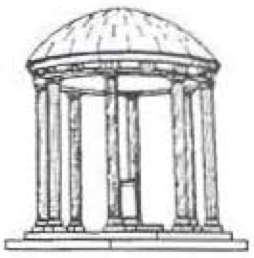
An Interpreter for EqL

TR87-037

August, 1987

Gopal Gupta

The University of North Carolina at Chapel Hill
Department of Computer Science
Sitterson Hall, 083A
Chapel Hill, NC 27599-3175



UNC is an Equal Opportunity/Affirmative Action Institution.

An Interpreter for EqL

by

Gopal Gupta

A thesis submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science.

Chapel Hill

Aug, 1987

Approved by:

[illegible]

Advisor

[illegible]

Reader

[illegible]

Reader

©1987
Gopal Gupta
ALL RIGHTS RESERVED

GOPAL GUPTA : **An Interpreter for EqL**† (Under the direction of Dr. BHARAT JAYARAMAN.)

Abstract

EqL is a general-purpose language that combines the capabilities of functional and logic programming languages. A program in EqL consists of a collection of conditional, pattern-directed rules, where the conditions are expressed as a conjunction of equations, and the patterns are terms built up of data-constructors and basic values. The computational paradigm in EqL is equation solving. In this paper, we describe EqL informally, giving examples illustrating the various features of the language: nondeterminism, logical variables, deferred evaluation of primitives, and user-defined constructors. We also describe the novel aspects of a sequential interpreter for EqL: compile-time flattening and re-ordering of equations; and run-time equation-delaying, last-equation optimization, and rule-indexing.

† The research has been supported by grant DCR-8603609 from the National Science Foundation.

To Deepa

Acknowledgements

First of all I would like to thank my advisor, Dr. Bharat Jayaraman, for his constant guidance and help throughout my work. He enthusiastically programmed in EqL and helped pinpoint the bugs in the interpreter. Most of the programs presented in this thesis have been written by him. This project could not have been completed without his help.

I would like to express my gratitude to the other members of my M.S. thesis committee, Dr. David Plaisted and Dr. J. Dean Brock, for consenting to be in the committee and for their comments.

I would also like to thank members of the SoftLab group, Sundar Varadarajan, Vikram Biyani and Alex Nelson for providing the expert help on Unix and C whenever I needed it.

I would also like to thank the Sun Corporation for their wonderful window environment and for dbxtool which made debugging C programs so easy and fun.

Thanks are also due to my parents, brothers, sisters, and friends for their help and encouragement all along.

Lastly, I would like to thank Mr. Ashok Kumar Singh, one of the greatest teachers I have known, for his encouragement and inspiration during my high school years.

Table of Contents

Chapter I. Introduction

The integration of functional and logic languages has received considerable interest recently, and a number of different approaches have been proposed [RS82, D83, F84, GM84, DP85, YS86]. EqL represents one such approach, unifying first-order functional and Horn-logic programming through the paradigm of *equational programming* [JS86, JSG86]. An EqL program is a collection of *conditional rewrite rules*, where the conditions are expressed as a set of *equations*, and the top-level goal to be solved is also a set of equations. It has been shown that this framework has the capabilities of first-order functional and Horn logic programming [J87, JSG86, JS86], with the following properties:

- pattern-directed rules;
- functional, rather than relational, notation;
- declarative semantics based on *complete set of solutions* [J87];
- operational semantics based on *object refinement* [J87];
- potential for parallel execution [JG86].

The computational paradigm underlying EqL is *equation solution*. This paradigm subsumes expression evaluation in a functional language because an expression e to be evaluated is simply viewed as an equation $v = e$, where v is some distinct variable. It also subsumes goal solution in logic languages because a goal g to be solved can also be viewed as an equation, namely, $g = true$.

In this thesis we illustrate EqL with examples drawn from the literature in functional and logic programming. We also describe the salient features of a sequential implementation of EqL, written in C. For the sake of efficiency, this sequential implementation searches for solutions depth-first with backtracking, similar to Prolog implementations [WPP77, H84]. Although depth-first search does sacrifice *completeness* in theory, we have not found this to be a serious limitation in practice.

Ultimately performance for such a language must be realized through parallelism—which would also facilitate a *complete* implementation. Because sequential machines are still in widespread use, we feel it is worthwhile investigating efficient sequential implementations. The current implementation has two phases:

1. *compilation*, in which the source code is transformed by *flattening* and *reordering* equations, in accordance with their semantics; and

2. *interpretation*, in which equations are solved using a seven-stack execution model.

The rest of this document is organized as follows: chapter II describes the data objects of EqL and the informal meaning of EqL rules. Chapter III describes the user interface of the EqL interpreter and its various features. Chapter IV presents examples of EqL programs for functional and logic programming, illustrating non-determinism, logical variables, and deferred evaluation of primitives. Chapters V and VI describe the implementation of the EqL interpreter. Chapter VII presents summary and conclusions.

Chapters II through IV can be used as a manual for the interpreter. Our convention throughout this document is to use type-writer font, e.g. `cons`, for EqL program text and italics, e.g. *equation*, for syntactic categories.

Chapter II. EqL: Language Features

In this chapter we describe the data objects in EqL and the informal meaning of rules.

II.1. Data Objects

The data objects in EqL are defined below:

(i) *Numbers*: The current implementation of EqL provides only integers, e.g. `10`, `-3999`, etc.

(ii) *Booleans*: `true`, `false`.

(iii) *Atoms*: Any identifier beginning with an upper-case letter or any sequence of characters enclosed within single quotes, e.g. `Apple`, `'also an atom'`, etc.

(iv) *Variables*: Normally begin with a lower-case letter, e.g. `x`, `tree`, `q1`, etc. "Anonymous" variables begin with the underscore symbol, and serve as place holders in data structures, e.g. `cons(h, _dontcare)`. The underscore symbol by itself is also an anonymous variable.

(v) *Structures*: In the current implementation of EqL, there are two built-in structured data objects: *trees* and *strings*. User-defined structures may be specified using the `constructor` declaration, explained later. We explain trees and strings below.

As in LISP, the built-in constructor `cons(x,y)` defines a binary tree. Examples:

```
cons(10, 20),
cons(10, cons(20,30)),
cons(cons(a,10), cons(B, 20))
```

In the third example, note that `a` is a variable, but `B` is a atom.

Because list-processing is a common application of functional and logic languages, EqL provides a special notation for *lists*. Similar to lists in LISP and

PROLOG, EqL lists are a special case of trees; they correspond to trees which "slope to the right" and end with the special symbol []. The following examples illustrate the connection between lists and trees:

List Notation	*Tree Notation*
[1,2,3]	cons(1,cons(2,cons(3,[])))
[[[1]]]	cons(cons(cons(1,[]),[]),[])
[]	[]

Similar to PROLOG, the notation

```
[ x | y ]
```

is used to stand for cons(x,y).

String constants are defined by a sequence of characters enclosed within a pair of double-quotes, e.g. "abc", "123", "longer string", etc. The empty string is "". Analogous to the above notation for lists, we use

```
[ x : y ]
```

to refer to the string obtained by prepending the one-character atom denoted by variable x in front of string denoted by y. For example, ['a' : "bc"] is the string "abc", and ['a' : ""] is the string "a".

We use the word *term* to refer to any data object of EqL that is built up from the above entities. We sometimes use the word *structured term* to refer to a term that has a constructor at the outermost level.

II.2. Rules

We illustrate program rules through examples. Below is a program to find the maximum depth of a binary tree of integers.

```
depth(x) => 0 where numberp(x) = true
depth([left | right]) => if dl > dr then dl+1 else dr+1
                           where
                                  dl = depth(left);
                                  dr = depth(right).
```

Program II.1: Maximum depth of a binary tree

The above program has two rules, which define the two cases for the depth function: the case for a leaf (an integer), and the case for a nonleaf (a tree built up from cons). In general, an EqL rule may take one of two forms:

1. *f(patterns)* => *expression.*
2. *f(patterns)* => *expression* where *equations.*

We explain each component of a rule below:

(i) *Patterns.* The word *pattern* is a synonym for *term*, which was defined in the previous section. Two or more patterns in a sequence are separated by commas, e.g.,

```
f([x1 | y1], [x2 :  y2], z) => ...
```

Zero-argument operations are permitted, and are defined by

```
f() => ...
```

(ii) *Expressions.* All expressions are evaluated in "applicative order," that is, leftmost-innermost expression first. The different kinds of primitives in the current implementation are listed below:

a. Terms: atoms, numbers, booleans, variables and *structures.*

b. Arithmetic: +, -, *, /, div, mod, and unary -. The operators + and - have lower precedence than * and /, which in turn have lower precedence than div and mod. Unary - has the highest precedence. The operators / and div both return the integer quotient. All binary operators are left associative. The function abs(x) returns the absolute value of x.

c. Relational: <, >, <=, and >=. The equality symbol = is reserved for defining equations. The function eq(x,y) maps identical atoms, numbers, and booleans to true; otherwise it returns false. The functions lessp(x,y), greaterp(x,y), lesseq(x,y), and greatereq(x,y) are identical to <, >, <=, and >= respectively.

d. Boolean: and, or and not. The operator not has precedence over and, which has precedence over `or`. The following domain predicates are provided in the language.

`numberp(x)`	: `true` if x is a number; `false` otherwise.
`boolean(x)`	: `true` if x is a boolean; `false` otherwise.
`atom(x)`	: `true` if x is an atom; `false` otherwise.
`var(x)`	: `true` if x is an unbound variable; `false` otherwise.
`listp(x)`	: `true` if x is a list; `false` otherwise.
`null(x)`	: `eq(x,[])`.

e. If then else:

`if` *boolean-expr* `then` *expr1* `else` *expr2*

returns *expr1* if *boolean-expr* reduces to `true`, and *expr2* if *boolean-expr* reduces to `false`. Both the then- and the else-part may have equations associated with them. Thus, for example,

(*expr1* `where` *equations*)

may be used in place of *expr1* above. We explain *equations* further below.

f. Application:

`f(`*arguments*`)`

where `f` is the name of some built-in or user-defined function, or a variable that is bound to such a function, and *arguments* is a sequence of zero or more expressions separated by commas.

g. Input:

`read(`*filespec*`)`

returns the next EqL data item in the file specified by *filespec*. The `read` operation is discussed in detail in Chapter IV.

h. Output:

`write(`*filespec*, $e_1, \ldots, e_n$`)`

evaluates e_1 through e_n and writes their values in the file specified by *filespec*. It returns the value of e_n, the last argument. The `write` operation is discussed in detail in Chapter IV.

(iii) *Equations.* An equation is of the form

$$expression_1 = expression_2.$$

However, for the purpose of explaining how equations are solved, we assume that an equation is of the form:

$$term = expression \qquad \text{or} \qquad expression = term.$$

Although permitted by our implementation, an equation $expression_1 = expression_2$ offers no extra power since it is equivalent to the pair of equations: `v`= $expression_1$; `v` = $expression_2$, where `v` is a distinct variable. Actually, it suffices to permit equations of the form `v` = *expression*, but permitting a *term* in place of variable `v` often leads to fewer intermediate variables in the source program, and hence clearer definitions.

Two or more equations in a sequence are separated by semi-colons. Because the interpreter is sequential, it solves equations in the sequence presented. There are two notable exceptions:

(1) An equation composed of terms, i.e., without any primitive or user-defined operations, would be solved prior to other types of equations. This re-ordering will not affect the correctness of the program's answer, and can help in avoiding unnecessary nontermination.

(2) An equation involving one of the primitive operators, such as `x = y + z` or `atom(x) = false`, would be deferred until sufficient information becomes available to solve the equation. This delaying is discussed in greater detail in section IV.2.

In the equation-solving rules described informally below, we refer to expressions being 'evaluated', but the reader should note that an expression e is evaluated by solving an equation $v = e$. We assume henceforth that an equation is of the form *term* = *expression*; the symmetric case is treated similarly. There are six cases for an equation, corresponding to the six forms of an *expression*:

1. $term_1 = term_2$. This is solved by syntactic unification. The equation is unsatisfiable if unification fails, and initiates backtracking.

2. *term* = $f(e_1, \ldots, e_n)$, where each e_i is an *expression*, and f is a user-defined operation, whose first rule is

   $f(t_1, \ldots, t_n)$ `=>` *expression* `where` *equations*.

   The equation reduces to the following sequence of equations, where each v_i is a distinct variable:

   $v_1 = e_1; \ldots ; v_n = e_n; \quad v_1 = t_1; \ldots; v_n = t_n;$ *equations*; *term* = *expression*.

   That is, the expressions $e_1 \ldots e_n$ are first evaluated and their results then unified with $t_1 \ldots t_n$ respectively; the *equations* in the body of f are then solved; and finally, the *expression* in the body of f is evaluated and unified with *term*. If at any stage unification fails, backtracking occurs to the dynamically most recent operation having an untried rule. Note that the evaluation order is essentially *call-by-value*. Also, the implementation ensures that each e_i is not re-evaluated when an alternate rule for f is attempted upon backtracking.

3. *term* = `if` p `then` e_1 `else` e_2. If expression p evaluates to a boolean, the equation *term* = e_1 or *term* = e_2 is solved. If p evaluates to an unbound variable, say x, two nondeterministic paths arise: (1) the equation *term* = e_1 is to be solved, where x ← `true`; and (2) the equation *term* = e_2 is to be solved, where x ← `false`. These two paths are tried out sequentially, with backtracking. Note that if e_1 or e_2 were accompanied by equations, these equations would be solved *before* solving *term* = e_1 or *term* = e_2.

4. *term* = *primitive*. Here *primitive* stands for an expression with an arithmetic, relational or boolean operator at the top-level. The arguments of the primitive operator are first evaluated. If they are fully instantiated, the operator is applied to the arguments to produce a result, which is then unified with *term*. Otherwise, the equation is *deferred*, or *delayed*, until all unbound variables are fully defined.

5. *term* = *read(filespec)*. The next EqL term is read from the file *filespec* and unified with the left hand side term. The equation is unsatisfiable if unification fails, and initiates backtracking.

6. *term* = *write(filespec*, $e_1, \ldots, e_n$*)*. This equation is equivalent to the following sequence of equations:

$$v_1 = e_1;\ \ldots\ ; v_n = e_n; \quad term = write(filespec, v_1, \ldots, v_n).$$

The value returned by *write* is that of v_n. This value is unified with *term*. The equation is unsatisfiable if unification fails, and initiates backtracking.

Finally, in any place where an equation might appear, EqL permits a membership assertion of the form:

$$term \in expression$$

The above membership is satisfiable if *term* is unifiable with some object in the set of objects that expression evaluates to. The remaining objects, if any, are immediately discarded from further consideration, analogous to the "cut" in Prolog. We explain $\in$ in greater detail in section IV.1.1.1.

Chapter III. The EqL Interpreter

In this chapter we describe the user interface of the interpreter, and the various capabilities provided, such as consulting files, tracing execution, etc. The interpreter is interactive: it reads the user queries, solves them, and prints results back.

III.1. Interacting with the Interpreter

The top-level query of an EqL program is either an expression or an equation or a set of equations, terminated by a period. A top-level expression e is actually treated internally as an equation, $\_ = e$, where $\_$ is the anonymous variable.

A typical session in EqL is a "conversation" between the user and the interpreter. The interpreter is first invoked by the command

```
% eql
```

where we assume `%` is the Unix command-level prompt. The interpreter would respond as follows:

```
EqL Version 1.0

eql>
```

For example, the response to a query

```
eql> 2+4.
```

would be

```
6

eql>
```

To exit EqL, type CTRL-d when the above prompt appears. The interpreter will respond with

```
[ EqL execution halted ]

%
```

III.2 Consulting Files

Often, a set of EqL rules are kept in a file, which can be read in by a consult operation (as in C-Prolog). For example, if depth is the name of a Unix file containing the two rules for the depth function, it can be read in as follows.

```
    eql> consult('depth').
```

EqL will respond with

```
    true
    eql>
```

Now, the depth function can be invoked on some input tree, e.g., [10 | [20 | [30 | 40]]], as follows:

```
    eql> depth([10 | [20 | [30 | 40]]]).
```

The interpreter would respond with

```
    3
    eql>
```

The consult operation can be used in any EqL rule, and will have the effect of reading in the rules contained in the file specified as the argument of consult. The consult returns true if the specified file is found, otherwise it returns false. Note that the rules from the specified file are read in only after the query evaluation is over and not at the time of the evaluation of the consult in the query. Note that consult augments the set of rules currently known to the interpreter; it does not replace any rule. Replacement of existing rules can be accomplished through the operation reconsult, described below.

Suppose that, after the file depth has been read in, the function depth is to be modified, say, to accept both atoms and numbers at the leaf of a tree. This could be achieved as follows: Use CTRL-z to suspend the interpreter; then edit the file depth so that the first rule reads as follows:

```
    depth(x) => 0 where numberp(x) or atom(x) = true.
```

Re-enter the interpreter using the Unix foreground command, and re-consult the changed file by typing:

```
eql> reconsult('depth').
```

The new rules in the file `depth` will replace the old rules, and the interpreter will respond with:

```
true

eql>
```

In general, rules read in by re-consulting a file would replace all existing rules that define the same operations as those defined in the rules read in.

The user can input rules directly, without suspending the interpreter, by executing:

```
eql> consult('tty').
```

After receiving the prompt,

```
true

|
```

the user can type in one or more rules. The end of the input is specified with a period on a new line. This mode of supplying rules is sometimes convenient when the rules are short.

The interpreter may also be initially invoked by specifying any number of input files in the command line, e.g.

```
% eql -f file1 file2 file3
```

A query may be included in a file by preceding it with a ?. Queries may be placed at the beginning of a file, between rules, or at the end of a file.

Before we proceed further, we note again that the top-level query can, in general, be an equation or a sequence of equations. For example, the following goal is equivalent to `depth([10 | [20 | [30 | 40]]])`:

```
eql> x = depth([10 | [20 | [30 | 40]]]).
```

The interpreter's response to this goal would be:

```
x = 3

eql>
```

Actually, the top-level query `depth([10 | [20 | [30 | 40]]])` would in fact be internally converted into an equation, `_ = depth([10 | [20 | [30 | 40]]])`.

If the EqL interpreter finds that it cannot solve the top-level query, it would respond with the message,

```
    no solution
```

This might happen, for example, when the top-level query is:

```
    eql> depth([]).
```

because `[]` is not an atom. To account for this case, an explicit rule,

```
    depth([]) => 0.
```

can be provided.

III.3. Obtaining Multiple Solutions

Consider the following definition for the familiar append function of LISP, for non-destructively concatenating two lists:

```
    append([], x) => x.
    append([h|t], y) => [h | append(t,y)].
```

Program III.1: List append

For example, the result of the query

```
    eql> append([1, 2], [3, 4]).
```

would be the list

```
    [1, 2, 3, 4].
```

It is just as easy to find out the lists x and y such that when appended together will yield the list `[1,2,3,4]`. This query can be expressed as follows:

```
    eql> append(x,y) = [1,2,3,4].
```

The interpreter will respond with:

```
    x = []
    y = [1, 2, 3, 4]
```

Upon typing a semi-colon at the end of the second line above, the interpreter will respond with the second solution:

```
        x = [1]
        y = [2, 3, 4]
```

Typing a carriage return instead of a semi-colon will cause the interpreter to discard remaining solutions and return with the

```
    eql>
```

prompt. All five solutions to the above query can be inspected by typing a semi-colon at the end of each preceding solution.

In general, when a semi-colon is typed and there are no further solutions, the interpreter will respond with the message:

```
    no solution

    eql>
```

The interpreter for EqL, like a PROLOG interpreter, explores alternative solutions to a query by depth-first search with backtracking.

III.4. Errors and Debugging

The EqL interpreter detects errors arising in different stages of the interpretation process. The parser detects all syntactic and lexical errors. A few runtime errors, such as divide by zero, undefined operation, etc. are also detected. Debugging facilities have been built into the interpreter to help detect other sources of programming error.

The EqL parser reports the line numbers on which errors occurred and the token on the line near which the error occurred. No further analysis of the error is made. Usual causes of errors are: unmatched parentheses and brackets; omitting important punctuation, such as commas and semicolons; and using a reserved word as a variable name. The reserved words are as follows:

`true`, `false`, `abs`, `div`, `mod`, `and`, `or`, `not`, `if`, `then`, `else`, `where`, `read`, `write`, `readb`, `writeb`, `line`, `list`, `char`, `int`, `consult`, `reconsult`, `trace`, `constructor`, cons, car, cdr, eq, atom, numberp, listp, var, boolean, greaterp, lessp, lesseq, greatereq, cputime, timer, save, null.

The keyword save, for saving the execution state, has not yet been implemented.

One of the most common sources of run-time error, during initial program development, is the invocation of an undefined operation, or an operation with incorrect number of arguments. The interpreter will fail when this happens, and backtrack as usual, but will print out a message indicating that it did not find the desired operation.

Errors in logic can often be detected using the trace feature. Two forms of tracing are available: non-selective tracing, and selective tracing. The former is specified by

```
    eql> trace.
```

which prints out a trace of every call of every function. To turn off the trace, the same command is issued. To selectively trace a function foo, type

```
    eql> trace('foo').
```

and to turn off the tracing of foo, repeat the above command.

Because the interpreter performs *last-equation optimization*, which is a generalization of tail-recursion optimization, exit information for an operation may not always be printed out. We illustrate the trace feature for the following program, which computes (naively) the reverse of a list:

```
    rev([])    => [].
    rev([h|t])=> app(rev(t), [h]).
```

Program III.2: Naive reverse

Suppose the top-level query were:

```
    eql> rev([1, 2, 3]).
```

The trace would be as follows:

```
trying rev at frame 0 with :    [1, 2, 3]
trying rev at frame 1 with :    [2, 3]
trying rev at frame 2 with :    [3]
trying rev at frame 3 with :     [ ]
Exiting frame 3
trying app at frame 2 with :     [ ]        [3]
Exiting frame 2
trying app at frame 1 with :     [3 ]       [2 ]
trying app at frame 1 with :     [ ]        [2 ]
Exiting frame 1
trying app at frame 0 with :     [3, 2]                [1]
trying app at frame 0 with :     [2]        [1]
trying app at frame 0 with :     [ ]        [1]

[3, 2, 1]
eql>
```

Notice that all invocations of the append operation (see program III.1) are performed on the same frame, and hence no exit information for these operations is printed. Also, because of *rule-indexing*, unproductive choice points are discarded based on the arguments to `rev` and `app` (rule-indexing is explained in section VI.3.2). A trace feature showing a fuller account of the execution has not been installed in this version of the interpreter. The above information, nevertheless, is of much use in identifying errors in the program.

The current version of the interpreter does not have a garbage collector. As a result if the interpreter runs out of space an error message is printed, stating that a stack overflow occurred, and the interpreter returns to the top level. In such a situation the user should reinvoke the interpreter with more space, using the command line options.

Errors in the interpreter would result in one of the following types of messages:

```
    ... : panic
    Bus error: ...
    Segmentation violation: ...
```

The latter message is also produced when attempting to unify two infinite objects or printing out an infinite object.

III.5. Miscellaneous features

Timing:

The current time, in microseconds, is obtained from the variable cputime. To time the execution of a goal, such as rev([1,2,3,4,5,6,7,8,9,10]), one may write the following query:

```
      eql> before = cputime;
           answer = rev([1,2,3,4,5,6,7,8,9,10]);
           time   = cputime - before.
```

The response, on a Sun-2, might be something like:

```
    before = 2516666
    answer = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
    time = 150000
```

indicating that the run-time for the rev(...) goal was approximately .15 Sun-2 seconds.

The automatic timing of goals can be requested by the command

```
    eql> timer.
```

which gives the cpu-time for all successive top-level queries, until turned off explicitly by the same command.

Interrupt:

The execution of any goal can be interrupted while in progress. The interpreter will trap the interrupt and respond as follows:

```
    What now?  (type h for help):
```

Upon typing h, the response would be:

```
    type a for abort
    type c for continue
    type t for trace
    type u for untrace
    type r for reset
    type e for erase constructors
    What now?  (type h for help):
```

Typing a causes the current query to be aborted; c continues execution; t starts tracing; u stop tracing; r causes all rules to be discarded; and e causes all constructor declarations to be wiped out.

Interpreter options

As mentioned in section III.2, the EqL interpreter may be invoked on any number of input files using the -f option. The other options allow the user to specify the sizes for various internal run-time data-structures except the *equation trail stack* which is implemented as a part of the *trail stack*. The general form of the Unix command line is

```
% eql [ -c control-stack-size ]
      [ -e equation-delay-stack-size ]
      [ -f file-names ]
      [ -h static-area-size ]
      [ -r read-stack-size ]
      [ -t trail-stack ]
      [ -v variable-stack-size ]
      [ -w write-stack-size ]
```

The *control-stack* holds the control information for each invocation of a user-defined operation; the space for variables is allocated separately in a *variable-stack*. Except for numbers and booleans, each entry in the variable-stack is a pointer. For structured objects, this pointer points to a *molecule-heap*. As in Prolog implementations, the *trail-stack* is used to record variables whose binding must be undone upon backtracking. The *read-stack* and *write-stack* are used to implement backtrackable read and write operations respectively. The *equation-delay-stack* is used to implement delayed execution of equations. The default allocations, measured in terms of stack entries, for each of the above data structures are: 75000 (-c), 10000 (-e), 75000 (-h), 10000 (-r), 75000 (-t), 150000 (-v), and 10000 (-w).

Chapter IV. Programming in EqL

We now illustrate through examples the various features of EqL: non-determinism, delayed evaluation of primitives, logical variables, and higher order operations.

IV.1. Nondeterminism

We already saw in the previous chapter that multiple solutions may exist for an equation. We illustrate nondeterminism in EqL further through two examples: the family database, and the N Queens problem.

IV.1.1. Family Database

Shown below are four operations: `f(x)`, which returns the father of some person x; `m(x)`, which returns the mother of x; `p(x)`, which returns the parent of x; and `gp(x)`, which returns the grand-parent of x.

```
f(Bob)  => Gary.
m(Bob)  => Mary.
f(Ann)  => Gary.
m(Ann)  => Mary.
f(Gary) => Joe.
m(Gary) => Jane.
f(Mary) => Steve.
m(Mary) => Sue.
p(x)    => f(x).
p(x)    => m(x).
gp(x)   => p(p(x)).
```

Program IV.1 : Family Relationships

The operation p(x) is nondeterministic because there are two rules for operation p. This reflects the fact that a person in this database has more than one parent—two to be precise. Because gp(x) is defined in terms of p(x), it is easy to see that gp(x) is also nondeterministic. The grand-parent of some person, say Bob, could be found by:

```
    eql> gp(Bob).
```

The first answer produced by the interpreter would be Joe, because p(Bob) would first return Gary, and p(Gary) would first return Joe. Upon requesting the next answer (by typing semi-colon), the interpreter would backtrack to the latest point where another choice is possible. Thus p(Gary) is recomputed, via m(x), to be Jane, which becomes the next answer to the top-level query. The other answers, Steve and Sue, are determined similarly.

The grand-children of some person, say Joe, could be found with query such as:

```
    eql> gp(x) = Joe.
```

IV.1.1.1. The use of <-

Suppose that we wanted to find the siblings of some person. Let us assume that two people are siblings if they have the same parents, and it suffices to check that they have one parent in common. We might initially try the following rule:

```
    sib(x) => y where p(x) = p(y); eq(x,y) = false.
```

With this definition of sibling, the response to a goal,

```
    eql> sib(Bob).
```

would first be Ann. Upon requesting another solution, we would once again receive Ann as the answer. The reason for this behavior is that the same answer, Ann, is discovered in two ways, first via the father of Bob, and again via the mother of Bob. We can avoid this behavior by selecting one parent of x, and then making sure that this parent is the same as that for y. In order to select an element from a set, the element-of construct, <-, can be used. The desired sibling definition is as follows:

```
        sib(x) => y where z <- p(x);
                          z = p(y); eq(x,y) = false.
```

Whenever the element-of construct is used to discard alternative solutions, the user should be aware that the "bi-directionality" of the operation using this construct could be affected. For example, a goal such as

```
eql> sib(x) = y.
```

will not enumerate all possible sibling pairs; rather, it will produce at most one answer, depending upon whether the first parent found in the database had two children or not. In our example, the response to the above query will be

```
x = Bob
y = Ann
```

Upon typing a semi-colon, the interpreter would respond

```
no solution
eql>
```

indicating thereby that there are no further solutions.

IV.1.2. N Queens Problem

We now present a more complex example of nondeterministic programming. The problem is to place N queens on an N by N chess board in such a way that no two queens are attacking one another. A simple approach to this problem is to place queens on successive columns so that each new queen placed is not attacked by any queen in the preceding columns. If a queen cannot be placed on a given column, we go back to the preceding column to see if the queen there has another "safe" position. If there are no safe positions remaining on that column, we back up to the preceding column, and so on. A solution is found if we can thus place queens on all N columns. We have exhausted all solutions if we attempt to go back from the first column to the 0-th column.

Below is an EqL program for specifying the desired search:

```
queens(n) => solve (1, [], n).
solve(col, safelist, n) => if eq(col,n+1) then safelist
                              else place ([col | row(n)], safelist, n).
place(q, safelist, n) => solve (col+1, [q|safelist], n)
                                   where q = [col|row];
```

```
                                          safe(safelist, q) = true.
safe([],q) => true.
safe([q1 | t], q) => safe(t,q) where threatened(q, q1) = false.
threatened([c1|r1], [c2|r2]) => eq(r1,r2) or eq(abs(r1-r2), abs(c1-c2)).
row(n) => n               where n>0 = true.
row(n) => row(n-1)        where n>0 = true.
```

Program IV.2 : N Queens Problem

The nondeterminism in the above program lies in the operation `row(n)`, which generates the sequence of integers n, n-1, ..., 1, one at a time. This provides a way for stepping through the rows in any particular column.

IV.2. Delayed Evaluation of Primitives

An interesting aspect of the execution of an EqL program is the way primitive operators are handled. Basically, when a primitive operation, such as `+`, `atom(x)`, `>`, etc., has all of its arguments defined, it is simplified to produce a result. However, when its arguments are not sufficiently defined when first encountered, it will be deferred until sufficient information is available to simplify it. This delayed evaluation has many uses, as we will illustrate in this and subsequent sections.

IV.2.1. Simulating Sets with Lists

The following rules define the familiar LISP operation `member`, which tests if an element x is a member of a list.

```
    member([],    x) => false.
    member([x|t],x) => true.
    member([y|t],x) => member(t,x) where eq(x,y) = false.
```

Program IV.3 : List Membership

It is easy to see that `member` will correctly check if an element, say 3, is a member of some list, say, `[1,2,3,4,5]`. Member could just as easily be used to enumerate the elements of a list, using a goal such as

```
    eql> member([1,2,3,4,5], z) = true.
```

which would return `1`, `2`, `3`, `4`, and 5 as the value for z, one at a time. What would happen if the goal

```
    eql> member([1,2,3,4,5], z) = false.
```

were presented to the interpreter? The first rule of member fails because `[]` does not match `[1,2,3,4,5]`. The second rule initially succeeds in unifying the goal arguments with its patterns, but its result, **`true`**, fails to match **`false`** in the top-level query. Thus the third rule of member is taken.

When the operation `eq(x,y)` is encountered, y is bound to 1 but x is unbound. The EqL interpreter therefore defers this equation (by moving it to a global stack of such deferred equations), and proceeds with the recursive call on **`member`**. Each succeeding call on member results in one new deferred equation, `eq(z,2) = false`, `eq(z, 3) = false`, etc. Finally, the first argument to **`member`** is `[]`, and rule 1 succeeds, and all equations are solved, except for the deferred equations. Because z is still unbound, the EqL interpreter responds as follows:

```
    Input equations not fully constrained

    z = _16

    eql>
```

The binding of z to a number preceded by the under-score symbol indicates that z is unbound. Although the above behavior might not seem useful at first, consider the following natural definition of set difference (in terms of **`member`**).

```
        diff(x,y) => d where member(y, d) = false;
                             member(x, d) = true.
```

Program IV.4 : Set Difference

The above rule states that the difference of two sets x and y (represented as lists) consists of elements d such that d is not a member of y and d is member of x. For example, the goal

```
    eql> diff([1,2,3,4,5], [1,3,5,6,7]).
```

will return 2 and 4 as answers, one at a time.

EqL is able to find these answers as follows. At the end of solving the first equation, `member(y, d) = false`, where `y = [1,3,5,6,7]`, there will be five deferred equations:

```
eq(d,1) = false; eq(d,3) = false; ...  ; eq(d,7) = false.
```

When attempting solve `member(x, d) = true`, with `x = [1,2,3,4,5]`, the binding of `d` to the values `1, 3, or 5` will cause one of the deferred equations to fail, and hence these are determined not to be solutions. However, the binding of `d` to the values 2 or 4 will cause all the deferred equations to succeed, and hence these are determined to be solutions.

IV.2.2. Arithmetic primitives

Deferring the evaluation of arithmetic primitives also has some interesting uses. Consider the conversion of centigrade to fahrenheit, expressed by the following rule:

```
f(c) => 32 + 9*c/5.
```

This rule can be used to convert centigrade to fahrenheit by a goal such as

```
eql> f(100).
```

It can also be used to find the centigrade for some particular fahrenheit, by a goal such as

```
eql> f(x) = 212.
```

To understand the process by which this equation is solved, it should noted that EqL converts the above rule into the following program:

```
f(c) => 32 + t2 where t1 = 9*c; t2 = t1/5.
```

The top-level goal, `f(x) = 212`, results the following sequence of equations to be solved:

```
t1 = 9*x; t2 = t1/5; 32 + t2 = 212.
```

EqL defers the first two equations, and solves the last equation to obtain the value of `t2`. With this information `t1` is determined from the second equation, and finally `x` from the first equation. Note that EqL will solve equations of the form $c = a \; op \; b$

where *op* is an arithmetic operator (+, -, *, /) and two of the three arguments (*a*, *b*, *c*) are known. With this capability, the reader may verify that the interpreter will be able to find the x such that

```
    eql> fact(x) = 24.
```

where `fact` is the familiar factorial function, defined as:

```
    fact(0) => 1.
    fact(n) => n*fact(n-1) where n>0 = true.
```

We also leave it to the reader to explain why, for example, `fact(x) = 23` will fail to terminate.

IV.3. Logical Variables

Much of the power in logic programming lies in its "logical variables." All variables in EqL are logical variables in that they derive their values not by direct binding but by the satisfaction of constraints. The following examples will serve as illustrations.

IV.3.1. Difference Lists

A classic example of logical variables is its use in defining "difference lists", which permit list concatenation in constant-time. The following is the EqL definition of difference-list concatenation.

```
    dconc([x|t], [t|y]) => [x|y].
```

Difference lists can be used to avoid the use of append in many places. Consider, for example, the following inefficient definition of quick-sort.

```
    qsort ([]) => [].
    qsort([p|l]) => append(qsort(a), [p|qsort(b)])
                        where [a|b] = part(l, p).
    part([], p) => [[] | []].
    part([h|t], p) => if p>h then [[h|a] | b] else [a | [h|b]]
                        where
                                    [a|b] = part(t, p).
```

Program IV.5 : Naive Quicksort

The above program is inefficient because it employs a linear-time append operation at each stage of the sorting process. This inefficiency can be avoided by concatenating the sorted lists in constant-time by representing them as difference lists, as follows.

```
sort(l)           => answer where [answer | []] = dsort(l).
dsort([])         => [x|x].
dsort([p | l])    => [sorta | tail] where
                                       [a | b] = part(l,p);
                                       [sorta | [p| sortb]] = dsort(a);
                                       [sortb | tail] = dsort(b).
```

Program IV.6 : Quicksort with difference lists

IV.4. Higher-order operations

One of the advantages of functional languages over logic language is their support of higher-order functions. A well-known example is LISP's `map` function, which "maps" a unary function to each element of a list in order to produce a new list with mapped elements. This function can be expressed in EqL as follows:

```
map([], f) => [].
map([h|t], f) => [f(h)   |  map(t,f)].
```

Program IV.7 : List Mapping

For example, the goal

```
eql> map([0,2,3,4], 'fact').
```

would return the list `[1,2,6,24]` as the answer.

(As an aside, the reader may note that, because of deferred evaluation of arithmetic operators, the goal

```
eql> map([a,b,c,d], 'fact') = [1,2,6,24]
```

would yield `a=0, b=2, c=3,` and `d=4`!)

Because function names are atoms, they can be held in data structures. The following example shows how to take advantage of this ability.

```
simplify([op, e1, e2]) => op(simplify(e1), simplify(e2)).
simplify(n) => n where numberp(n) = true.

add(x,y) => x+y.
times(x,y) => x*y.
sub(x,y) => x-y.
quo(x,y) => x div y.
```

Program IV.8 : Simplification

For example, the goal

```
eql> simplify(['times', ['add', 3, 4], 10]).
```

would yield 70.

Note that EqL does not permit nested definitions of rules or global variables. All functions exist at one level; functions do not have any lexically-scoped or dynamically-scoped environment from which they obtain information. Just as functions can take function names as arguments, they may also be returned as results.

IV.5 Strings and User-defined Structures

Consider the following definition for non-destructively concatenating two strings:

```
cat([], x) => x.
cat([h:t], y) => [h :  cat(t,y)].
```

Program IV.9 : String Concatenation

Using the above definition, we may determine the two portions, `front` and `back`, of some string `s`, such that they are separated by some specific word `w`, as follows:

```
split(w, s) => [front, back] where cat(front, cat(w, back)) = s.
```

User-defined constructors are declared before their use. They can be declared anywhere in a file between the rules, e.g.

```
constructor:  seq, fun.
```

where `seq` and `fun` are the names of the constructors.

IV.6. Input/Output

EqL provides two forms of I/O operations: non-backtrackable I/O, and backtrackable I/O. We first discuss non-backtrackable I/O.

Backtrackable I/O is done through the built in primitives **read** and **write**. The Unix file name which forms the source and the destination for read and write, respectively, is passed as an argument. The file name should be an atom. Full Unix path names of the files can be given provided they are enclosed within single quotes. The standard input and standard output are referred to by the file name TTY or 'tty'. The primitive

`read(`*filespec*`)`

returns the next EqL data item in the file specified by *filespec*. This data item could be an atom, number, boolean, or any structured term, including trees, lists and strings. The operation

`read(line,` *filespec*`)`

would read the EqL items (which may be numbers or atoms) on the current line, and return a list of these items. The operation

`read(list,` *filespec*`)`

would look for a list as the next item in the input. The operation

`read(char,` *filespec*`)`

would return the next character in the input, as an atom. The operations `read(int,` *filespec*`)` and `read(boolean,` *filespec*`)` allow reading an integer and boolean respectively.

All **read** operations would fail if an unexpected item is found in the input stream.

The **write** operation can take any number of arguments, which can be expressions. The expression

`write(`*filespec*, e_1, ..., e_n`)`

is treated as

`write(`t_1, ..., t_n`) where` $t_1 = e_1$; ...; $t_n = e_n$;

where each t_i is a distinct variable. The value returned by `write` is that of its last argument expression, namely, t_n. The non-graphic characters newline and tab are written following the C language convention, namely, `'\n'` (newline) and `'\t'` (tab).

EqL allows, as a special exception, a `write` to be used as an expression in any place where an equation might occur, e.g.,

```
write(TTY, 'Type a statement ending with period.', '\n')
```

This expression is converted internally into the equation:

```
_ = write(TTY, 'Type a statement ending with period.', '\n')
```

Because the current implementation is sequential, `read` and `write` operations have backtrackable variants, which, in this implementation, can occur only with respect to `'tty'`. To specify backtrackable read and write, use `readb` instead of `read`, and `writeb` instead of `write`, and omit the *filespec* in both cases. Backtracking past a `readb` would cause the data item read to be returned back to the input. Similarly, backtracking past a `writeb` would cause the output to be retracted. All output produced with a `writeb` will appear only when the top-level query has succeeded—the implementation maintains a buffer for all intermediate output; a similar buffer is maintained for backtrackable input.

The interpreter automatically opens a file when I/O is to be performed with it. At the end of each query, all files opened during the query are closed.

IV.7. Programming Hints

To conclude this chapter we offer a few suggestions to aid the construction of more efficient program. These are not meant to be rigid rules, but general guidelines.

- Try to distinguish the different rules defining an operation based on the first argument of each rule—this facilitates rule-indexing.
- Use pattern matching instead of if-then-else — this leads to clearer programs and faster execution.
- Ensure that operands of strict operators are fully instantiated when they are to be applied — equation delaying is an expensive operation.

Chapter V. Implementation of EqL

The implementation of EqL divides into two phases:

i. *The compilation phase*: the source code is read in and transformed into a binary-tree intermediate structure.

ii. *The interpretation phase*: the various execution stacks are set up, the query code evaluated and the results printed.

In the remainder of this chapter, we describe the compilation phase.

V.1. The Compilation Phase

Compilation involves lexical analysis, parsing and generation of intermediate structure. Lexical analysis and parsing are done using the Unix tools Lex and Yacc. Two key transformations are performed while generating the intermediate code: *flattening* and *equation reordering*. Below we describe the two transformations.

V.1.1. Flattening

To ensure left-most inner-most order of evaluation and to facilitate the temporary storage of subexpression values prior to expression evaluation, the source code is *flattened*. The *flattening* operation is simple: a subexpression `e`, which is not a term, is replaced by a compiler-generated variable `t`, and a new equation `t = e` is added to the existing set of equations. For example, consider the second definition of the inefficient version of `qsort` in section IV.3.1. After flattening it would become:

```
qsort([p|l]) => append(t1, [p|t2])
                    where [a|b] = part(l, p);
                          t1 = qsort(a);
                          t2 = qsort(b).
```

Here `t1` and `t2` are the compiler-generated temporary names. Note that the order of equations in flattened code reflects the desired evaluation order. As another example, the query

```
        f(x) + g(h(x)) = 23.
```

after flattening becomes

```
        t1 = f(x);
        x1 = h(x);
        t2 = g(x1);
        t1 + t2 = 23.
```

where `t1`, `t2` and `x1` are compiler-generated variable names.

The reader might note that the flattened EqL code resembles the source code for the comparable Prolog definition, in that the output variables of each Prolog predicate correspond to our compiler generated names. These output variables in Prolog pose an extra burden on the programmer, who has to manage them explicitly, and also lead to less clear definitions. EqL does not bar the programmer from using such output variables, but in most cases their use can be avoided.

V.1.2. Equation Re-ordering

Although EqL does not specify the order of solving equations, the interpreter we have implemented solves equations sequentially. There are, however, two exceptions to this general rule, referred to as *compile-time* and *run-time* equation re-ordering. We alluded to these re-orderings in section II.2. We discuss compile-time re-ordering in this section; run-time re-ordering is discussed in section VI.3.3.

In compile time re-ordering, an equation relating two *terms* is moved ahead of other kinds of equations. This re-ordering usually makes operation applications less non-deterministic, because unifying the two terms in such re-ordered equations usually results in variables being bound, and operation applications with more bound variables tend to be less non-deterministic.

Consider the query consisting of the equations

```
        f(x) = exp;
        x = [h|t].
```

where `f` is defined by n rules and `exp` is some arbitrary expression. Solving the second equation before the first binds `x` to `[h|t]`. With this binding for `x`, unifying f(x) against the various rules for f would very likely eliminate many of them without evaluating their bodies.

V.1.3. Compiling if-then-else

The condition part of an **if-then-else** may reduce either to a boolean or to an unbound variable (say x). In the former case, the then-part or the else-part is evaluated depending upon the value of the boolean. In the latter case, the interpreter has two non-deterministic choices: (i) bind x to **true** and return the value of the then-part as the first solution, and (ii) bind x to **false** and return the value of the else-part as the second solution.

The reader may note that implementing **if-then-else** using the rules shown below is not correct:

```
        if-then-else(true, T, E) => T.
        if-then-else(false,T, E) => E.
```

These rules are not a correct implementation because they cause the **if-then-else** to be strict in all arguments, instead of just the first argument.

In section VI.2.3 we describe the run-time support for executing an **if-then-else** expression. Below we illustrate the intermediate code generated for an **if-then-else** expression through an example. Consider the operation **solve** in the *N-queens* program.

```
solve(col, safelist, n) => if eq(col,n+1) then safelist
                           else place ([col | row(n)], safelist, n).
```

This is transformed into:

```
solve(col, safelist, n) => if bool then safelist
                            else (place ([col | t1], safelist, n).
                                     where t1 = row(n))
                               where t2 = n + 1;
                                     bool = eq(col, t2).
```

where bool is the compiler-generated name for the condition part of the if-then-else, and t1 and t2 are compiler-generated names for sub-expressions row(n) and n + 1 respectively. Note the placement of the flattened equations in the transformed code.

V.1.4. Intermediate Code Structure

The intermediate code is organized as a binary tree and is stored in the *static-area* (implemented as an array of *cells*). Each node of the intermediate code tree uses exactly one cell. A typical cell has the following information:

i. **A tag** which identifies the kind of node the cell stores, *e.g.* an atom, or a primitive operator, or a rule header, etc. The tag of a node determines the operation to be performed.

ii. **A left pointer** field which stores the pointer to the left subtree for an internal node of the tree, or the actual value of the constant for a leaf node.

iii. **A right pointer** field which stores the pointer to the right subtree for an internal node of the tree, or null for a leaf node.

iv. An **info** field which stores miscellaneous information, depending on the kind of node, e.g., the hash value of a rule name for a rule header cell.

As an illustration, the intermediate code for the body of the second rule of the append function (program III.1) is shown in figure V.1 (page 34).

The intermediate code of a rule is stored in the *rule store*, which is analogous to the *fact database* of Prolog. It is implemented as a hash table, where the hash value of a rule is the sum of the ASCII value of the characters in its name modulo 100. This value is computed and stored in the *info* field of the rule header cell at compile time. Rule definitions falling in the same bucket are stored as linear list, preserving the order they appear in the source program. Besides storing a pointer to the body of the rule, the following information is also stored:

i. rule name

ii. rule arity

iii. number of variables in the body of the rule

iv. list of formal arguments.

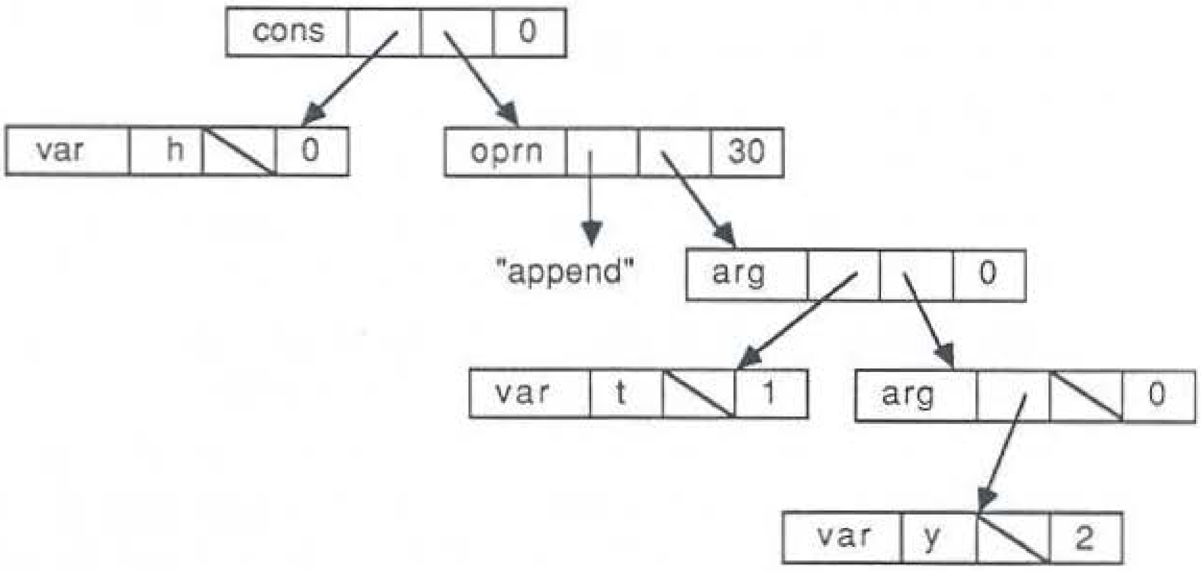


figure V.1.: Intermediate Code for append

Chapter VI. The Interpretation Phase

In this chapter we discuss the various operations involved in equation solution. We also discuss the optimizations incorporated in the interpreter for more efficient execution and quantify the improvements resulting from them.

VI.1. Overview

The top level of an EqL query has the following general form:

$e_{1l} = e_{1r};$

$\dots$

$e_{kl} = e_{kr}.$

where e_{il} and e_{ir} are expressions. The execution process consists of solving equations from the input query sequentially. The interpreter does case analysis to determine the appropriate action for solving an equation. In Chapter II we saw that equations are of 6 kinds:

1. $term_1 = term_2$
2. $term = f(e_1, \dots, e_n)$
3. $term =$ `if` p `then` e_1 `else` e_2
4. $term = primitive$
5. $term = read(filespec)$
6. $term = write(filespec, e_1, \dots, e_n)$

The EqL interpreter solves the equations using a seven-stack execution model. The seven stacks are:

1. *control stack*: The control stack records the *locus of control* during operation applications, and is similar to the activation record stack in conventional language implementations. Each operation application or `if-then-else` evaluation results in a *frame* being pushed on the control stack.

2. *variable stack*: Space for variables in a rule and the top level query is allocated on the variable stack.
3. *trail stack*: The trail stack is used for recording the addresses of the variables whose binding is to be undone upon backtracking, as in Prolog implementations [WPP77].
4. *equation-delay stack*: The equation-delay stack is needed to implement *equation delaying*. An equation is delayed (or its evaluation postponed) when the operands of the primitive operator in it are unbound. The delayed equation is solved later when operands get bound.
5. *equation-trail stack*: The equation-trail stack is used for recording those (delayed) equations which have been solved, but need to be re-solved on backtracking.
6. *read stack*: The read stack is needed for implementing backtrackable read, i.e. the operation `readb('tty')`. It stores EqL data objects read from standard input.
7. *write stack*: The write stack is needed for implementing backtrackable write, i.e. the operation `writeb('tty')`. It contains EqL data objects to be written out on the standard output upon completion of a query.

Apart from these seven run-time stacks the interpreter also accesses: (i) the *static area*, where the intermediate code is stored, (ii) the *I/O buffer* area, and (iii) the *molecule heap* where *molecules* are allocated space.

Note that control and data have been separated in the execution model by having separate stacks for them. This enables: (i) reclaiming the control stack frame upon completion of an operation application (*success exit*), and (ii) *last-equation optimization*, analogous to last-call optimization in Prolog implementations [H84].

VI.2. Equation Solution

We now describe how each kind of equation is solved.

VI.2.1. Solution of $term_1 = term_2$

This kind of equation is solved by unifying $term_1$ and $term_2$. If the unification succeeds, the equation is solved; otherwise backtracking is initiated. The variables in $term_1$ and $term_2$ are allocated space on the variable stack. The unification algorithm uses a small local stack to traverse structures. The algorithm does not do an *occurs check*, so a cyclic structure may be produced during runtime. It is the programmer's responsibility to make sure that programs do not generate cyclic structures during execution.

A term is either a constant, a variable, a structure, or a string. Representation of constants and variables is discussed later. A structure is either a primitive *cons* or a user defined constructor. In the next subsection we describe the representation of structures, using *structure sharing* [BM72].

VI.2.1.1. Structure Sharing : Constructors and Strings

As far as the implementation is concerned, there is only one built-in basic constructor – `cons`. All other constructors defined by the user are represented in terms of this basic constructor. For example, the term `tree(left, right)` would be represented as `cons('tree', cons(left, right))` where `'tree'` is an atom. Even the data type string is implemented in terms of `cons`.

A `cons`-object is allocated using a *molecule*. A molecule is a pair of pointers: one to the *skeleton* code (called the *code pointer*) and the other to the *environment* of the skeleton (called the *environment pointer*). The skeleton code consists of the intermediate code residing on the static area. Thus, if we were to solve the equation `x = cons(h, t)`, where x is unbound, the variable cell for x on the variable stack would be bound to a molecule as shown in figure VI.1. (page 38).

A string is implemented as list of characters, e.g., the string `"abc"` would be represented as `['a', 'b', 'c']`; the null string `" "` by the null list `[]`. A marker bit is set in each cell indicating whether it is a string cell or a list cell.

VI.2.2. Solution of $term =$ `f`$(e_1, \ldots, e_n)$

Solving this equation involves evaluating the operation application `f`$(e_1, \ldots, e_n)$ and then unifying the value returned with *term*. The first step in evaluation of `f`$(e_1, \ldots, e_n)$ is to obtain the definition of `f` from the *rule store*. In general, there might be more than one potential *candidate*, i.e. rules for `f` having the same arity.

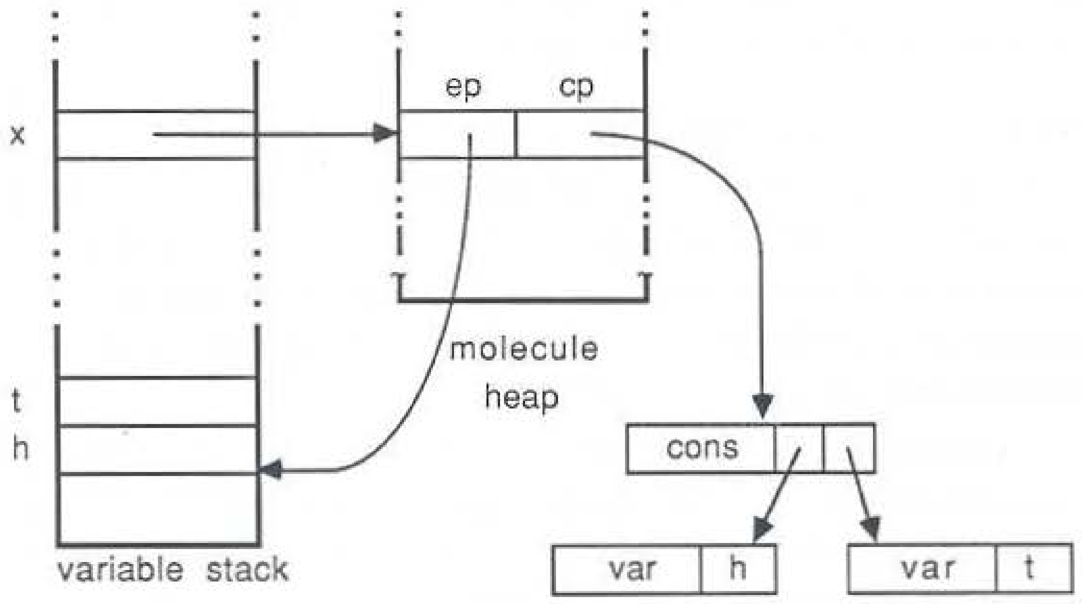


figure VI.1.: Binding a variable to a molecule

These candidates are tried one by one in *textual* order. Let us call these candidates C1, C2, ..., Cn, in order. The one to be tried first, C1, will have the general form:

C1: `f(`$s_1,\ldots,s_n$`) =>` exp_1 `where` $\mathcal{E}$

where $\mathcal{E}$ is a set of equations, and actual parameters of `f` unify with the terms $s_1,\ldots,s_n$.

The invocation of C1 by `f` leads to the following equations:

$\mathcal{E}$;

$term = exp_1$.

If the attempt to solve these equations succeeds, the original equation has been solved, and a *success exit* is said to have been taken from C1. If, however, the attempt to solve them fails, a *failure exit* is said to have been taken, and the original equation is retried with the next candidate C2 for `f`. Note that these equations might result in more operation applications. Of course, the attempt to solve them might also be non-terminating.

In the next two subsections, we first describe the two major data structures needed in supporting operation applications: the control stack and the variable stack. We then briefly describe the control algorithm.

VI.2.2.1. Control Stack

As the interpreter solves equations, it has to remember a variety of facts about the operations applications that have taken place so far. In other words, the interpreter should have some means of recording its locus of control. This control information is stored in a *frame* in the control stack. A typical frame contains fields which have the following information (name of the fields given in parentheses):

i. A pointer to the code to be executed by this frame (`cp`). This code is a list of equations.

ii. A pointer to the previous backtrack point (pb).

iii. A pointer to the trail stack (`tp`).

iv. Next candidate to be tried on failure (nc).

v. A pointer to the base of the area allocated to it on the variable stack, or the environment pointer(`ep`).

vi. A return point (rpt).

vii. A pointer to the parent frame, i.e. the frame which created this frame (pf).

ix. A pointer to the equation-delay stack(dsp).

x. A pointer to the read-stack (rp).

xi. A pointer to the write-stack (wp).

A frame corresponding to an operation with multiple definitions is called a *choice point*. The tp, nc and pb fields of a frame which is not a choice point are *nil*.

Figure VI.2 shows how the control stack would look when the frame for the candidate C1, from the example in the previous section, is pushed. In the figure the *current call* register (cc) points to the frame currently executing. The last three fields (dsp, rp and wp) have been omitted in the figure.

VI.2.2.2. The Variable Stack

The local variables of an operation application are allocated space on the variable stack. A pointer to the base of this area is stored in the ep field of the operation's frame. Each cell of the variable stack has two fields: one, to store the type of the datum, and the other, to store its actual value. For atoms, numbers and booleans the datum field contains the actual value of the constant. A variable-to-variable assignment, e.g., in the equation x = w, is recorded by storing a pointer to w's cell in x's cell. Constructors (including strings) are stored using the method of *structure sharing*, as discussed earlier.

VI.2.2.3. Control Algorithm: Overview

To conclude our description of an operation application, we briefly summarize the control algorithm. The algorithm has four phases:

i. *operation invocation*: The definition of the operation is retrieved from the rule store. The definition of the next candidate of that operation is also retrieved, if there is one. A frame is created for this definition, with all the required information put in the various fields, and pushed on the control stack. The current call register (cc) is updated to point to this new frame. Space is

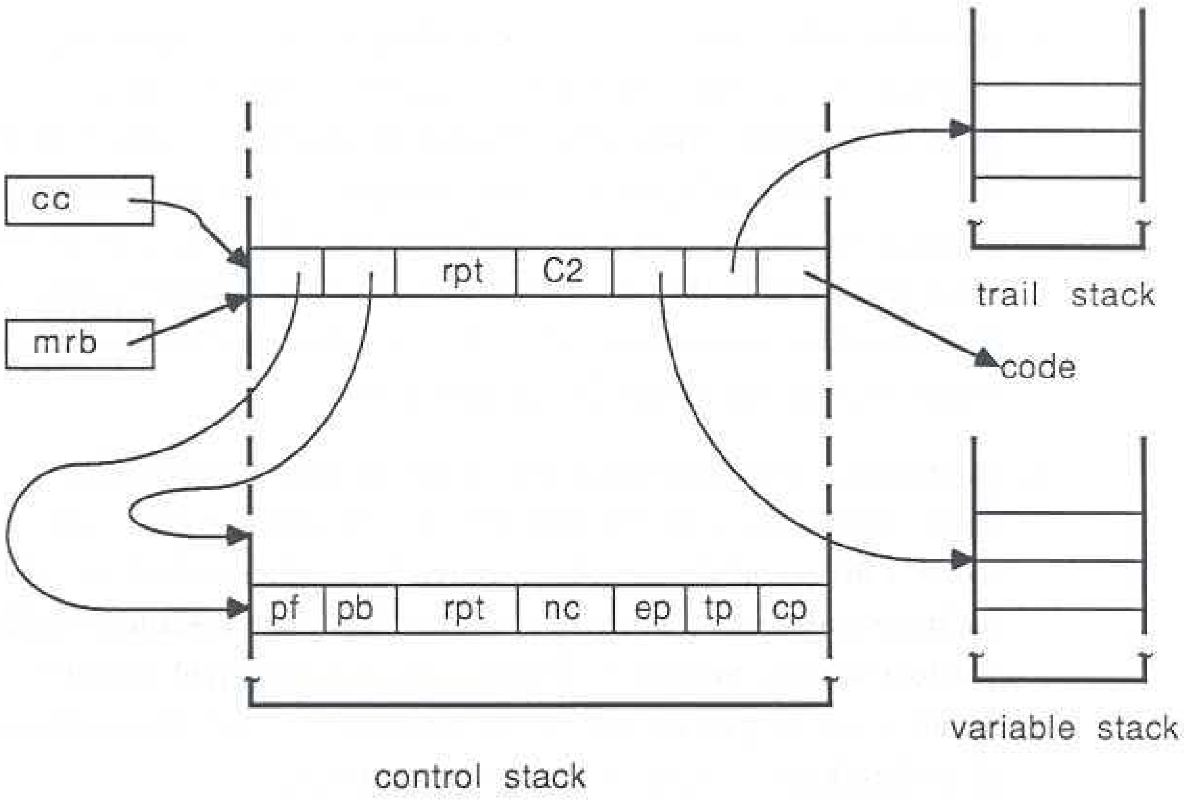


figure VI.2.: The Control Stack

allocated for the variables of this operation application on the variable stack, and execution is initiated.

ii. *operation execution*: The first step in the execution of the operation is the unification of its actual parameters with its formals. If the unification fails, backtracking takes place. If it succeeds, the equations in the body of the operation are solved sequentially. If any of these equations turn out to be unsatisfiable, backtracking takes place. These equations might lead to more operation applications.

iii. *operation exit*: Once all equations in the body of the operation have been solved, a *success exit* is taken from its frame. This is done by making the `cc` point to the *parent* of the current frame and restarting the execution from the equation pointed to by the return point (`rpt` field) of the current frame. If there are no choice points above the frame pointed to by `cc`, all frames above it are popped, since they are no longer needed. The variable bindings made by the deleted frames are preserved on the variable stack. This is one reason for separating data and control in the interpreter.

iv. *backtracking*: When an equation turns out to be unsatisfiable, the interpreter has to backtrack and retry the most recent call with an untried rule. A pointer to the most recent choice point is stored in a register called the *most recent backtrack point register* or the `mrb` register. When `mrb` needs to be updated, its previous value is stored in the *previous backtrack point* (`pb`) field of the current frame. Thus the `pb` field in the different choice points form an ordered chain of backtrack points with the first one being in `mrb`.

When failure occurs, control is transferred to the frame, say *f*, pointed to by the `mrb`. All the frames above *f* are discarded. The `mrb` is updated with the value of the *previous backtrack point* field of the frame *f*. The bindings of the variable recorded on the trail stack between the trail-stack top and the `tp` field of *f* are undone, and the trail-stack top is set to value of the `tp` field of *f*. The next candidate to be tried is retrieved using the nc field of *f*. Frame *f* is modified to store the information for the new definition and the computation is restarted.

VI.2.3. Solution of *term* = `if-then-else`

The interpreter executes the `if-then-else` like a user defined operation application, except that only the first argument, corresponding to the predicate condition, is evaluated in a call-by-value fashion. As we saw earlier, if the predicate condition is an expression it gets flattened out and replaced by a variable. We will refer to this variable introduced as the *predicate variable.* At the time of evaluation of the `if-then-else`, the predicate variable can be in one of the three valid states: unbound, bound to the constant *true*, or bound to the constant *false*. If the predicate variable is bound to something else, the `if-then-else` fails.

The cases when the predicate variable is bound to *true or false* are similar. We therefore explain just the *true* case: When the predicate variable is bound to *true*, the value of the then-part is returned. The interpreter treats the then-part as an anonymous operation application. Both the then-part and the else-part have the general form:

exp `where` *equations*

resembling the body of a operation. Thus, a frame gets pushed on the control stack after the various fields have been set up. This frame is not a choice point, and hence no backtrack information is required to be stored in it. It also does not get allocated any space on the variable stack, because the `if-then-else` shares all its variables with the operation it appears in. Thus the environment pointer of the frame for the then-part is the same as the environment pointer of its parent. Once the frame has been set up and pushed, execution continues normally.

When the predicate variable is unbound the `if-then-else` evaluates to more than one values. Thus, an equation:

```
z = if x then 10 else if y then 20 else 30.
```

would yield the following 3 solutions.

```
x = true
y = _
z = 10 ;

x = false
y = true
```

```
z = 20 ;

x = false
y = false
z = 30 ;

no solution
```

Implementing the `if-then-else` with unbound predicate variable is more involved. Although the frame for then-part shares the variable space with its parent frame, it has a separate pointer pointing into the trail-stack because only those variable which got bound as a result of the execution of the then-part need be unbound upon failure. Note that, in an `if-then-else` with an unbound predicate variable, the frame for the then-part can *always* be overwritten with that of the else-part. Thus, the execution of an `if-then-else` never requires more than a single frame. The advantages of treating the `if-then-else` evaluation as a pseudo operation application are:

i. the control mechanism remains simple and uniform;

ii. it is possible to provide full backtracking on the unbound predicate variable with no extra cost, and

iii. because the `if-then-else` is often deterministic the corresponding frame is not a choice-point. This aids last equation optimization (LEO), discussed later.

The mechanism described above can handle arbitrarily nested `if-then-else`.

VI.2.4. Solution of *term* = *primitive*

To solve an equation with a primitive strict operator, the arguments of the operator are first evaluated, the operation is then applied to arguments, and the result is then unified with *term*. If unification fails backtracking takes place. If one of the operands is not bound, the equation is *delayed*. If *term* is a number, or a variable bound to a number, and the operator is arithmetic (`+`, `-`, `*`, `/`), with only one operand unbound, then the interpreter computes the value of its other operand and binds it. Equation delaying and solution of arithmetic equations is discussed below.

VI.2.4.1. Equation Delaying

To implement equation delaying the interpreter maintains a stack called the *equation delay stack*. When an equation is found to have a primitive operator with unbound operands, a pointer to its code is stored in the equation delay stack. The `dsp` field in the control frame stores the top of the delay stack whenever a choice point is pushed. On backtracking to this choice point, the top of the delay stack is restored to the value in this field.

Every time a success exit is taken, the interpreter tries to solve all equations on the delay stack. If a delayed equation is solved and belongs to a frame lying below the `mrb`, a pointer to this equation is also recorded on the *equation trail stack*. This information is recorded so that upon backtracking this equation can be marked as still unsolved, and re-solved later with correct bindings.

VI.2.4.2. Solving Arithmetic Equations

Because of flattening, all arithmetic operations occur in equations of the form:

`a = b` *op* `c`, or

`b` *op* `c = a`

where *op* is `+`, `-`, `*`, / etc.

When an arithmetic expression with an unbound operand is found, the term on the other side of that equation is examined. If it is a number, or a variable bound to a number, then the value of the unbound operand is computed and stored in the variable cell of that operand on the variable stack.

Note that equations which have only one occurence of an unbound variable can be solved immediately. Equations of the kind `x + y = 20`, where `x` and `y` are unbound, would be delayed using the mechanism described in the previous section. It can be solved as soon as one of the variable gets bound. Solving such equations, rather than delaying them, makes the interpreter more efficient.

VI.2.5. Solution of *term* $\in$ *expression*

Implementing the $\in$ assertion requires actions to be taken at compile time as well as runtime. The intermediate code generated for the $\in$ assertion is essentially identical to that of an ordinary equation. Note that equations generated from flattening of the $\in$ assertion are treated as ordinary equations and not as $\in$ assertions. Thus the equation

```
v ∈ f(g(x)).
```

becomes

```
t = g(x);
v ∈ f(t).
```

The *term* $\in$ *expression* assertion behaves like an ordinary equation if *expression* yields a single value. If *expression* yields more than one value, then at runtime a bit is set in the frame corresponding to this expression, indicating that the rest of the solutions arising out of this frame are to be discarded upon success exit. Pruning the remaining solutions amounts to deleting all backtrack points above this frame at the time of success exit. This is done by making the `mrb` point to the most recent choice point below the expression's frame.

VI.2.6. Solution of *term = read(filespec)*

The non-backtrackable read is simple to implement. The interpreter keeps a list of files which have been opened as a result of the top level query. These files are closed once the query evaluation is over. If the file *filespec* is in the list of open files, the next EqL data object is read from that file and unified with *term*. If it is not, it is opened and added to the list of open files, and then the read is performed.

Implementing the backtrackable read (`readb`) requires a *read-stack* to be maintained. The `rp` field, in the control stack, stores the top of the read-stack when a choice point is pushed. A *read-pointer* records the position in the read-stack from where the next call to `readb` would get the data-object. During backtracking the read-pointer is updated with the value in the `rp` field. Actual I/O with the standard input occurs only when the read-pointer coincides with the top of the read-stack – in which case the data object read is also stored in the read-stack, in addition to being returned as the value of the `readb` expression.

Implementing `write` and `writeb` is very similar to `read` and `readb` hence we omit its description in this thesis.

VI.3. Optimizations

So far we have described the core of our interpreter. Now we will discuss some optimizations incorporated in our interpreter. These optimizations are rule-indexing, last-equation optimization, and run-time equation re-ordering. These optimizations lead to considerable efficiency in execution space and time as demonstrated by our experiments.

VI.3.1. Last Equation Optimization

It is not necessary to delay the reclamation of a frame on the control stack until a *success exit* is taken. It is possible to reclaim a frame, subject to certain conditions, at the time its last equation is being solved. The last equation optimization (LEO) is similar to the last call optimization (LCO) in Prolog implementations [H84, W83] and the tail recursive optimization (TRO) in Lisp implementations.

Functional language implementation have fewer opportunities to do TRO due to their expression nesting. Consider the second rule in the functional definition of append:

```
append([], y) => y.
append([h|t], y) => [h| append(t, y)].
```

Since cons is at the outermost level, it is not possible to apply TRO to the recursive call on append. On the other hand LCO can be applied to the recursive call to append in the Prolog definition for append given below:

```
append([], Y, Y).
append([H|T], Y, [H|Z) :- append(T, Y, Z).
```

This is an advantage of the relational style of programming over the functional style — more opportunities for LCO exist.

LEO in EqL is as powerful as LCO in Prolog. That is, LEO would be performed during the execution of EqL code where ever LCO would be performed in the execution of the equivalent Prolog code. This is achieved through compile-time as well as run-time equation reordering. We illustrate this point by considering the above rules for append, and the goal

```
append([1, 2], [3, 4]) = ans.
```

During execution, after the second rule has been matched, this goal would be transformed into:

```
append(t1, y1) = z1;
cons(h1, z1) = ans.
```

Note that **z1** appears due to compile time flattening of **append**. Due to equation re-ordering, the second equation would be moved before the first which then triggers a LEO.

The conditions for an equation to qualify for LEO are:

i. the equation should be the last to be solved in that operation code, and

ii. the called operation should not be a choice point.

Both these conditions are fairly easy to determine at runtime. In principle, the condition above can be further relaxed, and it is possible to do a LEO when the caller is not a choice point and the callee is. However, this requires extra information to be maintained on the control stack and hence we did not incorporate it in our interpreter. Note that it is the separation of control and data which enables the callee frame (which is not a choice point) to overwrite the calling frame even when the former is a backtrack point.

LEO not only saves space but also time. Space is reduced because frames get deleted from the control stack as soon as they are no longer needed. Programs run faster because when the called frame overwrites the calling frame only 50% of the fields need updating; others (such as the parent frame pointer, return pointer, etc.) remain the same and do not require modification. Thus, overwriting a frame is a much cheaper operation than pushing a new one. Later, we give some statistics showing the improvement in speed due to LEO.

VI.3.2. Rule-indexing

In order to preserve variable bindings during an operation application, a frame needs to be set up on the control stack before unifying the actuals and the formals. Very frequently, this unification fails and the time spent in setting up this frame goes wasted.

Rule-indexing avoids this wasted effort. The first actual argument of the operation application is unified with the first formal argument of the operation definition before the execution stacks are set up. The unification algorithm used here is only an approximation to the actual unification algorithm. If rule-indexing succeeds, the execution stacks are set up and true unification is performed. If rule-indexing fails, the next definition is considered in a similar fashion. Even though rule-indexing performs only an approximate unification, our experiments show that it cuts down considerably on the number of potential candidates for an operation application. Since it is performed before the frame for the operation is pushed, it leads to LEO being applied more often, thus multiplying the savings in time and space.

The *approximate* unification algorithm used for rule-indexing is as follows:

i. A constant term unifies with a constant term if they are identical.

ii. A formal argument which is a variable unifies with anything.

iii. Two structures unify only if their first components unify (in the *approximate* sense).

iv. An empty list does not unify with a `cons` structure.

v. An unbound variable in the actual argument unifies with anything.

For example, `[x, y]`, if present as the first formal argument would unify with `[1, 2, 3]` in the *approximate* sense, though it wouldn't in the *true* sense.

VI.3.3. Runtime Equation Re-ordering

Consider a typical operation

```
        f(...) => exp where S
```

where *S* is a set of equations, and an equation:

```
        f(...) = rhs.
```

At runtime this equation reduces to:

```
        S:
        exp = rhs.
```

Since the equation `exp = rhs` is generated at run-time, re-ordering is necessary if both `exp` and `rhs` are terms. The effect of this runtime equation reordering is twofold: (i) it leads to a greater number of LEOs, and (ii) it helps avoid non-termination in some cases. To illustrate this point, consider the definition of append, in the section VI.3.1., and the query

```
append(x, y) = [1, 2].
```

This goal would yield three solutions. Without run-time equation re-ordering, however, non-termination occurs after the last solution is reported. This non-terminating computation is caused due to the generation of the equations:

```
append(_, _) = _.
[] = cons(..., ...)
```

where _ denotes an unbound variable.

As a result of runtime re-ordering, the equation `[] = cons(..., ...)` is solved first, thus causing termination with failure. The interpreter then reports to the user that no more solutions exist.

The reordering also forces the recursive call to append in the second rule to be in the last equation, enabling LEO. As a result of LEO, the entire query gets solved in a *single* frame.

VI.3.4. Performance analysis

The two optimizations we consider for demonstrating the performance improvement are LEO and rule-indexing. The table below shows the runtimes for some standard programs – append (program III.1), naive reverse (program III.2) and, searching 30 elements in an association list. These programs were run on a Sun-3 work station. The last two digits after the name, in the first column of the table, tell the length of the list on which that program was run. The column labeled BASIC lists the timings for the version of the interpreter with no optimizations, the one labled RI for the version with rule indexing, and the last for the final version with rule-indexing and LEO. All the timings are in SUN-3 cpu seconds.

Program	BASIC	RI	RI+LEO
app30	0.045	0.030	0.010
app60	0.100	0.070	0.030
rev30	0.700	0.550	0.410
rev60	2.840	2.300	1.610
assoc30	0.070	0.065	0.060

The improvements from the optimizations should are obvious from the table.

Note that the time taken to traverse a 30 element association list is considerably more than the time taken to append a 30 element list to another list. This corroborates the observation that structure sharing favors data construction [M80].

Chapter VII. Conclusion

Although EqL supports functional programming more directly than logic programming (because of its functional syntax), we hope it is clear from the examples that many logic programming paradigms are also readily supported. In fact any pure Prolog program can be directly converted into an EqL program [JSG86]. EqL programs are often clearer than equivalent Prolog programs because function composition supplants the need for "output variables" in Prolog programs (see definition of *naive* reverse, for example). EqL's if-then-else and $\in$ correspond to well-structured uses of Prolog's cut, and their use also leads to simpler formulations in many cases.

EqL also supports arithmetic, relational and boolean primitives, and delays any equation with these primitives until sufficient information becomes available. Other useful features in the language are strings, user-defined constructors, and higher-order operations.

The EqL interpreter divides into two phases: compilation and interpretation. Compilation involves generation of binary-tree intermediate code, accompanied by flattening and equation re-ordering transformations. The intermediate code is stored in the *static area*. The interpretation phase employs seven stacks, each with fixed size entries. The novel aspects of this execution model are: (i) the separation of the conventional recursion stack into a *control stack* and *variable stack* (this separation is crucial for supporting LEO), (ii) the use of *equation delay* and *equation trail* stacks for delayed evaluation of primitives, and (iii) the use of *read* and *write* stacks for backtrackable read and write. The other stack is the *trail* stack, used for recording the addresses of the variables to be unbound on backtracking. The interpreter also performs *rule-indexing*, an optimization that reduces execution time by eliminating unnecessary choice points.

From our experiments, we found that last equation optimization and rule indexing improve runtime performance considerably. LEO is actually more powerful than tail recursive optimization (TRO) in functional language implementations, because it (LEO) is applicable to certain non-tail-recursive definitions such as LISP's append, which have a recursive call embedded inside a constructor at the outermost level. Note that because of the complete separation of the control and variable stacks, the variable stack shrinks only during backtracking. Garbage collection therefore is needed when no more space is left on it. Our current implementation does not support garbage collection yet.

The EqL interpreter runs at about half the speed of the C-Prolog interpreter for standard benchmark programs such as naive reverse. By compiling EqL programs to WAM-like code [W83], comparable performance with Prolog can be expected. We nevertheless hope we have demonstrated that the elegance of functional notation and the expressiveness of logic may be combined (via equation solving) to yield a practical declarative language.

References

[BM72] R. S. Boyer and J. S. Moore, "The sharing of structure in theorem proving programs," In *Machine Intelligence* **7**, B. Meltzer and D. Michie (eds.), 1972, pp. 101-116.

[CM81] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.

[DP85] N. Dershowitz and D. A. Plaisted, "Applicative Programming *cum* Logic Programming," In *1985 Symp. on Logic Programming*, Boston, pp. 54–66.

[F84] L. Fribourg, "Oriented Equational Clauses as a Programming Language." *J. Logic Prog.* **2** (1984) pp. 165-177.

[GM84] J. A. Goguen and J. Meseguer, "Equality, Types, Modules, and (Why Not?) Generics for Logic Programming," *J. Logic Prog.* **2** (1984) pp. 179–210.

[H84] C. J. Hogger, *Introduction to Logic Programming*, Academic Press, 1984.

[J88] B. Jayaraman, "Semantics of EqL," To appear in *IEEE Trans. on Software Engg.*, March 1988.

[JG87] B. Jayaraman and G. Gupta, "EqL User's Guide," TR 87-010, Dept. of Comp. Science, UNC - Chapel Hill, May 1987, 30 pages.

[JG86] B. Jayaraman and G. Gupta, "Parallel Execution of an Equational Language" *In Proceedings of the Workshop on Graph Reduction*, Santa Fe, 1986, Lecture Notes in Computer Science, Vol 279, pp 370-381.

[JS86] B. Jayaraman and F.S.K. Silbermann, "Equations, Sets, and Reduction Semantics for Functional and Logic Programming," *In 1986 ACM*

Symposium on LISP and Functional Programming, pp. 320-331, Boston, 1986.

[JSG86] B. Jayaraman, F.S.K. Silbermann, and G. Gupta "Equational Programming : A unifying approach to functional and logic programming", *In Proceedings of the International Conference on Computer Languages*, Miami Beach, 1986.

[M80] C. S. Mellish, "An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter", *In Proceedings of the Logic Programming Workshop*, Hungary, 1980.

[RS82] J. A. Robinson and E. E. Sibert, "LOGLISP: Motivation, Design, and Implementation," In *Logic Programming*, Ed. K. L. Clark and S.-A. Tãrnlund, Academic Press, 1982, pp. 299–313.

[W83] D. H. D. Warren, "Optimizing Tail Recursion in Prolog", *In Logic Programming and its Applications*, ed. Michel Van Caneghem and D. H. D. Warren, 1983.

[WPP77] D. H. D. Warren, F. Pereira, and L. M. Pereira, "Prolog: the Language and Its Implementation Compared with LISP," *SIGPLAN Notices* **12**, No. 8 (1977) pp. 109–115.

[YS86] P. A. Subrahmanyam and J-H. You, "Equational Logic Programming: an Extension to Equational Programming," In *13th ACM POPL*, St. Petersburg, 1986, pp. 209-218.