is the number of keys stored in the tree. In contrast, a different shape of tree could sharply increase this bound.

When inserting a new key K into a 2–3 tree, there are two cases to consider. If the proper place for inserting K is a 2-node leaf, then we simply convert the leaf to a 3-node. If the insertion should be made into a 3-node leaf, then we must somehow restructure parts of the tree to make space for K. The restructuring proceeds as follows. First, form a 4-node, that is, a node with three keys a, b, and c, and four subtrees, as shown in Figure 4a. Of course, if we begin with a leaf, then the subtrees are all empty. Now split the 4-node into three 2-nodes as shown in

Figure 4b. The key of the middle 2-node must be inserted into the node that is the former parent of the 4-node, since, through the splitting, we have increased the number of descendants. If the parent node is a 2-node, then it now becomes a 3-node; otherwise, the splitting process is repeated on the parent level. If there is no parent, that is, if we have just split the tree root, then no further work is required. Note that without the insertion of the middle node into the parent we would have destroyed the balance of the tree.

First, we must choose a representation for 2-3 trees. Since there is a natural correspondence between expressions and trees, we denote a 2-3 tree by an expression

$$\text{tree(X)}$$

where X reflects the labeling and structure of the tree. A nonempty subtree with a 2-node root is written

$$\text{t2(X1, L1, X2)}$$

where X1 and X2 represent its left and right subtree and L1 is the label of the node. Similarly,

$$\text{t3(X1, L1, X2, L2, X3)}$$

denotes a 3-node with labels L1 and L2 and subtrees X1, X2, and X3. The constant e denotes the empty subtree.

Example 3.1 The 2-3 tree of Figure 3 is represented by

```
tree(t2(t2(t2(e, 1, e), 2, t3(e, 3, e, 4, e)), 5,
        t3(t3(e, 11, e, 12, e), 14,
           t2(e, 15, e),        18, t2(e, 20, e)))).
```

Now we program the insertion of a key K into a 2-3 tree tree(X). Insertion of a key K proceeds by first locating the leaf in which to insert K. For this purpose, we must compare K to the labels of a node. If the comparison detects equality, then K is already in the tree and the insertion is done. Otherwise, the result of the comparison determines a subtree into which K is inserted.

```
ins(K, tree(X)) = tree(ins(K, X)).                                   (B1)

ins(K, t2(X, L, Y))
    = if(K < L, t2(ins(K, X), L, Y),
      if(K > L, t2(X, L, ins(K, Y)),
                t2(X, L, Y))).                                       (B2)

ins(K, t3(X1, L1, X2, L2, X3))
    = if(OR(K = L1, K = L2), t3(X1, L1, X2, L2, X3),
      if(K < L1, t3(ins(K, X1), L1, X2, L2, X3),
      if(K < L2, t3(X1, L1, ins(K, X2), L2, X3),
                 t3(X1, L1, x2, L2, ins(K, X3))))).                  (B3)
```

Note that we have accounted for the possibility of K already occurring in the tree.

If K does not occur in the tree, then we eventually obtain a subexpression ins(K, e). At that point we have found the proper place of insertion. We handle the actual key insertion into nodes with the expression put(X, K, Y), which denotes the 2-node which has to be inserted into the parent node.

$$\text{ins(K, e)} = \text{put(e, K, e)}. \tag{B4}$$

The key insertion into a 2-node is simple:

$$\text{t2(put(X, K, Y), L1, X2)} = \text{t3(X, K, Y, L1, X2)}; \tag{B5}$$

$$\text{t2(X1, L1, put(X, K, Y))} = \text{t3(X1, L1, X, K, Y)}. \tag{B6}$$

There are two equations because we must account for the relative order of keys.

Node splitting, which is necessary in case we insert into a 3-node, requires three equations. A fourth equation accounts for having split the root node of the tree.

$$\begin{aligned}&\text{t3(put(X, K, Y), L1, X2, L2, X3)}\\ &\quad = \text{put(t2(X, K, Y), L1, t2(X2, L2, X3))};\end{aligned} \tag{B7}$$

$$\begin{aligned}&\text{t3(X1, L1, put(X, K, Y), L2, X3)}\\ &\quad = \text{put(t2(X1, L1, X), K, t2(Y, L2, X3))};\end{aligned} \tag{B8}$$

$$\begin{aligned}&\text{t3(X1, L1, X2, L2, put(X, K, Y))}\\ &\quad = \text{put(t2(X1, L1, X2), L2, t2(X, K, Y))};\end{aligned} \tag{B9}$$

$$\text{tree(put(X, K, Y))} = \text{tree(t2(X, K, Y))}. \tag{B10}$$

The ten equations (B1) through (B10) now constitute a program for inserting a key into a 2–3 tree. Although the equations are very intuitive, they do not obey the restrictions of Section 2. For example, eqs. (B2) and (B5) overlap in the expression

$$\text{tree(ins(3, t2(put(e, 2, e), 4, e)))}.$$

The problem arises conceptually because the described insertion proceeds in two phases: a traversal from the tree root to the insertion point, followed by a reverse traversal restructuring the nodes encountered up to the nearest 2-node (or up to the root if no 2-node is found). The overlap in the above expression corresponds to the competition between two insertions, one of which is in the restructuring phase, when eq. (B5) applies, while the other is in the initial traversal stage, when eq. (B2) applies. Since we may have several redices to choose from, we are dealing essentially with a parallel environment. Thus overlapping equations may indicate operations which may not be carried out in parallel.

The problem can be solved in the traditional manner of setting locks to prevent the progress of subsequent insertions where they may interfere with previous updates which are not completed. This is easily done by indicating a locked node as t2′(...) instead of t2(...), and t3′(...) instead of t3(...). The equations for this solution are given in Appendix A. Note that we force complete sequentiality of insertions, because the root is locked in this solution.

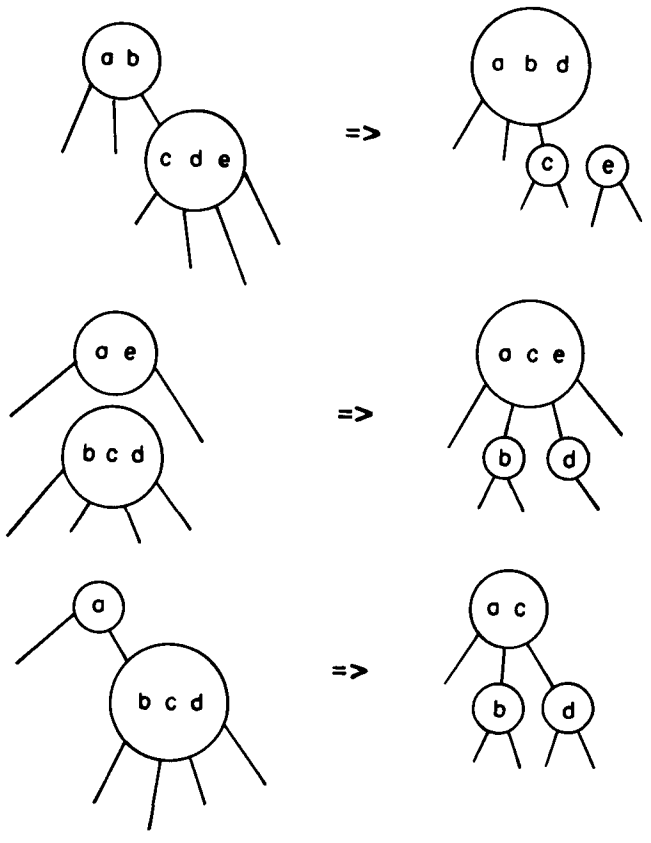

Figure 5

A different solution, which eliminates the need for locking the traversal path altogether, has been proposed in [17]. The trick is to split the nodes encountered on the downward traversal if they do not permit the insertion of another key without splitting. Since a 3-node at the root of a 2–3 tree cannot be split on the way down without destroying the tree's balance (there is no third key available), we must now deal with 2–3–4 trees, which also permit 4-nodes with three keys L1, L2, and L3, represented by

t4(X1, L1, X2, L2, X3, L3, X4).

The equational program now becomes a little more complex, since we have an additional node type. Nonetheless, 14 equations suffice. The transformations required to eliminate a 4-node as the point of insertion are given by Figure 5. Mirror-image cases have been omitted.

We give the equations based on the above representation in Appendix B. The elegant representation of binary trees, with nodes colored either red or black, proposed in [17] does not really simplify the equational program, since there still are the same number of basic transformations to account for. Note that with this program we may insert keys in parallel without interference problems.

3.4 Interpreters for Programming Languages

Last, we program three interpreters using equations. In [22] we explored this aspect of equational programming exclusively.

3.4.1 *A Small Procedural Language.* Consider an ALGOL-like language with a null statement, an assignment statement, **while** loops, compounding of statements, and statement sequencing. Programs contain the usual identifiers, which may be assigned simple integer and Boolean values. The Boolean values are **true** or **false**. We have a standard set of arithmetic and logical operators.

Because of the equational setting, programs in this language have to be transcribed into *program expressions*. Informally, the null statement is represented by the constant *empty*, sequencing of statements is expressed by a list of statements using as constructor a function *next*, **while** loops are written as expressions of the form

while(boolean expression, program expression),

and assignments are

set(identifier name, expression).

Statement compounding is implicit from the structure of the program expression. Note that **if** statements can be simulated by **while** statements. Identifiers have to be transcribed into nullary operations, which we uniformly denote by italicizing the identifier name. Thus, the identifier "A" in the program corresponds to the constant "*A*" in the corresponding program expression.

Example 3.2 The program fragment

```
while A > B do begin
  A := A − B;
  while B > A do begin
    T := A;
    A := B;
    B := T;
    end;
end;
```

would be transcribed into

```
next(while(gtr(A, B),
  next(set(A, minus(A, B)),
  next(while(gtr(B, A),
          next(set(T, A),
          next(set(A, B),
          next(set(B, T),
            empty))))),
        empty))),
```

According to our scenario, the machine is first presented the equations defining an interpreter for this language, and then it is given a program expression to be interpreted.

For simplicity, we assume that all data to be input are presented as part of the program expression, in the form of initial assignments, and that, at the end of "running" the program, all assigned values are printed. Because of assignment statements, identifier values are recorded in a store. For efficiency, the store could be organized as a search tree. Since we want to exclude unnecessary detail, we omit the equations for looking up an identifier value and the equations for updating (or newly inserting) an identifier name–value pair into the store.

The interpreter consists of three functions. The function

run(program expression, store)

interprets entire programs and is concerned with statement sequencing. The function

do(statement, program expression, store)

is charged with interpreting an individual statement, in the context of a store. The remaining program expression to be interpreted thereafter is the second parameter of this function. There is a function

value(expression, store)

which evaluates an expression (arithmetic or Boolean) using a store to find out identifier values. For the sake of brevity, we omit equations such as

value(plus(X, Y), T) = +(value(X, T), value(Y, T))

which make straightforward use of predefined functions, in this case +.

A program expression P to be interpreted is initially evaluated using an empty store:

interpret(P) = run(P, tree(e)). (A1)

Statements are done one by one in sequence. If none remains, then we should print the store from which the final identifier valuations are apparent:

run(next(S, P), T) = do(S, P, T); (A2)

run(empty, T) = T. (A3)

The empty statement has no effect:

do(empty, P, T) = run(P, T). (A4)

For an assignment, the appropriate identifier has to be updated with the expression value in the store:

do(set(V, E), P, T) = run(P, update(V, value(E, T), T)). (A5)

And, for a **while** loop, the controlling expression must be evaluated. If false, the body is ignored; if true, it is executed followed by the same **while** loop (which, in turn, is followed by the remaining program).

```
do(while(B, P), P1, T)
    = if(value(B, T),
          run(next(P, next(while(B, P), P1)), T), run(P1, T)).   (A6)
```

Finally, because of compounding, the first argument of the do function is possibly a program expression:

$$\text{do(next(S, P), P1, T)} = \text{run(P1, run(next(S, P), T))}. \qquad \text{(A7)}$$

Note that, after interpreting the program expression "next(S, P)," the updated store T1 is returned as the correct store for interpreting the remaining program expression P1.

Expressions are evaluated by the function *value*(E, T). The value function is computed by equations of the form

$$\text{value(op(E1, E2), T)} = \text{systemop(value(E1, T), value(E2, T))}$$

and of the form

$$\text{value(V, T)} = \text{lookup(V, T)} \textbf{ where } \text{V is an identifier name}$$
$$\text{value(C, T)} = \text{C} \textbf{ where } \text{C is a constant.}$$

These equations are straightforward and have been omitted. Their precise form is clear from the syntax specifications in Section 4.

The two main operations on the store are *lookup* and *update*. They are programmed by variants of the search tree equations developed in Section 3.3. The variation comes from the fact that the store has to maintain name-value pairs as node labels, with the name serving as the key for lookups. Note that only the function "update" may result in new leaves being added to the store. Lookup of a nonexistent identifier name would eventually result in a subexpression lookup(V, e) occurring in the store. It is not necessary to add an equation for this case. That way, the presence of the subexpression effectively blocks further computation depending on its value. In such a case, the resulting output is a partially interpreted program expression, along with a store containing subexpressions which inform exactly which identifier(s) were referenced before first assigning them. Because of the notational choices, this constitutes a meaningful error message.

3.4.2 *Pure LISP.* Next, we sketch the development of an equational program for a LISP interpreter. Our equations are very similar to the ones of [27]. Employing the function *apply*, however, enables us to do static scoping instead of the dynamic scoping implicit in McCarthy's equations. Static scoping agrees with the lambda-calculus origins of LISP. Implementers of LISP have usually viewed McCarthy's dynamic scoping as an error and have achieved static scoping by the FUNARG device.

Since LISP has been around for many years, it may seem pointless to provide yet another LISP interpreter. There are several reasons why LISP is an important example:

1. Past implementations of LISP have required special programming efforts, often in assembly language. When we gain the ability to compute directly from equations, it becomes possible to experiment with a wide variety of descriptive programming languages merely by modifying the equations specifying such languages. This ability is analogous to the ability of a compiler writer to experi-

ment with different syntaxes by modifying a context-free grammar, rather than by rewriting a hand-coded parser.

2. Ad hoc approaches to the implementation of languages such as LISP have raised serious questions about the exact relationship of the resulting interpreters to the original language specifications. LISP interpreters are generally not exact realizations of McCarthy's equations, and it is often difficult to discover exactly what they do. An interpreter that is generated automatically from equations is guaranteed to be exactly faithful to those equations, so that we always have an accurate specification of the language. In particular, a LISP interpreter generated by our methods must have the behavior of the lazy interpreters of [15, 19, 30], unless the equations for car and cdr are specifically restricted to hold only when both the left and right parts of an S-expression have been completely evaluated.

3. Although the main advantage of automatic interpreter generation from equations is the ability to work with specifications rather than code, we believe that a carefully polished equational interpreter can compete in efficiency with hand-coded LISP interpreters. We have used an experimental equation processor to generate a LISP interpreter; the results are reported in Section 6.

The LISP language is based on S-expressions, built from an unlimited supply of zeroary atomic symbols, including NIL, and a single binary symbol cons. The familiar parenthesis notation $(A_1\ A_2\ \cdots\ A_n)$ is merely an abbreviation for $\text{cons}(A_1, \text{cons}(A_2, \cdots \text{cons}(A_n, \text{NIL}) \cdots))$. Additional function symbols, such as car, cdr, cond, and eval, are used to build M-expressions, a superset of the S-expressions. Inputs to a LISP interpreter are M-expressions of the form $\text{eval}(S_p, \text{NIL})$ where S_p is an S-expression (LISP program) to be evaluated and the second argument provides an initially empty environment for the evaluation. Actual LISP interpreters leave the eval implicit and accept S_p alone. Outputs from a LISP interpreter are simply S-expressions.

We need to program the functions car, cdr, and cond (conditional). The equations for car and cdr are as in Section 3.1. The equations for cond are

$$\text{cond}(\text{T}, \text{X}, \text{Y}) = \text{X};$$
$$\text{cond}(\text{F}, \text{X}, \text{Y}) = \text{Y}.$$

Furthermore, we need a function *pair* in charge of pairing two lists, element by element, used to match actual and formal parameters in function calls. For efficiency, we have combined pair and append, saving an additional pass over the parameter list. We call the combined pair and append function *extend*.

$$\text{extend}(\text{NIL}, \text{NIL}, \text{E}) = \text{E};$$
$$\text{extend}(\text{cons}(\text{V}, \text{W}), \text{cons}(\text{X}, \text{Y}), \text{E}) = \text{cons}(\text{cons}(\text{V}, \text{cons}(\text{X}, \text{NIL})), \text{extend}(\text{W}, \text{Y}, \text{E})).$$

Variable substitutions are recorded in an environment, traditionally a list of variable–value pairs. As in [27], the pair containing X and Y is implemented as list(X, Y), that is, cons(X, cons(Y, NIL)). The paired parameter list is used to extend the environment during the evaluation of the function body. Looking up a variable binding is done by *assoc*, essentially a sequential lookup of the

environment list:

$$\text{assoc}(X, Y) = \text{cond}(\text{eq}(\text{car}(\text{car}(Y)), X), \text{car}(\text{cdr}(\text{car}(Y))), \text{assoc}(X, \text{cdr}(Y))).$$

The function *eq* tests equality of atomic symbols and is predefined. There is a function *atom* which tests whether its argument is an atomic symbol. Its equational form is deferred until Section 4, where we explain the conventions necessary to express when a function symbol is atomic.

The interpreter has to reduce expressions of the form

$$\text{eval}(\text{S-expression}, \text{environment}).$$

The evaluation of an S-expression which is a variable is just a lookup of its bound value in the environment.

$$\text{eval}(X, Z) = \text{assoc}(X, Z) \textbf{ where } X \text{ is an atomic symbol.}$$

All other S-expressions as the first argument of eval are of the form

$$(\text{operator}, \text{operandlist})$$

and are evaluated by first evaluating the operator using a nested eval and then applying it to the argument list. Note that, because of QUOTE, we must delay the evaluation of the argument list.

$$\text{eval}(\text{cons}(X, Y), Z) = \text{apply}(\text{eval}(X, Z), Y, Z).$$

Application is much as in [27]. For example,

$$\text{apply}(\text{eval}(\text{CAR}, Z), \text{cons}(X, Y), Z1) = \text{car}(\text{eval}(X, Z1)),$$

where we must evaluate the argument X in its environment, and

$$\text{apply}(\text{eval}(\text{QUOTE}, Z), \text{cons}(X, Y), Z1) = X,$$

where no evaluation of X takes place because of QUOTE. Function definitions are applied by

$$\text{apply}(\text{eval}(\text{cons}(\text{LAMBDA}, \text{cons}(V, \text{cons}(W, X))), Z), Y, Z1) = \text{eval}(W, \text{append}(\text{pair}(V, \text{evlis}(Y, Z1)), Z)).$$

V is the list of formal parameters, which are to be bound to the evaluated actual parameters Y. Z1 is the calling environment. So the function body W must be evaluated in an environment Z, resulting from Z1 by evaluation of the operator part of the LAMBDA expression, which is augmented by the argument pairing. A complete set of equations for LISP can be found in Section 4, along with all the paraphernalia needed to transform the equations into a LISP interpreter.

3.4.3 *Lucid.* Lucid is a programming language designed by Ashcroft and Wadge [2, 3] to mimic procedural computation with nonprocedural semantics. The primitive operations involved are quite different in character from the LISP primitives, and the first implementers of interpreters [11] and compilers [20] encountered serious difficulties.

The basic objects in Lucid are intended to be infinite sequences representing the histories of all values computed by a program. Traditional operations, such

as OR and +, operate componentwise on sequences. The additional operations first, next, asa (as soon as), latest, latestinv, and fby (followed by) are used to manipulate the structure of sequences.

The following set of equations, adapted from [30], could be used to produce a Lucid interpreter directly.

```
~(T) = F
~(F) = T
~(fby(W, X)) = fby(~(first(W), ~(X))
~(latest(X)) = latest(~(X))

OR(T, X) = T
OR(X, T) = T
OR(F, F) = F
OR(fby(W, X), fby(Y, Z))
    = fby(OR(first(W), first(Y)), OR(X, Z))
OR(fby(W, X), latest(Y))
    = fby(OR(first(W), latest(Y)), OR(X, latest(Y)))
OR(latest(X), fby(Y, Z))
    = fby(OR(latest(X), first(Y)), OR(latest(X), Z))
OR(fby(W, X), F) = fby(W, X)
OR(F, fby(Y, Z)) = fby(Y, Z)
OR(latest(X), latest(Y)) = latest(OR(X, Y))
OR(latest(X), F) = latest(X)
OR(F, latest(Y)) = latest(Y)

if(T, Y, Z) = Y
if(F, Y, Z) = Z
if(fby(W, X), Y, Z) = fby(if(first(W), Y, Z), if(X, Y, Z))
if(latest(X), Y, Z) = latest(if(X, Y, Z))

first(X) = X where X in {T, F} ∪ integer
first(fby(X, Y)) = first(X)
first(latest(X)) = latest(X)

next(X) = X where X in {T, F} ∪ integer
next(fby(X, Y)) = Y
next(latest(X)) = latest(X)

asa(X, Y) = if(first(Y), first(X), asa(next(X), next(Y)))

latestinv(X) = X where X in {T, F} ∪ integer
latestinv(fby(X, Y)) = latestinv(X)
latestinv(latest(X)) = X

+(fby(W, X), fby(Y, Z))
    = fby(+(first(W), first(Y)), +(X, Z))
+(fby(W, X), latest(Y))
    = fby(+(first(W), latest(Y)), +(X, latest(Y)))
+(latest(X), fby(Y, Z))
    = fby(+(latest(X), first(Y)), +(latest(X), Z))
+(fby(W, X), Y) = fby(+(first(W), Y), +(X, Y)) where Y in integer
+(X, fby(Y, Z)) = fby(+(X, first(Y)), +(X, Z)) where X in integer
+(latest(X), latest(Y)) = latest(+(X, Y))
+(latest(X), Y) = latest(+(X, Y)) where Y in integer
+(X, latest(Y)) = latest(+(X, Y)) where X in integer
```

The equations shown for addition extend the addition operation to act componentwise on complex structures. The usual equations for adding integer constants, such as +(2, 3) = 5, are included implicitly. Any other standard binary

operations on integers could be added just by repeating the above equations for addition with the new operation symbol instead of the symbol +.

A Lucid program is itself a set of equations, with a special symbol OUTPUT representing the result of the computation. To interpret a Lucid program, we would merely concatenate the program with the equations above, then reduce the expression OUTPUT to normal form.

4. AN EQUATIONAL PROGRAMMING LANGUAGE

Having illustrated the kinds of equational programs we wish to write, we now give a specific syntax for equations and input expressions to be simplified. With this syntax, we can fully implement our computing scenario of Section 1. It is not our aim to suggest the neatest form of syntax we can invent. Rather, we wish to identify the essential information which has to be communicated to the machine, and to provide a means for doing so which has the degree of convenience necessary to make it a practically useful programming language.

We need to define for the machine the set of allowable input expressions and give the equational program. To specify the input expressions, it suffices to list all meaningful function symbols along with the number of arguments they require, that is, their arities. Often, large sets of standard symbols need to be included, for example, integers, and we provide standard sets of symbols which may be specified by a single name. For our purposes here we use the set **integer** of all integer constants, the set **boolean** of the truth constants T and F, and the set **unspecified** containing, as zeroary symbols, all alphanumeric strings not otherwise accounted for. These sets are called *primitive domains.*

It is also useful to allow the specification of certain function symbols, called *standard functions,* which have predefined meanings. These are the operation symbols +, −, *, *div,* and *mod* and the relational symbols *eq, ne, lt, gt, le,* and *ge.* These symbols have meaning on the primitive domain **integer**, and the relational symbols are also defined on the domain **unspecified.**

The set of all allowed input expressions is then the set of terms composed in the usual way from the specified symbols.

From a theoretical point of view, it is sufficient to give all equations directly, for example,

$$\text{car}(\text{cons}(X, Y)) = X.$$

For practical purposes, however, standard functions, such as +, defined on primitive domains should be computed by a program. Thus the set of equations containing

$$+(0, 0) = 0$$
$$+(0, 1) = 1$$
$$+(1, 0) = 1$$
$$+(1, 1) = 2$$
$$\text{etc.}$$

is implicitly specified by a program which, given that a subexpression "+(X, Y)" where X and Y are integer constants is to be reduced, replaces it with an integer Z such that +(X, Y) = Z. In order to fit these pragmatics into the framework of

term rewriting, one may wish to visualize the effect of these programs as specified by schemata such as

+(X, Y) = Z **where** X, Y, Z in **integer** and Z is the sum of X and Y.

The predicate "Z is the sum of X and Y" is realized by a subroutine, whereas the recognition of an instance of "+(X, Y) **where** X, Y in **integer**" is done by the matching mechanism which also implements directly given equations. This view would suggest the possibility of allowing the user to associate subroutines written in some other programming language with equation schemata in order to implement more complicated predicates. This would be in analogy to the way in which semantic routines are associated with the reduction steps of a parser as prescribed by a context-free grammar. We wish to limit our device, however, to standard functions alone in order to ensure simple semantics and to avoid tedious separate correctness proofs of user-supplied subroutines.

Which ones of the predefined operation symbols should be included is communicated by

Primitives ⟨*list of operation symbols whose schemata should be included*⟩.

Sometimes a large number of equations must mention each expression in a large class. For example, in LISP the equations for the function *atom* would have to include a separate equation

atom(C) = T

for each integer constant and for each unspecified symbol C. Such sets of equations are condensed by using a variable restricted to range over a union of primitive domains and explicitly listed sets of zeroary symbols. Thus, we could program the function "atom" of LISP by the following equations:

atom(cons(X, Y)) = F;
atom(X) = T **where** X in **integer** ∪ **boolean** ∪ **unspecified**
∪ {CONS, CAR, CDR, ATOM, EQ, QUOTE,
LABEL, LAMBDA, COND, NIL}.

Equations are then presented using the notational conventions above by writing them out and prefixing the set with the clause

For All ⟨*list of variable names*⟩.

Example 4.1 To illustrate the form of our programming language, we give below the equational program for a LISP interpreter similar to that of [27]:

Symbols
QUOTE, LABEL, LAMBDA, ATOM, COND, CONS, CAR, CDR, EQ,
NIL: 0; **boolean**; **unspecified**;
cond, apply: 3; cons, eval, evlis, append, pair,
assoc, evcon: 2;
atom, car, cdr: 1.
Primitives
eq.

```
For All V, W, X, Y, Z, Z1:
  car(cons(X, Y)) = X;
  cdr(cons(X, Y)) = Y;
  cond(T, X, Y) = X;
  cond(F, X, Y) = Y;
  atom(cons(X, Y)) = F;
  atom(X) = T where X in boolean ∪ unspecified
                         ∪ {QUOTE, EQ, LABEL, LAMBDA, NIL, ATOM, COND,
                            CONS, CAR, CDR, NIL};
  eval(X, Z) = assoc(X, Z) where X in unspecified;
  eval(cons(X, Y), Z) = apply(eval(X, Z), Y, Z);
  apply(eval(COND, Z), X, Z1) = evcon(evlis(X, Z1));
  apply(eval(CAR, Z), cons(X, Y), Z1) = car(eval(X, Z1));
  apply(eval(CDR, Z), cons(X, Y), Z1) = cdr(eval(X, Z1));
  apply(eval(ATOM, Z), cons(X, Y), Z1) = atom(eval(X, Z1));
  apply(eval(QUOTE, Z), cons(X, Y), Z1) = X;
  apply(eval(EQ, Z), cons(W, cons(X, Y)), Z1)
      = eq(eval(W, Z1), eval(X, Z1));
  apply(eval(CONS, Z), cons(W, cons(X, Y)), Z1)
      = cons(eval(W, Z1), eval(X, Z1));
  apply(apply(eval(LABEL, Z), cons(V, cons(W, X)), Z1), Y, Z2)
      = apply(eval(W, cons(cons(V, cons(cons(LABEL,
            cons(V, cons(W, X))), NIL)), Z)), Y, Z1);
  apply(apply(eval(LAMBDA, Z), cons(V, cons(W, X)), Z1), Y, Z2)
      = eval(W, append(pair(V, evlis(Y, Z1)), Z));
  evlis(NIL, Z) = NIL;
  evlis(cons(X, Y), Z) = cons(eval(X, Z), evlis(Y, Z));
  evcon(cons(cons(T, cons(X, Y)), Z)) = X;
  evcon(cons(cons(F, cons(X, Y)), Z)) = evcon(Y);
  append(NIL, Y) = Y;
  append(cons(W, X), Y) = cons(W, append(X, Y));
  pair(NIL, NIL) = NIL;
  pair(cons(V, W), cons(X, Y)) = cons(cons(V, cons(X, NIL)), pair(W, Y));
  assoc(X, Y) = cond(eq(car(car(Y)), X),
                     car(cdr(car(Y))),
                     assoc(X, cdr(Y))).
```

From the point of view of human engineering, a graphical programming language in which, for example, expressions are drawn as trees would be very nice. This would be especially useful to those who, like us, have occasionally unbalanced parentheses. We find trees representing expressions easier to comprehend than prefix notation. Besides the problem of parsing such drawings, the same information has to be given as is given by the programming language defined above.

Another possibility is to present expressions in a syntactic guise. For example, we could write

if T **then** X **else** Y = X

instead of

if(T, X, Y) = X.

This is easily accomplished by employing known parsing techniques; see, for example, [1].

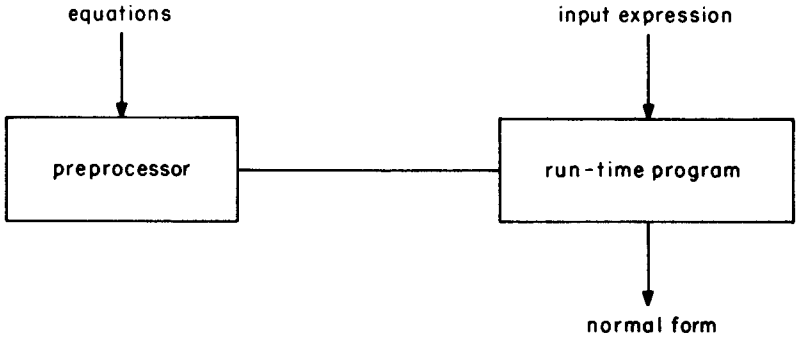

Figure 6

Finally, we might wish to impose a type structure on expressions, so that certain obviously meaningless expressions would be disallowed. Addition of type checking to the initial parsing of expressions and/or to the computation of reduction sequences is straightforward and involves no new techniques. We do not pursue these issues here, since they do not involve questions essential for implementing our computing scenario.

5. IMPLEMENTATION

In this section we describe how the equational programming language we have defined in Section 4 can be implemented. Specifically, we describe an implementation in which the equations are used as reduction rules. A related implementation approach has been taken by Bjorner's interpreter [8] for Backus' Red languages [5]. Bjorner's interpreter implements reduction rules which are simpler than our rules in that left-hand sides are always marked with special metasymbols (as indicated in private correspondence with Bjorner). Thus, there is no need for a sophisticated pattern matcher. Furthermore, Backus intends that an equation $f(X, Y, Z, \ldots) = E$ should hold only when all the arguments to f can be completely reduced. Thus, Red language interpreters do innermost replacements first, while we do outermost replacements in order to satisfy equations even when certain subexpressions of the left-hand sides have no normal form.

The key technical problems in implementing equational computations as reduction sequences are

(1) representing expressions,
(2) finding redices,
(3) choosing which redex to reduce at each step,
(4) performing a single reduction.

Since the same set of equations is likely to be used repeatedly with different input expressions, it is worthwhile to divide the implementation into a preprocessor, which does as much work as possible on the equations before seeing an input expression, and a run-time program, which performs the actual reductions of expressions to normal form. Figure 6 illustrates this organization.

The solutions to problems (2) and (4) affect both the preprocessor and the run-time program, while those to (1) and (3) affect only the run-time program.

5.1 Representing Expressions

Expressions are normally implemented as trees, using pointers to indicate the parent–child relationship. Figure 7a shows the expression

$$h(f(g(a, b), g(a, b)), g(a, b))$$

represented by pointers in the usual way. By allowing equivalent subexpressions (such as the three instances of g(a, b) above) to be shared in a directed acyclic graph, we save space. We save time as well if reductions are made in the shared subexpression. These reductions would be applied once only, instead of being applied to each of the many possible copies of the subexpression. Figure 7b shows the same expression represented with sharing.

For some sets of equations we know how to solve problems (2)–(4) with the shared representations of expressions given above. For our most general redex finding methods we need pointers from child to parent as well as parent to child. So we link each child with all its parents (with sharing there could be many) in a circular list. At most, this representation doubles the space used for pointers, but with sharing the increase in space may be much less. Figure 7c shows

$$h(f(g(a, b), g(a, b)), g(a, b))$$

again, with the g(a, b) subexpressions shared, represented with circular parent–child pointers.

5.2 Finding Redices

The problem of finding redices in an expression is a critical one for efficient implementation of equational computations. Essentially, the problem is one of pattern matching in trees, with two special considerations. First, the patterns are the left-hand sides of equations, which are known in advance by the preprocessor, while the subject trees are the expressions to be reduced, which are known only at run time. So, we want pattern matching techniques which do as much work as possible with the patterns before seeing a subject. Second, each time a redex is replaced, the subject tree changes slightly. Our pattern matching technique should adapt incrementally to such local changes rather than rescanning the whole subject tree.

The problem of tree pattern matching is treated in detail in [21]. We summarize here Algorithm M of that paper, which seems to be the best available method for most equational computations.

We associate with each node p of an expression a *match state*, which codes the set of all patterns which match the subtree rooted at p. The match state for p can be assigned from precomputed tables which are indexed by the symbol at p and the match states of p's children.

Initially, upon reading the expression E_0 to be reduced to normal form, the nodes representing E_0 can be assigned proper match states in a preconditioning step. Note that the match states now identify all redices in E_0. A replacement step changes the expression locally, and we now discuss how the match state assignment is affected.

When a redex occurrence r is replaced with its corresponding right-hand side r′, the nodes representing the function symbols of r′ are new, and their match

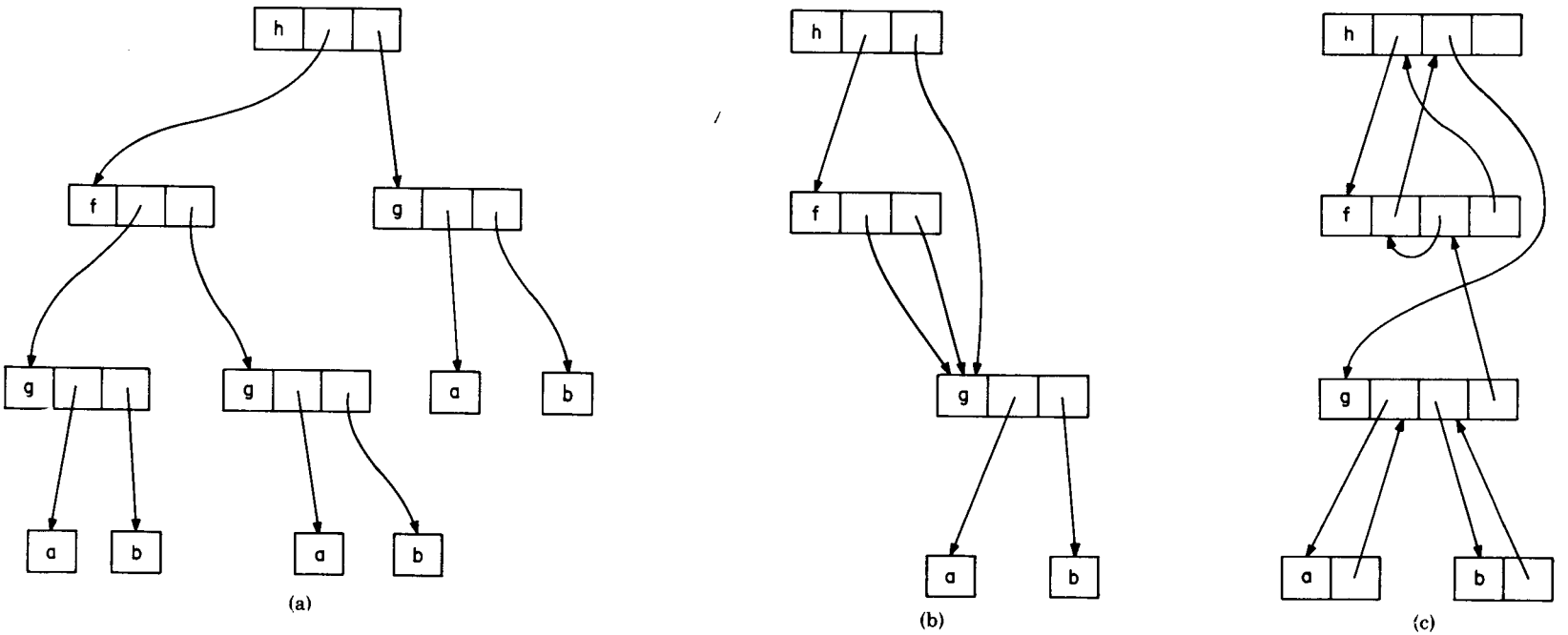

Figure 7

states have to be computed. Note that these nodes are either (1) leaves (thus the match state can be computed immediately from the function symbol) or (2) interior nodes whose children are either other nodes labeled with function symbols of r′ or roots of subtrees instantiating the variable symbols of r′. Since the match state assignments of the latter subtree nodes cannot be invalid—those subtrees have not changed in any way—we can therefore compute the match states of the new tree nodes proceeding from the leaves of r′ to the root of r′.

Now observe that, because of the sets represented by match states, the only nodes whose match states may have become invalid (through the replacement step) are nodes on the path(s) from the root of r′ to the root of the whole expression. We find the nodes on these ancestor paths using the backpointers. The algorithm therefore traverses the ancestor paths, breadth first, recomputing the match state assignments. Note that a path does not have to be traversed farther upward when the recomputed match state equals the previously assigned one. Note also that, because of the sets represented, equality must occur after traversing an ancestor path beyond the length of a longest path in any of the patterns. Thus the update process is confined to a local area in the expression graph.

5.3 Choosing Redices and Performing Reductions

In sequential systems, we may choose the next redex to replace by searching an expression in the appropriate sequential order and choosing the first redex encountered. For example, leftmost-outermost reductions may be found by a depth-first search from left to right: a preorder traversal. Reference [22, app. C] outlines an algorithm for detecting and exploiting sequentiality.

In inherently parallel systems of equations we keep a queue of all redices. At each step the first redex in the queue is reduced, and any newly created redices are added to the end of the queue. Some redices in the queue might be removed from the expression before reaching the head. By keeping reference counts we may detect and eliminate such redices. This queuing strategy may waste steps reducing redices which are not outermost. We have not yet found a way to keep track of only the outermost redices efficiently enough to justify such a technique.

Having chosen a redex to reduce, performing the reduction is straightforward. The preprocessor provides for each equation a pointer representation of the right-hand side and a list of the positions of variables. Another copy of this representation is made and linked into the expression in place of the chosen redex. Variable positions are replaced by pointers to the appropriate expressions. If a variable occurs more than once in a right-hand side, sharing occurs.

6. FIRST IMPLEMENTATION EXPERIENCE

We have implemented the equational programming language with the syntax described in Section 4. This implementation was written by Giovanni Sacco in PASCAL and contains 2700 lines of code for the preprocessor and 1200 lines for the run-time system. The implementation is presently running on two machines, the CDC 6600 at Purdue University and the Siemens 7.760 at the University of

Kiel, West Germany. We measured the following speeds of the implementation:

Machine	Reductions/second
CDC 6600	350
Siemens 7.760	780

This measurement was taken from the LISP interpreter of Example 4.2 (with minor modifications) interpreting a LISP program for reversal of a list.

As a first experiment, we compared the standard LISP program for list reversal with the equational program for list reversal of eqs. (L3)-(L6). For such a program, the standard LISP interpreter goes through the same abstract steps as the equation interpreter; so any difference in speed is the result of the respective overheads of the manipulation of the recursion stack in the LISP interpreter and the pattern matching and replacement steps in the equational interpreter. The interpretation of the equational program was about six times as slow as the LISP program compiled by the LISP compiler written at the University of Texas at Austin. This is an encouraging result, considering that we compare the interpretation of equations by a PASCAL program with compiled machine code.

As a second experiment, we compared the running time of the equational program for LISP interpreting the list reversal program with the compiled version of the same program. Here the results were disappointing, for the slowdown was by a factor of 120.

The second experiment raises the question whether the reduction approach to implementing equational programs can attain production-level efficiency. In light of the first experiment, we have hope. The problem seems to be, not that reduction is inefficient, but that the particular equations used to describe LISP cause too many reductions. The following table shows the number of reductions required by this interpreter to reverse lists of increasing length:

List length	No. of reductions
2	283
4	788
8	2,578
16	9,278
32	35,158

We have begun to vary the LISP equations to see if a slightly different set of LISP equations might yield a more efficient interpreter. By altering the environment structure we have already achieved an improvement of better than 20 percent. We estimate that with a more liberal pattern-matching algorithm a further reduction of 25 percent is possible, but this requires modifications in the run-time system which are not yet implemented. It is also likely that a relaxation of the restrictions of Section 2 to allow certain harmless cases of overlap would increase the efficiency of an equational LISP interpreter enough to make it competitive with hand-coded implementations. This last possibility requires further theoretical work to identify the harmless types of overlap.

7. CONCLUSIONS

Equational programs can serve as a useful programming language in a diversity of applications. The great strength of this approach to programming lies in the simplicity of its semantics, which is accessible even to the nonspecialist. As our examples show, equational programs are easy to develop and very intuitive.

We perceive certain limitations also. In particular, the nonoverlapping requirement seems overly restrictive. Results are needed to classify benign cases of overlap which do not destroy uniqueness and termination properties of reduction sequences. Such results would improve flexibility and reduce the number of equations needed to define a particular computation. Presently there are straightforward methods to detect whether a set of equations satisfies the restrictions of Section 2. It would be helpful to find algorithms to massage equations into the correct form automatically. Reference [25] presents a partial algorithm which sometimes succeeds in modifying equations to have the Church–Rosser property, but this technique only applies when all reduction sequences are finite.

The possibility of compiling rather than interpreting equations is intriguing. Given that equations are used as reduction rules, we might compile them into routines for matching the specific left-hand sides involved and for making the particular replacements given by the right-hand sides. This is the approach of the PROLOG compiler [36]. We do not believe that this approach will lead to a significant improvement in the speed with which reductions can be made. Considering the nature of the matching algorithms employed [21], and considering the reduction speed achieved by the present system, it would be unlikely.

A more radical approach to compiling would attempt to implement an equational program in such a way that only the input–output behavior is preserved, but the individual reduction steps are not followed. It is usually argued that the increased flexibility can result in a significant increase in efficiency. We believe that such an approach must heavily exploit semantic properties of the particular language constructs involved, as in Hoffmann's Lucid compiler [20]. In the case of equational programming in general, the question of how to compile equations without using replacements is entirely open.

APPENDIX A. SEQUENTIAL 2-3 TREE INSERTION

```
ins(K, tree(X)) = tree'(ins(K, X))                                        (SB1)

ins(K, t2(X1, L1, X2))
    = if(K < L1, t2'(ins(K, X1), L1, X2),
      if(K > L1, t2'(X1, L1, ins(K, X2)),
                 unlock(t2(X1, L1, X2))))                                 (SB2)
ins(K, t3(X1, L1, X2, L2, X3))
    = if(OR(K = L1, K = L2), unlock(t3(X1, L1, X2, L2, X3)),
      if(K < L1, t3'(ins(K, X1), L1, X2, L2, X3),
      if(K < L2, t3'(X1, L1, ins(K, X2), L2, X3),
                 t3'(X1, L1, X2, L2, ins(K, X3)))))                       (SB3)
ins(K, e) = put(e, K, e)                                                  (SB4)

t2'(put(X, K, Y), L1, X2) = unlock(t3(X, K, Y, L1, X2))                   (SB5)
t2'(X1, L1, put(X, K, Y)) = unlock(t3(X1, L1, X, K, Y))                   (SB6)
t3'(put(X, K, Y), L1, X2, L2, X3) = put(t2(X, K, Y), L1, t2(X2, L2, X3))  (SB7)
t3'(X1, L1, put(X, K, Y), L2, X3) = put(t2(X1, L1, X), K, t2(Y, L2, X3))  (SB8)
```

```
t3'(X1, L1, X2, L2, put(X, K, Y)) = put(t2(X1, L1, X2), L2, t2(X, K, Y))   (SB9)
tree'(put(X)) = tree(X)                                                    (SB10)

t2'(unlock(X1), L1, X2) = unlock(t2(X1, L1, X2))                           (SB11)
t2'(X1, L1, unlock(X2)) = unlock(t2(X1, L1, X2))                           (SB12)
t3'(unlock(X1), L1, X2, L2, X3) = unlock(t3(X1, L1, X2, L2, X3))           (SB13)
t3'(X1, L1, unlock(X2), L2, X3) = unlock(t3(X1, L1, X2, L2, X3))           (SB14)
t3'(X1, L1, X2, L2, unlock(X3)) = unlock(t3(X1, L1, X2, L2, X3))           (SB15)
tree'(unlock(X)) = tree(X)                                                 (SB16)
```

APPENDIX B. PARALLEL 2-3 TREE INSERTION

```
ins(K, tree(e)) = tree(t2(e, K, e))                                        (PB1)
ins(K, tree(t2(X1, L1, X2))) = tree(ins(K, t2(X1, L1, X2)))                (PB2)
ins(K, tree(t3(X1, L1, X2, L2, X3))) = tree(ins(K, t3(X1, L1, X2, X3)))    (PB3)
ins(K, tree(t4(X1, L1, X2, L2, X3, L3, X4)))
    = tree(ins(K, t2(t2(X1, L1, X2),L2, t2(X3, L3, X4))))                  (PB4)
ins(K, t2(X1, L1, X2))
    = if(K < L1, chk2(K, X1, X2, L1, 1),
      if(K > L1, chk2(K, X2, X1, L1, 2),
                 t2(X1, L1, X2)))                                          (PB5)
chk2(K, e, Y, L, I) = if(I = 1, t3(e, K, e, L, e), t3(e, L, e, K, e))      (PB6)
chk2(K, t2(X1, L1, X2), Y, L, I)
    = if(I = 1, t2(ins(K, t2(X1, L1, X2)), L, Y),
                t2(Y, L, ins(K, t2(X1, L1, X2))))                          (PB7)
chk2(K, t3(X1, L1, X2, L2, X3), Y, L, I)
    = if(I = 1, t2(ins(K, t3(X1, L1, X2, L2, X3)), L, Y),
                t2(Y, L, ins(K, t3(X1, L1, X2, L2, X3))))                  (PB8)
chk2(K, t4(X1, L1, X2, L2, X3, L3, X4), Y, L, I)
    = if(K = L2,
        if(I = 1,t2(t4(X1, L1, X2, L2, X3, L3, X4), L, Y),
                 t2(Y, L, t4(X1, L1, X2, L2, X3, L3, X4))),
      if(K < L2,
        if(I = 1,t3(ins(K, t2(X1, L1, X2)), L2,
                    t2(X3, L3, X4), L, Y)
                 t3(Y, L, ins(K, t2(X1, L1, X2)), L2,
                    t2(X3, L3, X4))),
        if(I = 1,t3(t2(X1, L1, X2), L2,
                    ins(K,t2(X3, L3, X4)), L, Y)
                 t3(Y, L, t2(X1, L1, X2), L2,
                    ins(K, t2(X3, L3, X4))))))                             (PB9)

ins(K, t3(X1, L1, X2, L2, X3))
    = if(OR(K = L1, K = L2), t3(X1, L1, X2, L2, X3),
      if(K < L1, chk3(K, X1, X2, X3, L1, L2, 1),
      if(K < L2, chk3(K, X2, X1, X3, L1, L2, 2),
                 chk3(K, X3, X1, X2, L1, L2, 3))))                         (PB10)
chk3(K, e, X, Y, L, M, I)
    = if(I = 1, t4(e, K, e, L, e, M, e),
      if(I = 2, t4(e, L, e, K, e, M, e),
                t4(e, L, e, M, e, K, e)))                                  (PB11)
chk3(K, t2(X1, L1, X2), X, Y, L, M, I)
    = if(I = 1, t3(ins(K, t2(X1, L1, X2)), L, X, M, Y),
      if(I = 2, t3(X, L, ins(K, t2(X1, L1, X2)), M, Y),
                t3(X, L, Y, M, ins(K, t2(X1, L1, X2)))))                   (PB12)
chk3(K, t3(X1, L1, X2, L2, X3), X, Y, L, M, I)
    = if(I = 1, t3(ins(K, t3(X1, L1, X2, L2, X3)), L, X, M, Y),
      if(I = 2, t3(X, L, ins(K, t3(X1, L1, X2, L2, X3)), M, Y),
```

```
                t3(X, L, Y, M, ins(K, t3(X1, L1, X2, L2, X3)))
    ))                                                              (PB13)
chk3(K, t4(X1, L1, X2, L2, X3, L3, X4), X, Y, L, M, I)
    = if(K = L2,
        if(I = 1, t3(t4(X1, L1, X2, L2, X3, L3, X4),
                     L, X, M, Y),
        if(I = 2, t3(X, L, t4(X1, L1, X2, L2, X3, L3, X4), M, Y),
                  t3(X, L, Y, M,
                     t4(X1, L1, X2, L2, X3, L3, X4))
        )),
      if(K < L2,
        if(I = 1, t4(ins(K, t2(X1, L1, X2)), L2,
                     t2(X3, L3, X4), L, X, M, Y),
        if(I = 2, t4(X, L, ins(K, t2(X1, L1, X2)),
                     L2, t2(X3, L3, X4), M, Y),
                  t4(X, L, Y, M, ins(K, t2(X1, L1, X2)),
                     L2, t2(X3, L3, X4))
        )),
        if(I = 1, t4(t2(X1, L1, X2), L2,
                     ins(K, t2(X3, L3, X4)),
                     L, X, M, Y),
        if(I = 2, t4(X, L, t2(X1, L1, X2), L2,
                     ins(K, t2(X3, L3, X4)), M, Y),
                  t4(X, L, Y, M, t2(X1, L1, X2), L2,
                     ins(K, t2(X3, L3, X4)))
    ))))                                                            (PB14)
```

Computation using eqs. (PB5)–(PB14) proceeds as follows. Upon encounter of a 2-node or a 3-node, the node is locked (via chk2 or chk3), and the proper subtree in which the insertion has to be made is located. The subtree becomes the second parameter of the function chk2 or chk3. If the root of that subtree is a 4-node, then this node plus the locked parent is restructured according to the transformations of Figure 5. After restructuring, the parent node is released. If the root of the subtree is a 3-node or a 2-node, then no restructuring is needed, and the parent node is released. If the subtree is empty, then the locked parent node is a leaf, and we insert the key. The equations also account for the possibility that the key to be inserted is in the tree already. Using these equations, we can only attempt inserting a key into a 2-node or a 3-node; thus no upward traversal is needed, and insertions can be done in parallel.

REFERENCES

(Note. References [4, 6, 14, 24] are not cited in the text.)

1. AHO, A., AND ULLMAN, J. *The Theory of Parsing, Translation, and Compiling*, vol. 1: *Parsing*. Prentice-Hall, Englewood Cliffs, N.J., 1972.
2. ASHCROFT, E.A., AND WADGE, W.W. Lucid, a nonprocedural language with iteration. *Commun. ACM 20*, 7 (July 1977), 519–526.
3. ASHCROFT, E., AND WADGE, W. Lucid—A formal system for writing and proving programs. *SIAM J. Comput. 5*, 3 (1976), 336–354.
4. BACKUS, J. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Commun. ACM 21*, 8 (Aug. 1978), 613–641.
5. BACKUS, J. Programming language semantics and closed applicative languages. In Conf. Rec., ACM Symp. Principles of Programming Languages, 1974, pp. 71–86.

6. BAUER, F.L., BROY, M., GNATZ, R., HESSE, W., KRIEG-BRUCKNER, B., PARTSCH, H., PEPPER, P., AND WOSSNER, H. Towards a wide spectrum language to support program development by transformations. In *Lecture Notes in Computer Science*, vol. 69: *Program Construction: International Summer School*. Springer-Verlag, New York, 1979, pp. 543–552.
7. BERRY, G., AND LÉVY, J.-J. Minimal and optimal computations of recursive programs. In Conf. Rec., 4th ACM Symp. Principles of Programming Languages, Los Angeles, Calif., Jan. 17–19, 1977, pp. 215–226.
8. BJORNER, D. Finite state tree computations, pt. I. IBM Res. Tech. Rep. RJ 1053 (#17598), 1972.
9. BRUYNOOGHE, M. An interpreter for predicate logic programs, pt. I. Rep. CW10, Applied Math. and Programming Div., Katholieke Univ., Leuven, Belgium, 1976.
10. CADIOU, J. Recursive Definitions of Partial Functions and Their Computations. Ph.D. dissertation, Computer Science Dep., Stanford Univ., Stanford, Calif., 1972.
11. CARGILL, T. Deterministic operational semantics for Lucid. Res. Rep. CS-76-19, Univ. Waterloo, Ontario, 1976.
12. CHEW, L.P. An improved algorithm for computing with equations. In 21st Ann. IEEE Symp. Foundations of Computer Science, 1980, Syracuse, N.Y., pp. 108–117.
13. DOWNEY, P., AND SETHI, R. Correct computation rules for recursive languages. *SIAM J. Comput. 5*, 3 (1976), 378–401.
14. FARAH, M. Correct Compilation of a Useful Subset of Lucid. Ph.D. dissertation, Dep. Computer Science, Univ. Waterloo, Ontario, 1977.
15. FRIEDMAN, D., AND WISE, D. Cons should not evaluate its arguments. In *3rd Int. Colloq. Automata, Languages and Programming*. Edinburgh University Press, Edinburgh, Scotland, 1976, pp. 257–284.
16. GOGUEN, J. Abstract errors for abstract data types. In *IFIP Working Conference on Formal Description of Programming Concepts*, E.J. Neuhold (Ed.). Elsevier North-Holland, New York, 1977, pp. 491–522.
17. GUIBAS, L., AND SEDGEWICK, R. A dichromatic framework for balanced trees. In 19th IEEE Symp. Foundations of Computer Science, Ann Arbor, Mich., 1978, pp. 8–21.
18. GUTTAG, J., HOROWITZ, E., AND MUSSER, D. Abstract data types and software validation. Information Science Res. Rep. ISI/RR-76-48, Univ. Southern Calif., Los Angeles, 1976.
19. HENDERSON, P., AND MORRIS, J.H., JR. A lazy evaluator. In Conf. Rec., 3d ACM Symp. Principles of Programming Languages, Atlanta, Ga., Jan. 19–21, 1976, pp. 95–103.
20. HOFFMAN, C. Design and correctness of a compiler for a nonprocedural language. *Acta Inf. 9*, 3 (1978), 217–241.
21. HOFFMANN, C.M., AND O'DONNELL, M.J. Pattern matching in trees. To appear in *J. ACM 29*, 1 (Jan. 1982), 68–95.
22. HOFFMANN, C.M., AND O'DONNELL, M.J. An interpreter generator using tree pattern matching. In Conf. Rec., 6th Ann. ACM Symp. Principles of Programming Languages, San Antonio, Tex., Jan. 29–31, 1979, pp. 169–179.
23. HUET, G., AND LÉVY, J.-J. Computations in nonambiguous linear term rewriting systems. IRIA Tech. Rep. 359, 1979.
24. JOHNSON, S.D. An interpretive model for a language based on suspended construction. Tech. Rep. 68, Dep. Computer Science, Indiana Univ., Bloomington, Ind., 1977.
25. KNUTH, D., AND BENDIX, P. Simple word problems in universal algebras. In *Computational Problems in Abstract Algebra*, J. Leech (Ed.). Pergamon Press, Elmsford, N.Y., pp. 263–297.
26. KOWALSKI, R. Algorithm = logic + control. *Commun. ACM 22*, 7 (July 1979), 424–436.
27. MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine, pt. I. *Commun. ACM 3*, 4 (April 1960), 184–195.
28. NELSON, G., AND OPPEN, D.C. Fast decision procedures based on congruence closure. *J. ACM 27*, 2 (April 1980), 356–364.
29. NELSON, G., AND OPPEN, D.C. A simplifier based on efficient decision algorithms. In Conf. Rec., 5th Ann. ACM Symp. Principles of Programming Languages, Tucson, Ariz., Jan. 23–25, 1978, pp. 141–150.
30. O'DONNELL, M. *Lecture Notes in Computer Science*, vol. 58: *Computing in Systems Described by Equations*. Springer-Verlag, New York, 1977.
31. ROBERTS, G. An Implementation of PROLOG. M.S. thesis, Dep. Computer Science, Univ. Waterloo, Ontario, 1977.

32. ROSEN, B.K. Tree-manipulating systems and Church-Rosser theorems. *J. ACM 20*, 1 (Jan. 1973), 160–187.
33. STAPLES, J. A class of replacement systems with simple optimality theory. *Bull. Aust. Math. Soc. 17*, 3 (1977), 335–350.
34. VUILLEMIN, J. Correct and optimal implementations of recursion in a simple programming language. *J. Comput. Syst. Sci. 9*, 3 (1974), 332–354.
35. WAND, M. First order identities as a defining language. Tech. Rep. 29, Dep. Computer Science, Indiana Univ., Bloomington, Ind., 1976.
36. WARREN, D. Implementing PROLOG. Res. Reps. 39, 40, Dep. Artificial Intelligence, Univ. Edinburgh, Scotland, 1977.

Received July 1979; revised May 1980 and May 1981; accepted June 1981