Implementation of an Interpreter for Abstract Equations

Christoph M. Hoffmann
Michael J. O'Donnell

Purdue University
The Johns Hopkins University

ABSTRACT

This paper summarizes a project, introduced in [HO79, HO82b], 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

(1) strict adherence to semantics based on logical consequences;

(2) "lazy" (outermost) evaluation applied uniformly;

(3) an implementation based on table-driven pattern matching, with no run-time penalty for large sets of equations;

(4) strict separation of syntactic and semantic processing, so that different syntaxes may be used for different problems.

1. Introduction

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, and

(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 involving those symbols and variables. The *meaning* of a program is completely described by the following:

This research was supported in part by the National Science Foundation under grants MCS 78-01812, and MCS 82-17996.

Definition 1.1
A term containing no instance of a left-hand side of an equation 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.

1.1 Syntax of the Equational Programming Language

Input to the equation interpreter is in the following form:

Symbols
symdes$_1$;
symdes$_2$;
symdes$_m$.

For all *var*$_1$,*var*$_2$, · · · *var*$_n$:
equn$_1$;
equn$_2$;
equn$_p$.

Symbol descriptors 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. A *symbol descriptor* (*symdes*) is either of the form

sym$_1$,*sym*$_2$, · · · *sym*$_m$: *arity* $m \geq 1$

or of the form

include *symclass*$_1$, · · · *symclass*$_n$ $n \geq 1$

Syntactically, *symbols* and *symbol classes* (*symclass*) 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**. Symbols that have been explicitly declared in the **Symbols** section are called *literal symbols*, to distinguish them from members of the predefined classes.

Variables (*var*) are identifiers, of the same sort as symbols. An *equation* (*equn*) is either of the form

term$_1$=*term*$_2$

of the form

term$_1$=*term*$_2$ **where** *qualification* **end where**

or of the form

include *equclass*$_1$, $\cdots$ *equclass*$_m$

Equation classes (*equclass*) are identifiers indicating the inclusion of a large number of predefined equations. These 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.

A *qualification* is of the form

qualitem$_1$, $\cdots$ *qualitem*$_m$ $\quad m \geq 1$

and *qualitems* are of the forms

var **is** *qualterm*

var$_1$, $\cdots$ *var*$_p$ **are** *qualterm*

and *qualterms* are of the forms

in *predefined_symbol_class*

term

qualterm **where** *qualification* **end where**

either *qualterm*$_1$ **or** $\cdots$ *qualterm*$_r$ **end or**

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. In this paper, we will use two different term syntaxes: 1) the standard mathematical notation in which function application is denoted f(a,b), and 2) 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 *yacc* and *lex* programs for other syntaxes as needed.

The following example illustrates all of the constructs described above, using LISP notation.

Example 1.1

Symbols

```
: List constructors
  cons: 2;  nil: 0;

: Standard arithmetic operators
  add: 2;

: nonils[x] is the list containing all of the nonnil
: 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. (x . y) is merely an abbreviation for cons[x;y]. "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. Presently, 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.

2. Every variable appearing on the right side of an equation must also appear on the left. For instance, $f(x)=y$ is prohibited.

3. Two different left sides may not match the same expression. So the pair of equations

   $g(0,x)=0; \quad g(x,1)=1$

   is prohibited.

4. When two (not necessarily different) left-hand sides match two different parts of the same expression, the two parts must not overlap. E.g., the pair of equations

   $first(pred(x))=predfunc; \quad pred(succ(x))=x$

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

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

   $f(g(x,a),y)=0; \quad g(b,c)=1$

   is prohibited, since after scanning $f(g$ it 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.

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 in Section 4.

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

1.2 Contents

The remainder of the paper discusses the major ideas of the project and its execution, as well as our experience with using the system. Section 2 gives the history of the project and compares the equation interpreter to PROLOG and HOPE [BMS80]. Sections 3-5 discuss the most important

design decisions and their consequences. Section 6 discusses our experience in using the system, and Section 7 demonstrates one of the programming advantages of the outermost (lazy) evaluation strategy.

2. History of the Project

The logical foundations of the project, results concerning uniqueness of normal forms and correct orders of evaluation, come from [O'D77]. From 1978 to 1981, work toward an implementation, both in theoretical development of algorithms and in the building of prototype systems, was performed by the authors, with programming aid from two students, Giovanni Sacco and Paul Golick [HO82b]. Final preparation for distribution was done by O'Donnell in 1982-1983, while Hoffmann ported an earlier version to Kiel, Germany, and led two projects involving alternative approaches to pattern-matching in the interpreter, and use of the interpreter to define interpreters and compilers for PASCAL.

Among other nonprocedural programming languages, the one which is very similar in flavor to the equation interpreter is PROLOG [Ko79]. 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 back tracking. 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 much simpler, our equation interpreter can produce all of the logically entailed outputs without making unacceptable resource demands.

Another language processor similar syntactically to the equation interpreter is HOPE [BMS80], 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 lefthand 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., [GMW80, Mon80]. 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 without sacrificing semantic rigor.

3. 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. Nearly all programming languages evaluate conditionals in this way, and like treatment of the LISP function *cons* has been proposed in [HM76, FW76, BMS80], but we know of no other language processor which implements lazy evaluation in all cases. Kahn and McQueen [KM77] 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. Lazy evaluation has advantages for the user, allowing straightforward use of a certain type of parallel programming. [FW76, HM76] demonstrated some of these advantages in the case of LISP. Section 7 shows how lazy evaluation automatically performs one of the design tasks in dynamic programming which otherwise must be explicitly programmed.

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

$$\text{add}(0,x) = x$$
$$\text{add}(s(y),x) = s(\text{add}(y,x))$$

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 may 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 in Section 4, single-precision arithmetic does not have to pay the overhead of checking whether the inputs are single precision.

The current version of the equation interpreter allows use of the 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, so that the even the user who has not written multiprecision equations sees correct, but possibly less helpful, output. Of course, the equations for arithmetic and other natural primitives should be written once and saved away 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 in Section 5. We have chosen to await results in the more general area, rather than to perform an *ad hoc* extension for primitive operations.

4. The Importance of Pattern-Matching Algorithms

4.1 Motivation

In order that programming with equations be really different from 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 lookup.

For multiple-pattern string matching, the Aho-Corasick generalization of the Knuth-Morris-Pratt algorithm [AC75, KMP77] solves a problem closely analogous to ours. Extension of these string-matching techniques to terms (equivalently, trees) was treated separately in [HO82a]. 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.

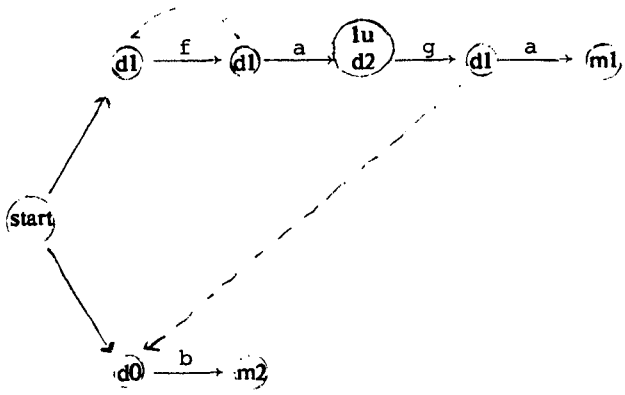
4.2 A Specialized Pattern-Matching Algorithm

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.

Tree patterns are flattened into preorder strings, omitting variables. The Aho-Corasick algorithm [AC75] 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 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 $g(x,b)$ generate the strings $ffaga$ and gb, and the automaton given in Figure 4.1. The automaton cannot be annotated consistently if conflicting moves are associated with the same state. 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



forward edge
failure edge
failure edges not shown all lead to the start state
lu means move up one level in the tree
d1 means move down to son number 1
m1 means a match of pattern number 1

Figure 4.1

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) of Section 1.1. 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 $\alpha=ffa$, $\beta=g$, $\gamma=a$, $\delta=b$. That is, after scanning $ffag$, 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 those two annotations.

4.3 Completeness of the Pattern-Matching Algorithm

The restriction imposed on equations by the pattern-matching strategy above may be justified in a fashion similar to the justification of deterministic 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. In order to define sequentiality, we need some special terms for discussing computation steps in the interpreter. All of the discussion in this subsection refers to an arbitrarily given set of equations.

Definition 4.1

A set of equations is *regular* if it satisfies restrictions 1-4 of Section 1.1 (but not necessarily restriction 5).

A *context* is a term built from the constants and operators in the given set of equations, as well as the new constant symbol ω.

An *instance* of a context C is any term or context S resulting from the replacement of one or more occurrences of ω in C.

A *left context* is a context C such that there is a path from the root of C to a leaf, with no occurrences of ω on or to the left of the path, and nothing but ωs to the right of the path.

A *left-traversal context* is a pair $<C,I>$, where C is a left

context, and l is a node on the path dividing ωs from other symbols in C.
A *redex* is an occurrence of an instance of a left-hand side of an equation (letting each variable occurrence be treated as an ω).
A term S *l-reduces* to a term T if S may be transformed into T by replacing redexes by arbitrarily chosen terms.
A redex R in a term S is *essential* if there is no way to l-reduce S to normal form without reducing R.
A term S is *root stable* if there is no redex T such that S l-reduces to T.
A redex R in a term S is *root essential* to S if there is no way to l-reduce S to a redex or a root stable term without reducing R.

A context represents the information known about a term after a partial traversal. The ωs stand for unknown portions. A left-traversal context contains exactly the part of a term that has been seen by a depth-first left traversal that has progressed to the specified node. l-reduction is the best approximation to reduction that may be derived without knowing the right-hand sides of equations.

In the process of reducing a term by outermost reductions, our short-term goal is to make the whole term into a redex. If that is impossible, then the term is root stable, and may be cut down into independent subproblems by removing the root.

Definition 4.2
A set of equations is *strongly left-sequential* if there is a set of left-traversal contexts L such that the following conditions hold:
1. For all $<C,l>$ in L, the subtree of C rooted at l is a redex.
2. For all $<C,l>$ in L, S an instance of C, l is essential to S.
3. For all left-traversal contexts $<C,l>$ not in L, S an instance of C, l is not root-essential to S.
4. Every term is either root stable or an instance of a left context in L.

In a strongly left-sequential system, we may reduce a term by traversing it in preorder to the left. Whenever a redex is reached, the left-traversal context specifying that redex is checked for membership in L. If the left context is in L, the redex is reduced. Otherwise, the traversal continues. When no left context in L is found, the term must be root stable, so the root may be removed, and the resulting subterms processed independently. (1) and (2) guarantee that only essential redexes are reduced. (3) guarantees that no root-essential redex is skipped. (4) guarantees that the reduction never hits a dead end by failing to choose any redex. The analogous property to strong left-sequentiality, using reduction instead of l-reduction, is undecidable. Notice that strong left-sequentiality depends only on the left-hand sides of equations, not on the right-hand sides.

Strong left-sequentiality is a special case of the strong sequentiality defined by Huet and Lévy [HL79], who give a thorough technical treatment of these concepts. Huet and Lévy have a pattern-matching algorithm that is much more general than ours, but its practical implementation has not yet been studied. We expect that our algorithm will continue to be useful because of its simplicity, even when implementations of the Huet-Lévy method are available to cover their wider class of sequential systems.

Strongly left-sequential sets of equations are intended to include all of those systems that one might reasonably expect to process by scanning from left to right. Notice that definition 4.3 does not require L to be decidable. Also, a strongly left-sequential system may not necessarily be processed by leftmost-outermost evaluation. Rather than requiring us to reduce a leftmost redex, definition 4.3 merely requires us to *decide* whether or not to reduce a redex in the left part of a term, before looking to the right. Every redex that is reduced must be essential to finding a normal form. When the procedure decides not to reduce a particular redex, it is only allowed to reconsider that choice after producing a root-stable term and breaking the problem into smaller pieces. While strongly left-sequential systems are defined to allow a full depth-first traversal of the term being reduced, the algorithm of Section 4.2 avoids searching to the full depth of the term in many cases by recognizing that certain subterms are irrelevant to choosing the next step.

Theorem 4.1
The pattern-matching algorithm of Section 4.2 succeeds (i.e., generates no conflicts) if and only if the input patterns are left-hand sides of a regular and strongly left-sequential set of equations.
Proof sketch:
($\Rightarrow$) If the pattern matching-automaton is built with no conflicts, then L may be defined to be the set of all left-traversal contexts $<C,l>$ such that l is the root of a redex in C, and l is visited by the automaton, when started at the root of C.

($\Leftarrow$) If a conflict is found in the pattern-matching automaton, then there are two flattened preorder strings $\alpha\beta\gamma$ and $\beta\delta$ derived from the patterns, with conflicting tree moves at from β to γ and from β to δ. Without loss of generality, assume that there are no such conflicts within the two occurrences of β. $\alpha\beta$, with its associated tree moves, defines a context C, which is the smallest left context allowing the traversal specified by $\alpha\beta$. β defines a smaller left-traversal context D in the same way. D is contained as a subterm in C, in such a way that the last nodes visited in the two traversals coincide. If one or both of γ,δ is empty, then C demonstrates a violation of restriction (4) or (3), respectively. So, assume that γ,δ are not empty, and the annotations at the ends of the βs are both tree moves.

Consider the two positions to the right of C specified by the two conflicting traversal directions for $\alpha\beta$ and β. Expand C to E by filling in the leftmost of these two positions with an arbitrary redex, and let n be the root of this added redex. Let equ_1 be the equation associated with whichever of $\alpha\beta\gamma$, $\beta\delta$ directed traversal toward this leftmost position, and let equ_2 be the equation associated with the remaining one of $\alpha\beta\gamma$, $\beta\delta$. $<E,n>$ cannot be chosen in L, because there is an instance S of E in which a redex occurs above n matching the left-hand side of equ_2, and S may be l-reduced to normal form at this redex, without reducing the redex at n in E. $<E,n>$ cannot be omitted from L, because there is another instance T of E in which everything but n matches the redex associated with equ_1, and n is therefore root-essential to T.
□

For example, the pair of equation left-hand sides $f(g(x,a),y)$ and $g(b,c)$ have the preorder strings fga and gbc. A conflict exists with $\alpha=f$, $\beta=g$, $\gamma=a$, $\delta=c$. The first equation directs the traversal down edge 2 after seeing fg, and the second equation directs it down edge 1. The conflicting prefixes fg and g produce the context $f(g(\omega,\omega),\omega)$. The context above is expanded to the left-traversal context consisting of $f(g(g(b,c),\omega),\omega)$ with the root of $g(b,c)$ specified. This left-

traversal context cannot be chosen in I. (i.e., it is not safe to reduce the redex $g(b,c)$ in this case), because the leftmost ω could be filled in with a to produce $f(g(g(b,c),a),\omega)$, which is a redex of the form $f(g(x,a),y)$, and can be I-reduced to normal form in one step, ignoring the smaller redex $g(b,c)$. But, this left-traversal context may not be omitted from I. (i.e., it is not safe to omit reducing $g(b,c)$), because the leftmost ω may also be filled in with c to produce $f(g(g(b,c),c),\omega)$, and reduction of $g(b,c)$ is essential to get a normal form or a root-stable term.

4.4 Interpreting Nonsequential 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 of Section 4.2 may be extended to handle nonsequential systems by annotating each state in the automaton with a nonempty 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 (else work will be duplicated), and a process must be killed whenever a second process creates a redex containing the first one (else 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 very small time and space overhead. Even when such an implementation is accomplished, sequential algorithms such as ours and Huet and Lévy's will be useful because they can avoid the overhead of the parallel methods.

5. Separation of Syntactic Processing From Semantics

One of the main problems in making the equation interpreter useful to a human programmer, is the syntactic form of the terms written within equations, and those presented for reduction to normal form. 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 hairy 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 notation, and we know of none that is universally acceptable. So, we decided to communicate with the equation interpreter through a number of different front ends, stored in a standard library. A user may, of course, use his or her 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 a 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 interpreter input, but the (much easier) pretty-printers are written separately. While a program to generate parsers and unparsers (pretty-printers) from the same grammar would be very nice, we prefer to await the availability of grammar-driven structure editors, with which the only syntactic transformation required will be the pretty-printing.

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. Second, once a context-free parser has done its task, there may remain issues, such as checking symbol declarations against use, that are purely syntactic (in spite of compiler-writer's jargon), 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 postprocessors 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 postprocessing 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 very 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 [BG65] 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 is integrated into the system between the front end and the semantic part of the equation preprocessor.

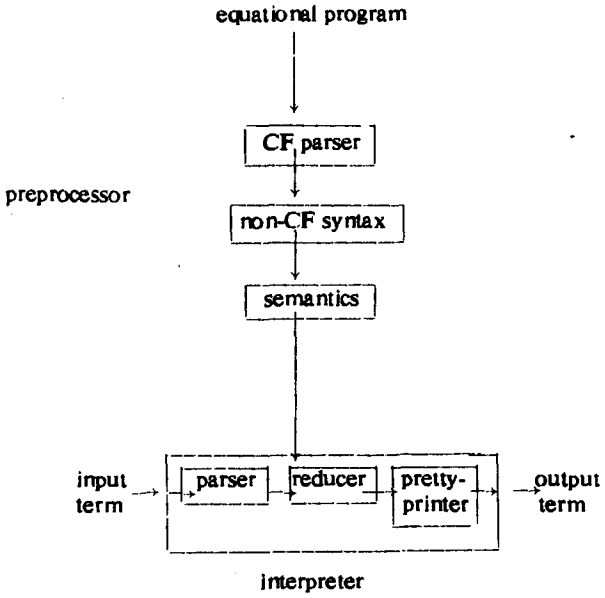


Figure 5.1

The considerations above, along with the separate preprocessing step for pattern matching, lead naturally to the system configuration shown in Figure 5.1. Communication between modules is always by UNIX text files.

6. Experience with the System

In [HO82b], 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 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 [HO82a]. We found that the top-down method with counter coordination 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 bitstring coordination performed better in detecting matches and update processing, but its match time also increases 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 bitstring coordination did not offer a clear advantage over the bottom-up method, despite its cheap preprocessing. Bottom-up matching has more expensive preprocessing and requires tables to direct the matching algorithms which can be fairly large, 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 (e.g., [GMW80]), who has used this matching method extensively in his equational approach to compiler writing. In Wilhelm's experience (as in ours), the patterns which 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, as 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 was to investigate the suitability of equational programs for writing compilers for procedural languages. We chose PASCAL as 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 very concise and the students felt that their programming of it was much less error-prone. The project also pointed out a need for a structured specification technique similar to the ones advocated in [BG65] (e.g., "derive"), which allow a single, common specification of subtasks whose equations differ only in inessential ways.

7. Avoiding Repeated Evaluation of Subterms.

Outermost evaluation, while 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 (Section 4.4) more difficult, because several processes may simultaneously occupy the same subterm.

We have gone farther 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 by Paul Golick 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 Paul Chew [ChL80], and is left to future work. The current level of identity detection already has interesting consequences for programming.

7.1 Automatic Dynamic Programming.

Dynamic programming may be viewed as a general technique for transforming an inefficient recursive program into a more efficient, iterative one which 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 of [AHU74]. The input to the problem is a list of integers $(d_0 \cdots d_m)\ m \geqslant 1$, representing a sequence $M_1, \cdots M_m$ of matrices of dimensions $d_0 \times d_1, d_1 \times d_2, \cdots d_{m-1} \times d_m$ 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

$$cost[(d_0 \cdots d_m)] =$$
$$\min\{cost[(d_0 \cdots d_i)] + cost[(d_i \cdots d_m)] + d_0 * d_i * d_m \mid 0 < i \leqslant m\}$$
$$cost[(d_0\ d_1)] = 0$$

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 following equational program solves the optimal matrix multiplication problem, using LISP notation. Lines beginning with colon are comments.

```
: In the following equations, the function cost is represented
: by an infinite-dimensional infinite list giving the graph of
: the function:
:
:   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[(d0 ... dm)] is the first element of the list
: which is element dm + 1 of element dm-1 + 1 of ...
: element d0 + 1 of costgraph[()]. cost[(i)] is always 0, but
: inclusion of these 0s simplifies the structure of costgraph.
: costgraph[a], for a<>() is the fragment of costgraph[()]
: whose indexes are all prefixed by a.

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;
  aftern: 2;
  last: 1;
  addend: 2;
  cond: 3;
  add: 2;
  equ: 2;
  less: 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 header.

  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; i] = 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[equ[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 greatly 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 [ChL80] would allow the same efficiency to be achieved by the straightforward recursive program.

Bibliography and References

AC75 Aho, A., and M. Corasick, Efficient String Matching: an Aid to Bibliographic Search, *CACM* 18:6 (1975) 333-343

AHU74 Aho, A., J. E. Hopcroft, J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

AU72 Aho, A. and J. Ullman, *The Theory of Parsing, Translation, and Compiling, Volume I: Parsing*, Prentice-Hall, 1972.

AW76 Ashcroft, E. and W. Wadge, Lucid – A Formal System for Writing and Proving Programs. *SIAM Journal on Computing* 5:3, 1976, 336:354.

AW77 Ashcroft, E. and W. Wadge, Lucid, a Nonprocedural Language with Iteration, *CACM* 20:7, 1977, 519-526.

Ba74 Backus, J. Programming Language Semantics and Closed Applicative Languages. ACM Symposium on Principles of Programming Languages, 1974, 71-86.

Ba78 Backus, J. Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs, *CACM* 21:8, 1978, 613-641.

BB79 Bauer, F. L., M. Broy, R. Gnatz, w. Hesse, B. Krieg-Bruckner, H. Partsch, P. Pepper, H. Wossner. Towards a Wide Spectrum Language to Support Program Development by Transformations. *Program Construction: International Summer School, Lecture Notes in Computer Science* v. 69, Springer-Verlag, 1979, 543-552.

BL77 Bérry, G. and Lévy, J. J. Minimal and Optimal Computations of Recursive Programs. 4th ACM Symposium on Principles of Programming Languages, 1977, 215-226.

BL79 Bérry, G. and Lévy, J. J. Letter to the Editor, SIGACT News, v. 11, no. 1, Summer 1979, 3-4.

Bj72 Bjorner, D. Finite State Tree Computations (Part I). IBM Research Technical Report RJ 1053 (#17598), 1972.

Br76 Bruynooghe, M., An Interpreter for Predicate Logic Programs Part I, Report CW10, Applied Mathematics and Programming Division, Katholieke Universiteit, Leuven, Belgium, 1976.

BG65 Burstall, R. M. and Goguen, J. A. Putting Theories Together to Make Specifications. 5th International Joint Conference on Artificial Intelligence, Cambridge, Mass., 1965.

BMS80 Burstall, R., MacQueen, D., Sannella, D. HOPE: An Experimental Applicative Language. Internal Report CSR-62-80, University of Edinburgh, 1980.

CaJ72 Cadiou, J., Recursive Definitions of Partial Functions and Their Computations, Ph.D. Dissertation, Computer Science Dept., Stanford University, 1972.

CaT76 Cargill, T., Deterministic Operational Semantics for Lucid, Research Report CS-76-19, University of Waterloo, 1976.

ChL80 Chew, L. P. An Improved Algorithm for Computing With Equations. 21st Annual Symposium on Foundations of Computer Science, 1980, 108-117.

ChA41 Church, A. *The Calculi of Lambda-Conversion.* Princeton University Press, Princeton, New Jersey, 1941.

deB72 de Bruijn, N. G. Lambda Calculus Notation with Nameless Dummies, Nederl. Akad. Wetensch. Proc. Series A 75, 1972, 381-392.

CF58 Curry, H. B., and Feys, R., *Combinatory Logic* volume I. North-Holland, Amsterdam, 1958.

DS76 Downey, P. and R. Sethi, Correct Computation Rules for Recursive Languages. *SIAM Journal on Computing* 5:3, 1976, 378-401.

Fa77 Farah, M., Correct Compilation of a Useful Subset of Lucid, Ph.D. Dissertation, Department of Computer Science, University of Waterloo, 1977.

FW76 Friedman, D., and D. Wise, Cons should not evaluate its arguments, 3rd International Colloquium on Automata, Languages and Programming, Edinburgh, Edinburgh University Press, 1976, 257-284.

GMW80 Glasner, I., Möncke, U., and Wilhelm, R. OPTRAN, a language for the specification of program transformations *Informatik-Fachberichte,* Springer-Verlag 1980, 125-142

Go77 Goguen, J., Abstract Errors for Abstract Data Types, IFIP Working Conference on Formal Description of Programming Concepts, E. J. Neuhold, ed., North-Holland, 1977.

GS78 Guibas, L. and R. Sedgewick, A Dichromatic Framework for Balanced Trees, 19th Symposium on Foundations of Computer Science, 1978, 8-21.

GHM76 Guttag, J., E. Horowitz and D. Musser, Abstract Data Types and Software Validation, Information Science Research Report ISI/RR-76-48, University of Southern California, 1976.

HM76 Henderson, P., and J. H. Morris, A Lazy Evaluator, 3rd ACM Symposium on Principles of Programming Languages, 1976, 95-103.

Ho78 Hoffmann, C., Design and Correctness of a Compiler for a Nonprocedural Language, *Acta Informatica* 9, 1978, 217-241.

HO79 Hoffmann, C. and O'Donnell, M. J., Interpreter Generation Using Tree Pattern Matching, 6th Annual Symposium on Principles of Programming Languages, 1979, 169-179.

HO82a Hoffmann, C. and O'Donnell, M. J., Pattern Matching in Trees, *JACM.* January 1982, 68-95.

HO82b Hoffmann, C. and O'Donnell, M. J., Programming With Equations, *ACM TOPLAS.* January 1982, 83-112.

HL79 Huet, G. and J.-J. Lévy, Computations in Nonambiguous Linear Term Rewriting Systems, IRIA Technical Report #359, 1979.

Jo77 Johnson, S. D., An Interpretive Model for a Language Based on Suspended Construction, Technical Report #68, Dept. of Computer Science, Indiana University, 1977.

KM77 Kahn, G. and MacQueen, D. B. Coroutines and Networks of Parallel Processes, Information Processing 77, B. Gilchrist ed., North-Holland, 1977, 993-998.

KMP77 Knuth, D., J. Morris and V. Pratt, Fast Pattern Matching in Strings, *SIAM J. on Comp.* 6:2 (1977) 323-350

Kl80 Klop, J. W. Combinatory Reduction Systems, Ph. D. dissertation, Mathematisch Centrum, Amsterdam, 1980.

KB70 Knuth, D., and P. Bendix, Simple Word Problems in Universal Algebras. *Computational Problems in Abstract Algebra.* J. Leech, ed., Pergammon Press, Oxford, 1970, 263-297.

Ko79 Kowalski, R. Algorithm = Logic + Control. *CACM* 22:7, 1979, 424-436.

McC60 McCarthy, J., Recursive Functions of Symbolic Expressions and Their Computation by Machine, *CACM* 3:4, 1960, 184-195.

McI68 McIlroy, M. D., Coroutines, Internal report, Bell Telephone Laboratories, Murray Hill, New Jersey, May 1968.

Mon80 Möncke, U. An Incremental and Decremental Generator for Tree Analysers Bericht Nr. A 80/3, Fachber. Informatik Univ. des Saarlandes, Saarbrücken, April 1980

NO78 Nelson, G. and D. C. Oppen, A Simplifier Based on Efficient Decision Algorithms, 5th Annual ACM Symposium on Principles of Programming Languages, 1978, 141-150.

NO80 Nelson, G. and D. C. Oppen, Fast Decision Algorithms Based on Congruence Closure, *JACM* 27:2, 1980, 356-364.

O'D77 O'Donnell, M. J., *Computing in systems Described by Equations, Lecture Notes in Computer Science* v. 58, Springer-Verlag, 1977.

O'D79 O'Donnell, M. J. Letter to the Editor, SIGACT News, v. 11, no. 2, Fall 1979, p. 2.

RoG77 Roberts, G., An Implementation of PROLOG, M.S. Thesis, Dept. of Computer Science, University of Waterloo, 1977.

RoB73 Rosen, B. K., Tree Manipulation Systems and Church-Rosser Theorems, *JACM* 20:1, 1973, 160-187.

St77 Staples, J., A Class of Replacement Systems with Simple Optimality Theory, *Bulletin of the Australian Mathematical Society,* 17:3, 1977, 335-350.

St79 Staples, J. A Graph-Like Lambda Calculus For Which Leftmost-Outermost Reduction Is Optimal. *Graph Grammars and Their Application to Computer Science and Biology, Lecture Notes in Computer Science,* volume 73, V. Claus H. Ehrig, G. Rosenberg eds., Springer-Verlag, 1979.

St72 Stenlund, S. *Combinators, Lambda-Terms, and Proof Theory.* D. Reidel Publishing Company, Dordrecht, Holland, 1972.

Vu74 Vuillemin, J., Correct and Optimal Implementations of Recursion in a Simple Programming Language, *JCSS* 9:3, 1974, 332-354.

WaM76 Wand, M., First Order Identities as a Defining Language, Technical Report #29, Dept. of Computer Science, Indiana University, 1976.

WaD77 Warren, D., Implementing PROLOG, Research Reports #39, 40, Dept. of Artificial Intelligence, University of Edinburgh, 1977.