Implementation of an Interpreter for Abstract Equations

CHRISTOPH M. HOFFMANN
Department of Computer Sciences, Purdue University, West Lafayette, IN 47907, U.S.A.

AND

MICHAEL J. O'DONNELL AND ROBERT I. STRANDH
Electrical Engineering and Computer Science Department, The Johns Hopkins University, Baltimore, MD 21218, U.S.A.

SUMMARY

This paper summarizes a project whose goal is the implementation of a useful interpreter for abstract equations that is absolutely faithful to the logical semantics of equations. The interpreter was first distributed to Berkeley UNIX VAX sites in May 1983. The main novelties of the interpreter are strict adherence to semantics based on logical consequences, 'lazy' (outermost) evaluation applied uniformly, an implementation based on table-driven pattern matching, with no run-time penalty for large sets of equations, and strict separation of syntactic and semantic processing, so that different syntaxes may be used for different problems.

KEY WORDS equation reduction normal form lazy evaluation pattern-matching logic programming interpreter

INTRODUCTION

To illustrate how to write programs in the form of equations, we start by giving an example of a program to reverse a list of elements.

Example 1

```
Symbols
  cons:2;
  nil:0;
  rev:1;
  catrev:2.

For all l, h, t:
  rev[l] = catrev[l;()];

  catrev[();l] = l;
  catrev[(h . t);l] = catrev [t;(h . l)].
```

0038-0644/85/121185-20$02.00

Received 6 February 1984
Revised 17 December 1984 and 26 March 1985

The first part is a declaration of the symbols used in the program, together with their arities, i.e. the number of parameters they take. The symbol cons is the normal LISP list constructor, and takes two arguments. Another familiar symbol is nil, with no arguments. We do not distinguish here between a constant and a function with no arguments. Notice that neither cons nor nil is explicitly mentioned in the equations. However, the notation () is another way of writing nil, and (h . t) stands for cons[h;t]. Starting the second part of the program is a declaration of all the variables used in the program. Here, we use l,h and t for list, head and tail, respectively. Finally, the actual equations are given. The method we use to reverse the list is to call an auxiliary function (catrev) with two parameters. This new function takes elements from the beginning of its first argument and conses them to its second argument (initially the empty list). When the first argument is empty, the second is the reversed list.

This program is input to an equation preprocessor. The preprocessor creates an executable program, the interpreter. For instance, suppose we want to reverse the list (a b c). We would start the interpreter and give it the input rev[(a b c)]. It would print (c b a) and exit. The input and output are examples of terms. A term is processed by replacing in it any instance of a left-hand side of an equation by the corresponding right-hand side. This process is called reduction, and the term (c b a) is said to be in normal form, since it cannot be further reduced. In reducing the term rev[(a b c)], we first find that it matches the left-hand side of the first equation, with l=(a b c), so we substitute the right-hand side of the term and obtain catrev[(a b c];()]. This matches the third equation, with h=a and t=(b c). We substitute the right-hand side of the third equation, giving a new term, catrev[(b c);(a)]. Again this matches the third equation, this time with h=b and t=(c). We substitute catrev[(c),(b a)]. Another match with the third equation occurs and catrev[();(c b a)] is substituted. This matches the second equation with l=(c b a), so we substitute (c b a). Recall that (c b a) is short for the less readable cons[c;cons[b;cons[a;nil]]]. Since there is no equation that matches this term, it is in normal form, and is printed. In the example above, we used LISP syntax. The equation processor permits other syntaxes as described below.

The prime motivation for the equation interpreter project was to develop a programming language whose semantics can be described completely in terms of simple mathematical concepts. We chose equations as the notation for the project because E=F has

(1) an obvious mathematical interpretation — E and F are different names for the same thing
(2) a natural and simple computational interpretation — replace E by F whenever possible
(3) well-documented theoretical results on the equivalence of these two interpretations — Church–Rosser or confluence theorems.

A *program* for the equation interpreter is a list of symbols to be used, followed by a list of equations invoking those symbols and variables. The *meaning* of a program is completely described by the following.

Definition 1

A term containing no instance of a left-hand side of an equation is in *normal form*. The term (c b a) in the example above is in normal form.

An *interpreter* for a set of equations is a program that, given an input term E, produces a term F, in normal form, such that E=F is a logical consequence of the equations, if such an F exists. If no such F exists, the interpreter must not produce output. In the example above rev[(a b c)] = (c b a) is a logical consequence of the equations, and (c b a) is in normal form. A BNF syntax for the input to the preprocessor is shown in Figure 1. The structure of the non-terminal symbol ⟨*term*⟩ depends on the syntax, and is described below. Symbol descriptions indicate one or more symbols in the language to be defined, and give their arities. Intuitively, symbols of arity 0 are the constants of the language, and symbols of higher arity are the operators. Syntactically, *symbols* and *symbol classes*

```
<program> ::= Symbols <symbol description list>. For all <variable list>:<equation list>.

<symbol description list> ::= <symbol description>; . . . ; <symbol description>

<symbol description> ::= <symbol list>:<arity> | include <symbol class list>

<symbol class list> ::= <symbol class>, . . . , <symbol class>

<symbol class> ::= atomic_symbols | integer_numerals | truth_values

<symbol list> ::= <symbol>, . . . , <symbol>

<symbol> ::= <string of nonblank alphanumeric characters, starting with alphabetic>

<arity> ::= <number>

<variable list> ::= <variable>, . . . , <variable>

<variable> ::= <string of nonblank alphanumeric characters, starting with alphabetic>

<equation list> ::= <equation>; . . . ; <equation>

<equation> ::= <term> = <term> |
               <term> = <term> where <qualification> end where |
               include <equation class list>

<qualification> ::= <qualification item list>

<qualification item list> ::= <qualification item>, . . . ,<qualification item>

<qualification item> ::= <variable> is <qualified term> | <variable list> are <qualified term>

<qualified term> ::= in <symbol class> |
                     <term> |
                     <qualified term> where <qualification> end where |
                     either <qualified term list> end or

<qualified term list> ::= <qualified term>or . . . or <qualified term>

<equation class list> ::= <equation class> |
                          <equation class> <equation class list>

<equation class> ::= addint | subint | multint | divint | modint |
                     lessint | equint | equatom
```

Figure 1.

are identifiers. A symbol class indicates the inclusion of a predefined class of symbols. The classes available are atomic_symbols, integer_numerals and truth_values. The predefined symbol classes correspond to the basic data types in other programming languages such as Pascal. More complex data types can be built by using constructors, as in LISP. Symbols that have been explicitly declared in the Symbols section are called *literal symbols*, to distinguish them from members of the predefined classes.

Variables are identifiers, just like symbols: *Equation classes* are identifiers indicating the inclusion of a large number of predefined equations. The equation classes include the defining equations for the standard arithmetic operations. For example, addition is defined by add[1;1] =2, add[1;2] =3, etc. Of course, such equations are not stored explicitly, but their effect is produced by efficient machine operations. Note that equation classes do not provide a mechanism for users to define their own equations for later inclusion, but a way to include the equational equivalent of library routines.

Qualifications, as defined above, restrict the ranges of variables to terms of given forms. Normally, variables range over all terms in the language described by the Symbols section. Different syntaxes for terms may be chosen to suit different problems. The module that takes a term and translates it to its internal abstract syntax is separate from the semantic processor, and may be replaced by another, possibly supplied by the user. The same goes for the unparser, i.e. the module that takes the internal form of a term and converts it to readable format. In this paper, we use two different term syntaxes: the standard mathematical notation in which function application is denoted f(a,b), and LISP notation, in which such application is f[a;b], and parentheses are used to abbreviate uses of the special binary operator cons to build lists and trees. A lambda notation is also available, and it is straightforward to add, for example, *yacc* and *lex* programs for other syntaxes as needed. The standard syntax modules are written this way, but the users can also write their own syntax from scratch.

The following example illustrates all of the constructs described above, using LISP notation. A colon initiates a comment line.

Example 2

```
Symbols

 : List constructors
   cons: 2; nil: 0;

 : Standard arithmetic operators
   add: 2;

 : nonils[x] is the list containing all of the non-nil
 :    elements of the list x
   nonils: 1;

 : leafcount[x] is the number of leaves in the tree x
   leafcount: 1;

   include atomic_symbols, integer_numerals.
```

```
For all x, y, z, rem:

  nonils [()] = ();
  nonils [(() . rem)] =nonils[rem];

  nonils [(x . rem)] = (x . nonils[rem])
    where x is either (y . z)
                 or in atomic_symbols
               end or
    end where;

  leafcount [()] =0;
  leafcount [(x . y)] =add[leafcount[x]; leafcount[y]];
  include addint.
```

Notice that the symbols cons, nil and add must be declared. They appear implicitly in the equations. include addint is semantically equivalent to the set of equations defining addition of integer numerals.

In order for the reduction strategies used by the equation interpreter to be correct according to the logical-consequence semantics, some restrictions must be placed on the equations. At present, 5 restrictions are enforced.

1. No variable may be repeated on the left side of an equation. For instance
   if(x, y, y) = y
   is prohibited, because of the 2 instances of y on the left side. The interpreter does not look at the values of the variables. It just keeps their values in order for the right-hand side to be able to refer to them. Allowing repeated variable names on the left-hand side requires tests for equality that would consume large amounts of time, or even fail to halt.
2. Every variable appearing on the right side of an equation must also appear on the left. For instance, f(x) = y is prohibited. The idea is that a variable on the right-hand side of an equation refers to some part of the term matched by the left-hand side. Note that variables on the left need *not* appear on the right, as in left(pair(x,y)) =x.
3. Two different left sides may not match the same expression. So the pair of equations
   g(0,x) =0; g(x,1)=1
   is prohibited. If we were to reduce the term g(0,1), which matches both equations, the result would be 0 if we decide to match the first equation, and 1 if we match the second.
4. 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
   first(pred(x)) = predfunc; pred(succ(x)) =x
   is prohibited, since the left-hand sides overlap in first(pred(succ(0)). Again the result of reducing the term would give different results, depending on which equation we decide to use.
5. It must be possible, in a left-to-right preorder traversal of any term, to identify an instance of a left-hand side without traversing any part of the term below that instance. For example, the pair of equations

f(g(x,a),y) =0; g(b,c) =1
is prohibited, since after scanning f(g is is impossible to decide whether to look at the first argument to g in hopes of matching the b in the second equation, or to skip it and try to match the first equation.

Note that in restriction 5, to 'look at' means to possibly have to reduce. Suppose we feed the equations the term f(g(h(a), k(b)),c). In the first equation x would match h(a), so if we decide to match that one, we proceed to try to find a as the second argument to g. We find k(b). We must then reduce k(b), since it does not match. Suppose k(b) reduces to c. Then we do not match the first equation, but possibly the second one. In this case, we must go back and reduce h(a), hoping that it will reduce to b, so that we can match the second equation. We want to avoid going back in this manner. If we instead decide to match the second one, we have to reduce h(a). Suppose that h(a) eventually after a long computation reduces to d. Then we have done the computation unnecessarily. It may not even be relevant if k(b) reduces to a, given that the first equation matches. Even worse, h(a) may involve an infinite computation that could have been avoided.

Sets of equations satisfying 1–4 above are called *regular*. The property described in (5) is *strong left-sequentiality*. Violations of strong left-sequentiality may often be avoided by reordering the arguments to a function. Strong left-sequentiality is treated in more detail below.

Restrictions 1–4 guarantee that normal forms are unique, and that outermost evaluation will find all normal forms. Restriction 5, which technically subsumes the other four, will be removed in a later version with an implementation of parallel evaluation. Restriction 3 will also be relaxed to allow different left-hand sides to match when the corresponding right-hand sides agree, as in or(True,x) =True, or(x,True) =True.

HISTORY OF THE PROJECT

The logical foundations of the project, results concerning uniquenss of normal forms and correct orders of evaluation, come from Reference 1. From 1978 to 1981, work focused on theoretical development of algorithms and the building of prototype systems.[2] The system was rewritten for distribution in 1982–1983, and an earlier version was ported to Kiel, Germany,for two projects involving alternative approaches to pattern-matching in the interpreter, and use of the interpreter to define interpreters and compilers for Pascal. Current work is directed towards several aspects of equational processing, such as incremental preprocessing techniques to improve preprocessing time and allow for modular constructs, compiling equations into efficient machine code, and table compression techniques to improve preprocessing time and size of the interpreter/machine program.

Among other non-procedural programming languages, the one most similar in flavour to the equation interpreter is Prolog.[3] Prolog accepts Horn clauses in the first-order predicate calculus as programs. All existing Prolog interpreters and compilers are incomplete — they sometimes fail to produce output even though an output follows logically from the program. This is a consequence of Prolog's computationally very expensive semantics: a complete implementation of Prolog requires a breadth-first evaluation of the proof tree, but this would require an unacceptable amount of space. Prolog implementors have therefore chosen to evaluate the proof tree depth-first, with backtracking. This evaluation strategy introduces a procedural element absent in the strict semantics, and is responsible for the implementations' failure to produce logically entailed output in certain cases. Since equation semantics is computationally simpler than Prolog semantics, our equation interpreter can produce all of the logically entailed outputs without making unacceptable resource demands. Some Prolog programs rely on the fact that the interpreter backtracks. These programs are of course easier to express in a backtracking Prolog system, than in equations. To program such a computation with equations involves programming the backtracking explicitly.

Another language processor similar syntactically to the equation interpreter is HOPE,[4] which also uses equations as programs. HOPE has more stringent restrictions on equations than our interpreter. For example, HOPE distinguishes function and constructor symbols and prohibits equations in which subexpressions involve function symbols. This restriction greatly simplifies the pattern matching required to find instances of equation left-hand sides. We believe that in view of our pattern matching algorithms such a simplification does not lead to a significant performance improvement. HOPE uses conventional innermost evaluation, instead of lazy evaluation, for all operators except the conditional and cons. So, it is possible to write equations, involving constructors other than cons, for which HOPE will fail to find a logically entailed normal form because it follows an irrelevant infinite evaluation of a subterm not included in the final output.

There have been hybrid approaches to equational programming in which the equations are assigned a priority, e.g. References 5 and 6. If several reductions are possible at the same position, then the one whose equation has the highest priority is chosen. Such programming systems do not have a neat, well-understood semantics, but their proponents consider them easy to use and of practical importance. If one were to compare this approach to ours, it should be remembered that we wish to obtain 'a practically useful programming system witout sacrificing semantic rigour.

SEMANTIC STRICTNESS AND ITS CONSEQUENCES

The first and foremost design decision was to be absolutely faithful to the logical semantics. The only type of failure tolerated in the process of reducing a term to normal form is exhaustion of the available space resources. The main consequence of this decision was the necessity of implementing lazy evaluation uniformly. Lazy, or demand driven, evaluation means that an expression gets evaluated only if its value is needed. Suppose we have a list, l, of integers, say l = (1 2 3 4 5). Suppose further that we want to evaluate the expression head[add1tolist[l]], where head is the function that gives back the first element of the list given as its argument, and add1tolist is a function that adds one to every element of a list. In an ordinary programming language, we would first evaluate the arguments to the function head, i.e. we would add 1 to every element of l. Then we would feed the value, in this case (2 3 4 5 6), to the function head, giving the value 2. Note how we first added one to all the elements of the list, and then threw away 4 out of 5 of the elements. The equational program would look something like this:

```
head[(h . t)] = h;

add1tolist [()] = ();
add1tolist [(h . t)] = (add1 [h] . add1tolist [t]);
```

The evaluation of the expression head [add1tolist[l]] would proceed as follows

```
head[add1tolist[(1 2 3 4 5)]]
head[(add1[1] . add1tolist[(2 3 4 5])]
add1[1]
2
```

Note how the additions to the rest of the list were never performed. The example is a little artificial, since most programmers would not write code like that; they would write add1[head[l]] instead. However, if we want to form a list of the first n elements of add1tolist[l], where n and l are both unknown at compile time, the example is more realistic. The difference is even more dramatic if l happens to be infinite. This would not be possible in most programming languages, but in the equation interpreter it makes sense. For instance

```
listof ones[] =(1 . listofones[])
```

is an infinite list of the number one, repeated over and over. A more useful example would be a list of all the primes or all the Fibonacci numbers. In a normal programming language, if you tried to evaluate the expression firstn[3;listofones[]] hoping to get the first three elements of the infinite list of ones, i.e. (1 1 1), the arguments would be evaluated first, giving an infinite computation. With lazy evaluation, there is no problem. The expression listofones[], would be evaluated to (1 1 1 . listofones[]), at which point firstn has the list it needs in order to take the first three elements and form a list out of. Firstn would give back the result (1 1 1) as desired.

Nearly all programming languages use lazy evaluation to evaluate conditionals, and like treatment of the LISP function cons has been proposed in References 4, 7 and 8, but we know of no previous language processor that implements lazy evaluation in all cases. Kahn and MacQueen[9] have a Pascal-like dataflow language in which all communication between coroutines is performed in a demand-driven fashion equivalent to lazy evaluation, but expressions inside routines are evaluated conventionally. Recent correspondence indicates that J. Schwartz now has a lazy HOPE interpreter. Lazy evaluation has advantages for the user, allowing straightforward use of a certain type of parallel programming. References 7 and 8 demonstrate some of these advantages in the case of LISP. As described below, lazy evaluation automatically performs one of the design tasks in dynamic programming that otherwise must be programmed explicitly.

Another important consequence of semantic strictness involves the inclusion of efficient machine operations as primitives. Take, for example, integer addition. In principle, addition may be defined from zero and successor by equations such as

```
add(0,x) =x
add(s(y),x) =s(add(y,x))
```

These equations produce an addition that is semantically correct, but unacceptably inefficient. The conventional course is to invoke the machine's addition operation to evaluate add(i,j) whenever i and j are integer numerals. The latter course is efficient, but cannot be explained very well by logical consequence semantics because of the possibility of overflow. In the equation interpreter, we may combine the good points of both approaches. What the machine addition really does is to implement the large, but finite, set of equations add(1,1) =2, add(1,2) =3, etc., representing those additions not causing overflow. Those machine-implemented equations can be augmented by equations for addition in base maxint, where maxint is the largest integer represented by the machine. Thus, the user has the benefit of the precise semantics of integer addition on arbitrarily large numbers, and the efficiency of machine addition in the usual case where the numbers are not large. Because of lazy evaluation and the pattern-matching techniques described below, single-precision arithmetic does not have to pay the overhead of checking whether the inputs are single precision. In fact, only the normal amount of work — to determine if the expression can be reduced or not — is required.

The current version of the equation interpreter allows use of machine-implemented single-precision arithmetic, and leaves to the user the definition of multiprecision arithmetic. Operations that, in conventional programs, would cause overflow, are simply not performed. This means that even the user who has not written multiprecision equations sees correct, but possibly less helpful, output. The output of something like fibonacci[n], may be add[a,b], where the sum of a and b gives the correct Fibonacci number, but the addition cannot be performed, since it would cause arithmetic overflow. Of course, the equations for arithmetic and other natural primitives should be written once and saved to avoid duplication of effort. The facility to do so seems to be a special case of the general need for facilities to structure and combine equational definitions, discussed below. We have chosen to await results in the more general area, rather than to perform an *ad hoc* extension for primitive operations.

THE IMPORTANCE OF PATTERN-MATCHING ALGORITHMS

In order for programming with equations to be a real improvement over more conventional programming styles, it is important to be able to write many equations, preferably between small terms, rather than a few huge ones. If all of the information about a function f is given by a single equation, f(x) =T, then the term T is essentially just a lazy LISP program for f. In order for equational programming to serve a purpose not already served by LISP, one must have an interpreter capable of processing many equations giving different pieces of the definitions of functions. The implementation must not penalize programs for using a large number of equations by sequential checking of the left-hand sides of equations to see which ones apply. In order to compete in performance with conventional LISP interpreters, the process of finding the next subexpression to replace must have a cost comparable to the cost of manipulating the recursion stack in LISP.

Instead of sequential checking, we preprocess the equations and produce tables to drive the reduction. These tables describe state transitions during a traversal of the term that indicate immediately when an instance of an equation left-hand side is found, and tell which equation is involved. The overhead of each traversal step at run-time is only a table look-up.

For multiple-pattern string matching, the Aho–Corasick generalization of the Knuth–Morris–Pratt algorithm[10, 11] solves a problem closely analogous to ours. Extension of these string-matching techniques to terms (equivalently, trees) was treated separately in Reference 12. In the last year of the project, we discovered that the restrictions already imposed upon equations for other reasons allow for a much simpler extension of string matching techniques. The following subsection assumes an understanding of the Aho–Corasick algorithm.

A specialized pattern-matching algorithm

The current version of the equation interpreter is *left-sequential*. That is, a term to be reduced is traversed to the left first, and any left-hand side that is found is replaced before the traversal continues. Such a strategy cannot deal with certain equations, such as the *parallel or* equations:

or (True,x) = True; or(x,True) = True.

The interpreter preprocessor detects and rejects such equations. For left-sequential equations, a special and simple pattern-matching algorithm may be used. Left-sequentiality is a special case of strong sequentiality of Huet and Lévy,[13] which has an algorithm that is theoretically more powerful than ours, but it is more complex, and its space efficiency is not known.

Tree patterns are flattened into preorder strings, omitting variables. The Aho–Corasick algorithm is used to produce a finite automaton recognizing those strings. Each state in the automaton is annotated with a description of the tree moves needed to get to the next symbol in the string, or with the pattern that is matched, if the end of the string has been reached. Such descriptions need only give the number of edges (≥ 0) to travel upwards toward the root, and the left-right number of the edge to follow downwards. For example, the patterns (equation left-hand sides) f(f(a,x),g(a,y)) and g(b,x), with corresponding tree representations in Figure 2, generate the strings ffaga and gb, and the automaton given in Figure 3. Note the directives in the nodes to go up or down in the tree representing the term. Suppose that we start with the term f(f(a,c),g(b,d)). The tree representation for this term is shown in Figure 4. To see if our input string matches a pattern we follow the sequence of steps shown in Figure 5. Initially (Figure 5(a)) we start with the automaton in state 0 and look at the root node of the expression. The root node is labelled f. We therefore take the f branch in the automaton leading to state number 1. The directions in state number 1 say 'd1' for 'go down the first branch'. So we move the pointer in our expression tree down the first (the left) branch. This gives the new situation of Figure 5(b). Again we see an f in the expression tree. So we take the f branch of the automaton, leading to state number 2, where we are instructed to go down the first branch, giving the situation of Figure 5(c). Here we are looking at an a so we move to state number 3, where we are told to go up 2 steps and then down the second branch. So we go up to the root and then down the right branch, giving the situation shown in Figure 5(d). Here we see a g so we move to state number 4 and go down the first (left) branch of the expression tree. We now have the situation of Figure 5(e). We are now looking at a b, but there is no arc labelled b from state 4. So we have to follow the failure arc, leading to state number 6 and retry with our b input. This time there is an arc labelled b to state number 7, where we find that we have matched pattern number 2 (Figure 5(f)), namely g(b,x), with d for the variable x.

If we instead have the slightly different pair of patterns f(f(a,x),g(a,y)) and g(x,b), we cannot build an automaton. To see this possibility, notice that the strings generated are the same as in the example above, namely ffaga and gb. The automaton would look like

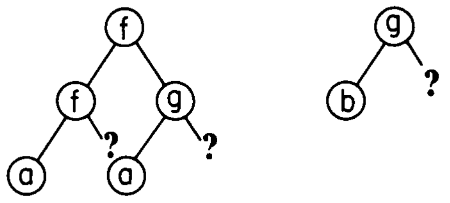

Figure 2.

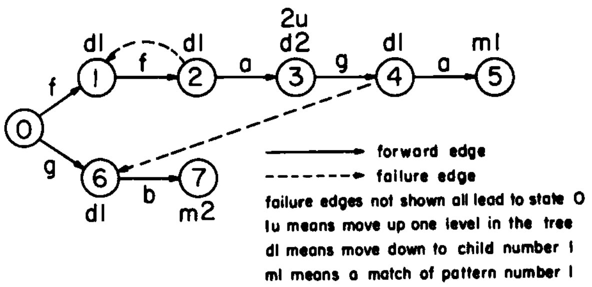
2u
d2
dl
dl
dl
ml
f
f
a
g
a
g
b
dl
m2
forward edge
failure edge
failure edges not shown all lead to state 0
lu means move up one level in the tree
dl means move down to child number l
ml means a match of pattern number l

Figure 3.

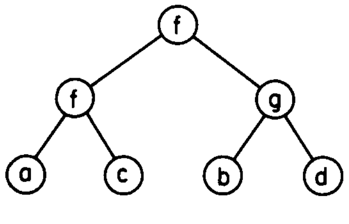

Figure 4.

Figure 6. The only difference from the example above is the annotation of the states. Notice how states 4 and 6, which are connected by a failure arc, have conflicting directions for moving in the expression tree. This conflict cannot be allowed. Such conflicts occur precisely when there exist preorder flattened strings of the forms $\alpha\beta\gamma$ and $\beta\delta$, such that the annotations on the last symbol of β in the two strings are different. These differences are discovered directly by attempts to reassign state information in the automaton when α is the empty string, and by comparing states at opposite ends of failure edges when α is not empty. When γ and δ are not empty, the conflicting annotations are both tree moves, and indicate a violation of restriction (5) given in the first section. When one of γ, δ is the empty string, the corresponding annotation reports a match, and indicates a violation of restriction (3) or (4). In the example above, there is a conflict with $\alpha = \mathtt{f\ f\ a}$, $\beta = \mathtt{g}$, $\gamma = \mathtt{a}$, $\delta = \mathtt{b}$. That is, after scanning f f a g, the first pattern directs the traversal down edge number 1, and the second pattern directs the traversal down edge number 2. This conflict is discovered because there is a failure arc between states with these two annotations.

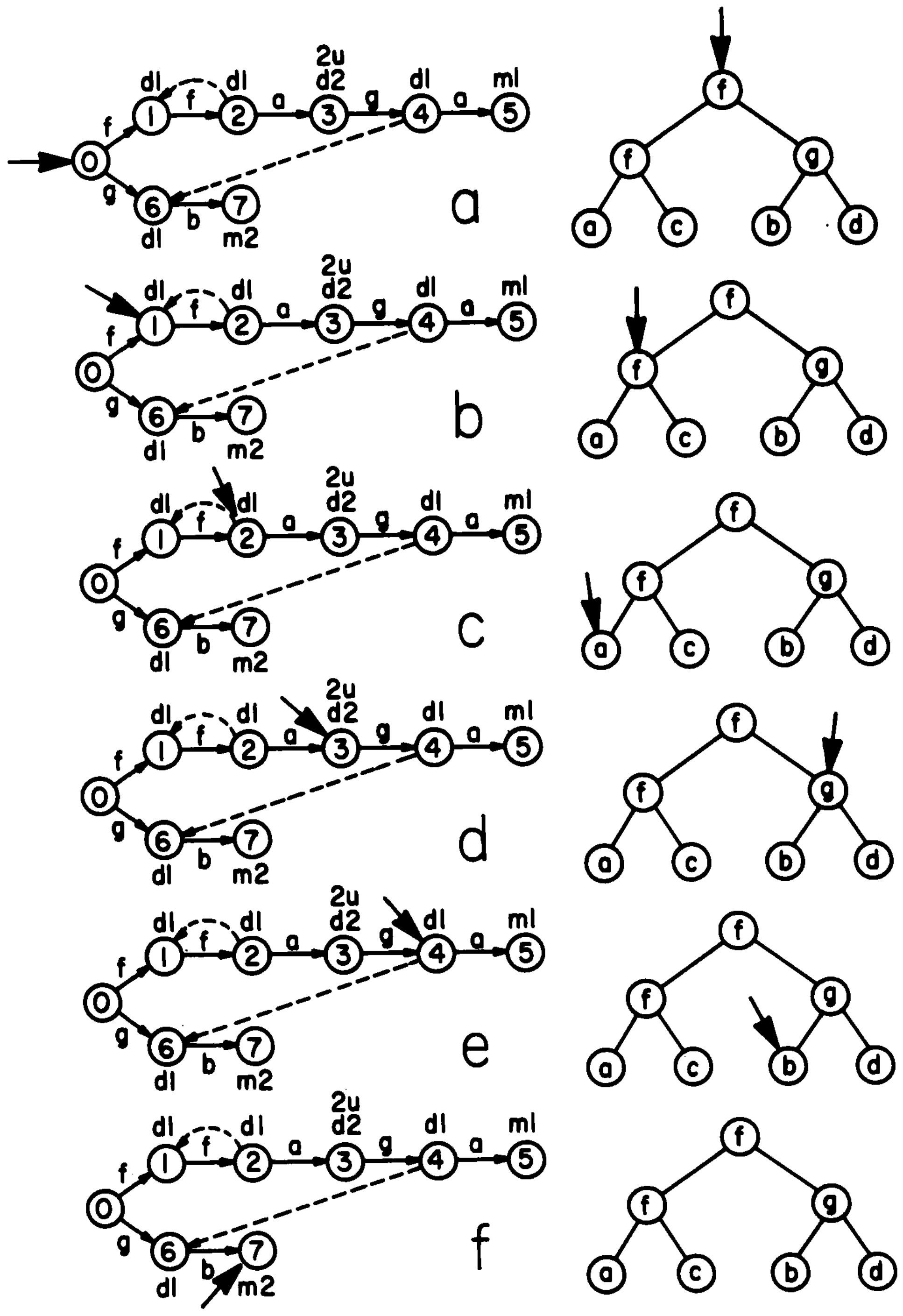

Figure 5.

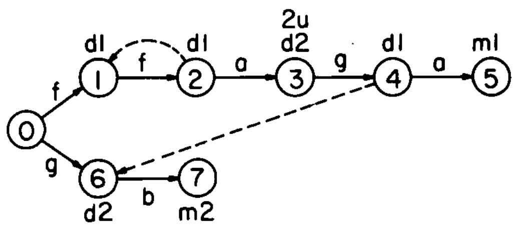

Figure 6.

The restrictions imposed on equations by the pattern-matching strategy above may be justified in a fashion similar to the justification of determininstic parsing strategies. That is, we show that the algorithm succeeds (generates no conflicts) on every set of equations that is left-sequential according to a reasonable abstract definition of sequentiality. The details can be found in References 14 and 15.

Interpreting non-sequential sets of equations

In the future, an improved version of the equation interpreter should eliminate the restriction to strongly left-sequential systems, and allow definitions of constructs such as the parallel or. The pattern-matching algorithm may be extended to handle non-sequential systems by annotating each state in the automaton with a non-empty set of tree moves. When more than one move is specified, parallel processes must be initiated to follow the different possibilities. This approach keeps the degree of parallelism low (but not always the lowest possible), which is desirable on sequential hardware. To maintain acceptable performance, these processes must be able to wait for results produced by other such processes when two or more of them wander into the same region (otherwise work will be duplicated), and a process must be killed whenever a second process creates a potential reduction containing the first one (otherwise wasted work may be done on a subterm that has been discarded). Solutions to these problems are well known in principle, but careful study is required to implement them with a small time and space overhead. Even when such an implementation is accomplished, sequential algorithms such as ours and Huet and Lévy's[13] will be useful because they can avoid the overhead of parallel methods.

SEPARATION OF SYNTACTIC PROCESSING FROM SEMANTICS

One of the main problems in making the equation interpreter useful to a human programmer is the syntax of terms, whether they occur in equations or as input to the system. Prefix notation is the standard of reference in mathematics, but is almost never convenient for a specific application. We discovered this problem with a prototype interpreter, when we tried to write equations defining LISP. Most of our time was spent wrestling with complex expressions for simple lists, such as cons(1,cons(2,cons(3,cons(4,nil)))), for (1 2 3 4), instead of thinking about semantic issues. Unfortunately, different domains of computation seem to have developed different notations, and we know of none that is universally acceptable. So, we decided to communiate with the equation interpreter through a number of different front ends, stored in a standard library. Users may, of course, use their own if the ones provided do not suffice. It is important to be able to use the same syntactic definitions of terms to parse terms in equations, and to parse terms before evaluation.

A way to separate syntax and semantics thoroughly is to use an explicit uniform internal form for the abstract syntax of terms and equations, into which special syntaxes are translated. This internal syntax is string-based, which greatly simplifies porting the system to a new machine. These front ends may be written in any programming language. Structure editors are the ideal front ends in our view, but at present we use *lex* and *yacc* to produce parsers. Of course, for consistency the interpreter also produces its output in internal form, and the output is then sent to one of a library of pretty-printers for display. Current parsing technology makes it easy to use the same grammar for terms in parsing both preprocessor and intepreter input, but the (much easier) pretty-printers are written separately.

Several advantages result from the discipline of using an explicit intermediate form between text produced by the user and semantic processing by the system. First is the complete separation of syntactic and semantic modules. Conventional use of grammars to generate parsers requires a complex interface between the parser and the semantic processor, specialized to the particular parser generator. We require no internal connection whatsoever between syntactic and semantic processors. Secondly, once a context-free parser has done its task, there may remain issues, such as checking symbol declarations against use, that are purely syntactic, but are not expressible by a context-free grammar. By letting the parser produce an explicit syntax tree, we are at liberty to process that tree further before submitting it to the semantic processor. In fact, we have implemented the non-context-free parts of syntactic analysis in the equation interpreter itself by equational programs that transform the abstract syntax after context-free parsing and before semantic processing. Systematic encodings of notation, such as Currying (transforming f(a,b,c) into apply(apply(apply(f,a),b),c)) may be implemented at this level.

Last, and perhaps most important in the long run, the use of an explicit abstract syntax allows applications of the system to develop far beyond the simple context of a user who types in a program, preprocesses it, types in an input, and awaits the results at his terminal. Many future applications of our interpreter may involve input terms, and even equational programs, that are themselves produced automatically by other programs, and the outputs may often be subject to other processing before, or instead of, being displayed. The very syntactic sugar that makes program and input entry easier for a human, makes it harder to produce automatically. Simply by omitting the syntactic pre- and post-processors when appropriate, we may build useful systems containing equational programs, and the communication within these systems need not deal with the inefficiencies and notational problems (especially quoting conventions) of the humanly readable syntax. We have already taken advantage of this feature, by omitting the pre- and post-processing steps from the equational programs that do syntactic analysis of equational programs. A more important use of this feature to extend the usefulness of equational programming is described below.

Although equational programs require substantial translation to be executed on conventional machinery, our current language is low level in the sense that no facilities are provided for organizing or modularizing large programs. The implementation of a high-level approach to equational programming should include the ability to combine separately written equational programs into larger ones, in a semantically meaningful, rather than purely lexical, way. Combining forms such as those described by Burstall and Goguen[16] should provide a good starting point for development of higher-level techniques in equational programming. We expect to implement such combining forms by equational programs that transform the abstract syntax of other equational programs. Once we have chosen a pleasant mechanism for resolving name clashes, this capability will be integrated into the system between the front end and the semantic part of the equation preprocessor.

The considerations above, along with the separate preprocessing step for pattern-matching, lead naturally to the system configuration shown in Figure 7. Communication between modules is always by UNIX text files.

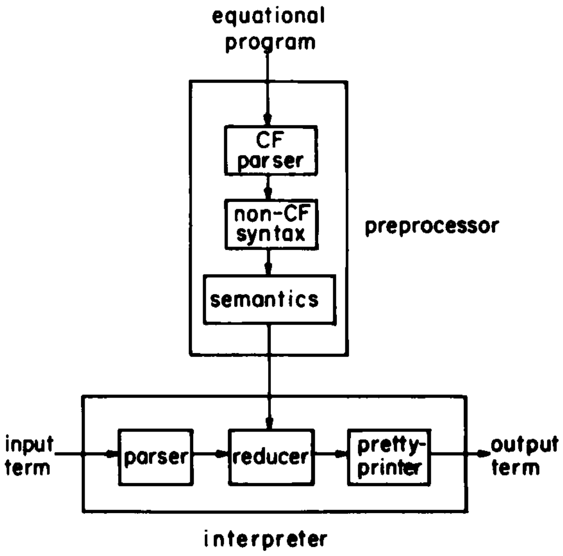

Figure 7.

EXPERIENCE

In Reference 2, we reported our experiences with an earlier version of the system. Briefly, we concluded that the bottom-up matching strategy is extremely fast, permitting reductions at very high rates. Since then we have conducted two major experiments in graduate seminars.

The purpose of the first experiment was to evaluate the practical performance of the various pattern matching algorithms proposed in Reference 12. We found that the top-down method with counter co-ordination is inferior to the other two methods, because it is slightly slower in detecting matches and requires more processing after reductions to maintain matching information. In particular, the matching time is proportional to the number of patterns to be matched. Since we wish to encourage writing many small equations, the large number of resulting patterns is noticeable in the performance. The top-down method with bit-string co-ordination performed better in detecting matches and update processing, but its match time also increased in proportion to the number of patterns matched. The perceived performance differential is probably due to the smaller locality in which update processing has to be performed.

Top-down matching with bit-string co-ordination did not offer a clear advantage over the bottom-up method, despite its inexpensive preprocessing. Bottom-up matching has more expensive preprocessing and requires tables, which can be fairly large, to direct the matching algorithms. However, it affords better diagnostics and is fastest in locating matches and update processing. This comparative appraisal of the bottom-up technique is corroborated by the work of Wilhelm,[5] who has used this matching method extensively in his equational approach to compiler writing. In Wilhelm's experience (as in ours), the patterns that give rise to poor preprocessing times do not normally arise in applications. Moreover, there are heuristics to reduce space demands and compress the tables needed by the matching algorithm, resulting in acceptable sizes.

In the case of left-sequential equations, the new method derived from string matching is, in our opinion, the best choice, since it is as fast as the bottom-up approach at run-time and usually as space efficient as the top-down methods.

A second experiment investigated the suitability of equational programs for writing compilers for procedural languages. We chose Pascal as a compromise between source language complexity and the time constraints in a class-room situation. Results indicate both pros and cons of writing compilers with equations: on the one hand, for attribute maintenance, equations are not especially convenient, but on the other, the equational compiler was concise and the students felt that their programming effort was less error-prone. The project also pointed out a need for a structured specification technique similar to the ones advocated in Reference 16, e.g. 'derive', which allow a single, common specification of subtasks whose equations differ only in inessential ways.

AVOIDING REPEATED EVALUATION OF SUBTERMS

Outermost evaluation, although avoiding evaluation of subterms that are irrelevant to the final result, allows unnecessary duplication of relevant subterms. Whenever a variable appears more than once on the right-hand side of an equation, innermost evaluation would evaluate the term substituted for that variable once, before applying the equation in question. Outermost evaluation appears to create multiple copies of such a term, which apparently will be evaluated separately. It is easy to avoid this particular duplication of effort by implementing multiple instances of the same variable by multiple pointers to the same subterm. Such collapsing, of course, makes future implementation of parallel reductions more difficult, because several processes may simultaneously occupy the same subterm.

We have gone further in avoiding repetition. Whenever an instance of a right-hand side is created, the newly created nodes are hashed, and coalesced with any existing identical nodes. This innovation was introduced as an optimization in a prototype version of the interpreter. As a result, if a subterm T is created repeatedly, it is still evaluated only once. Further improvements are possible. If, as a result of reduction of one of its proper subterms, T *becomes* identical with an existing subterm, we do not detect such an identity. To do so would require restructuring of the hash table, and a noticeable extra overhead. Such a dynamic detection of identical subterms would lead to an implementation of the *directed congruence closure* algorithm of Chew,[17] and is left to future work. The current level of identity detection already has interesting consequences for programming.

Automatic dynamic programming

Dynamic programming may be viewed as a general technique for transforming an inefficient recursive program into a more efficient, iterative one that stores some portion of the graph of the recursively defined function in a data structure, in order to avoid recomputation of function values. In a typical application of dynamic programming, the programmer must specify how the graph of the function is to be stored, as well as the order in which the graph is to be computed. The latter task may be handled automatically by the equation interpreter.

We illustrate this automation on equations to solve the optimal matrix multiplication problem.[18] The input to the problem is a list of integers $(d_0 \ldots d_m), m \geq 1$, representing a sequence $M_1, \ldots, M_m$ of matrices of dimensions $d_0 \times d_1, d_1 \times d_2, \ldots d_{m-1} \times d_m$, respectively. The problem is to find the cost of the cheapest order for multiplying such matrices, assuming that multiplication of an $i \times j$ by a $j \times k$ matrix costs $i*j*k$. There is an obvious recursive solution given by

$$\text{cost}[(d_0 \ldots d_n)] = \min \{\text{cost}[(d_0 \ldots d_i)] + \text{cost}[(d_i \ldots d_n)] + d_0 * d_i * d_n \mid 0<i<n\}$$
$$\text{cost}[(d_0 d_1)] = 0$$

This recursive solution, implemented directly, requires exponential time, because it recomputes the same values of the cost function many times. Dynamic programming achieves a polynomial solution by producing the graph of the cost function as a static data structure, into which each value is stored only once, but inspected repeatedly. Instead of the conventional approach of defining only a small finite part of the graph of the cost function, we define the infinite graph, and the outermost evaluation strategy of the equation interpreter guarantees that only the relevant part of the graph is actually computed, and in the right order. The more conventional solution of this problem requires the programmer to specify just the right finite portion of the graph of cost to compute, and the precise order of its computation. The infinite graph, called costgraph, has the following structure:

```
costgraph[()] =
        (0
               (cost[(1)]
                      (cost[(1  1)]
                             (cost[(1  1  1 )]...)(cost[(1  1  2)]...)...)
                      (cost[(1  2)]
                             (cost[(1  2  1)]...)(cost[(1  2  2)]...)...
                      ...)
               (cost[(2)]
                      (cost[(2  1)]
                             (cost[(2  1  1)]...)...)
                      ...)
               ...)
```

That is, cost[$(d_0 \ldots d_m)$] is the first element of the list which is element $d_m + 1$ of element $d_{m-1} + 1$ of ... element $d_0 + 1$ of costgraph[()].cost[(i)] is always 0, but inclusion of these 0s simplifies the structure of costgraph . costgraph[a], for $a \neq ()$ is the fragment of costgraph[()] whose indexes are all prefixed by a.

The following equational program solves the optimal matrix multiplication problem, using LISP notation:

```
Symbols
:operators directly related to the computation of cost
    cost:1; costgraph: 1; costrow: 2; reccost: 1; subcosts:2;

:list-manipulation, logical and arithmetic operators
    cons: 2; nil: 0;
    min: 1; index: 2; length: 1; element: 2; firstn: 2;
    first: 1; tail: 1; last: 1;
    aftern: 2;
    addend: 2;
    cond: 3;
    equ: 2; less: 2;
    add: 2; subtract: 2; multiply: 2;
    include integer_numerals, truth_values.

For all a, b, i, j, k, x, y:
    cost[a] =index [a; costgraph[()]];
: costgraph [a] is the infinite graph of the cost function for
: arguments starting with the prefix a.
    costgraph[a] =(reccost[a] . costrow[a; 1]);

: costrow[a; i] is the infinite list
: (costgraph[ai] costgraph[ai+1] ...)
: where ai is a with i added on at the end.
    costrow [a; i] =(costgraph[addend[a; i]] . costrow[a; add[i; 1]]);

: reccost[a] has the same value as cost[a], but is defined
: by the recursive equations from the discussion preceding the program
    reccost[(i j)] =0; reccost [(i)] =0; reccost[()] =0;
    reccost[(i j . a)] =min[subcosts[(i j . a); length[a]]]
                        where a is (k . b) end where;

: subcosts[a; i] is a finite list of the recursively computed
: costs of (d0 ... dm), fixing the last index removed at
:i, i- 1, ... 1.
    subcosts [a; 1] =cond[equ [i; 0]; ();
                          (add[add[
                                   cost[firstn[add[i; 1]; a]];
                                   cost[aftern[i; a]]];
                                   multiply [multiply[
                                             first [a];
                                             element[add[i;1]; a]];
                                             last [a]]]
                           subcosts[a; add[i; -1]])];

: Definitions of list-manipulation operators,
: logical and arithmetical operators.
    min[(i)] =i;
```

```
min[(i . a)] =cond less[i; min[a]]; i; min[a]]
              where a is (k . b) end where;
index[(); (x . b)] =x;
index[(i . a); x] =index[a; element[add[i; 1]; x]];
length [()] =0;
length[(x . a)] =add[length[a]; 1];
element[i; (x . a)] =cond[equi[i; 1]; x; element[subtract[i; 1]; a]];
firstn[i; a] =cond[equ[i; 0];(); (first[a] . firstn[subtract[i; 1]; tail[a]])];
first[(x . a)] =x; tail[(x . a)] =a;
aftern[i; a] =cond[equ[i; 0]; a; aftern[subtract[i; 1]; tail[a]]];
last[(x)] =x;
last[(x y . a)] =last[(y . a)];
addend[(); y] =(y);
addend[(x . a); y] =(x . addend[a; y]);
cond[true; x; y] =x; cond[false; x; y] =y;
include addint, equint, subint, multint.
```

While understanding the mapping of the graph of the function cost onto the structure costgraph[] is somewhat tedious, such tediousness might be ameliorated by a specialized notation for such problems, without losing the advantage of automatic discovery of the correct order of computation.

The efficiency (but not the correctness) of the program above depends on the fact that all instances of costgraph[()] will be detected and coalesced by the interpreter. A future implementation of the dynamic identity detection embodied in the directed congruence closure algorithm[17] would allow the same efficiency to be achieved by the straightforward recursive program.

ACKNOWLEDGEMENTS

This research was supported in part by the National Science Foundation under grants MCS 78–01812, and MCS 82–17996, and by the National Security Agency under contract 84H0006.

REFERENCES

1. M. J. O'Donnell, *Computing in systems Described by Equations, Lecture Notes in Computer Science*, **58**, Springer-Verlag, 1977.
2. C. Hoffmann and M. J. O'Donnell, 'Programming with equations', *ACM TOPLAS*, January, 83–112 (1982).
3. R. Kowalski, 'Algorithm = logic + control', *CACM*, **22**, (7), 424–436 (1979).
4. R. Burstall, D. MacQueen and D. Sannella, 'HOPE: an experimental applicative language', *Internal Report CSR-62-80*, University of Edinburgh, 1980.
5. I. Glasner, U. Möncke and R. Wilhelm, 'OPTRAN, a language for the specification of program transformations', *Informatik-Fachberichte*, Springer-Verlag, 1980, pp. 125–142.
6. U. Möncke, 'An incremental and decremental generator for tree anaysers', *Bericht Nr. A 80/3*, Fachber. Informatik Univ. des Saarlandes, Saarbrücken, April 1980.
7. P. Henderson and J. H. Morris, 'A lazy evaluator', *3rd ACM Symposium on Principles of Programming Languages*, 1976, pp. 95–103.

8. D. Friedman and D. Wise, 'Cons should not evaluate its arguments', *3rd International Colloquium on Automata, Languages and Programming*, Edinburgh, Edinburgh University Press, 1976, pp. 257–284.
9. G. Khan and D. B. MacQueen, 'Coroutines and networks of parallel processes', in B. Gilchrist (ed.), *Information Processing 77,* North-Holland, 1977, pp. 993–998.
10. A. Aho and M. Corasick, 'Efficient string matching: an aid to bibliographic search', *CACM*, **18**, (6), 333–343 (1975).
11. D. Knuth, J. Morris and V. Pratt, 'Fast pattern matching in strings', *SIAM J. Comp.*, **6**, 2, 323–350 (1977).
12. C. Hoffmann and M. J. O'Donnell, 'Pattern matching in trees', *JACM*, January, 68–95 (1982).
13. G. Huet and J.-J. Lévy, 'Computations in non-ambiguous linear term rewriting systems', *IRIA Technical Report #359,* 1979.
14. C. Hoffmann and M. J. O'Donnell, 'Interpreter generation using tree pattern matching', *6th Annual Symposium on Principles of Programming Languages*, 1979, pp. 169–179.
15. M. J. O'Donnell, *Equational Logic as a Programming Language*, M.I.T. Press, Cambridge, Mass., 1985.
16. R. M. Burstall and J. A. Goguen, 'Putting theories together to make specifications', *5th International Joint Conference on Artificial Intelligence,* Cambridge, Mass., 1965.
17. L. P. Chew, 'An improved algorithm for computing with equations', *21st Annual Symposium on Foundations of Computer Science,* 1980, pp. 108–117.
18. A. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.