MCC Technical Report Number AI-194-86

Rewrite Rule Compilation

Robert S. Boyer

MCC Nonconfidential

June 1986
Reclassified from ACA Confidential and Proprietary to
MCC Nonconfidential - June 1988

Abstract

We describe a method for compiling rewrite rules which produces a substantial speedup over simpler rewriting strategies. Common Lisp code to perform the compilation is presented, and a timing camparison with one of the Gabriel benchmarks is provided. A speed-up factor of about 350 is achieved.

Microelectronics and Computer Technology Corporation
Advanced Computer Architecture Program
Artificial Intelligence Laboratory
3500 West Balcones Center Drive
Austin, TX 78759
(512) 343-0978

Contents

1 Introduction

The development of fast algorithms for the execution of logic programs suggests that other kinds of theorem-proving algorithms can be substantially speeded up. In this report we examine one of the simplest theorem-proving operations, called *rewriting* or *demodulation*. We present a simple algorithm for one strategy of rewriting, an algorithm that forms the basis of one of the Gabriel Common Lisp benchmarks [2]; we then present a compilation-based means of obtaining the same ends. The compilation-based technique executes several hundred times faster than the simpler strategy.

This paper may be regarded as an informal introduction to the topic of compilation-based techniques for automatic theorem-proving. However, because we are concerned in this report only with rewriting, it should be noted that important topics in compilation-based techniques (e.g., backtracking) are not discussed here. In some sense, the ideas presented in this report are "obvious" to the half-dozen experts in the field and perhaps are already part of the "folklore" on the subject. But the actual speed-up of about 350 obtained by applying these techniques to this particularly well-studied benchmark was quite a surprise to the author.

2 What is Rewriting?

Among the oldest algebraic operations are *instantiation* and the *substitution of equals for equals*. Although these operations are simple, they are also powerful. For example, these two operations alone form the basis for the Herbrand-Gödel system of general recursive functions. For example, if we know that

```
fact(0) = 1
```

and

```
fact(s(x)) = s(x) * fact(x)
```

we can, by only instantiation and substitution of equals for equals, prove that:

```
fact(s(s(x))) = s(s(x)) * s(x) * fact(x).
```

and that

```
fact(s(s(0))) = s(s(0)) * s(0) * 1
```

The question this report concerns is "How fast can we performs these operations of instantiation and substitution of equals for equals?" No theoretical answers are given, but we do present an example of a method that has been in use in automatic theorem-proving for years and another "compiler-based" method that is several hundred times faster.

2.1 The Utility of Rewriting

In the language of automatic theorem-proving the use of equations to *simplify* or *reduce* an expression is a very common operation. It seems to be an excellent heuristic to keep expressions in simplified form, and therefore after every significant theorem-proving operation, it is common to reduce the resulting expressions by rewriting. For example, in the Argonne ITP-LMA system, demodulation with all "demodulators" may be performed whenever a clause is generated, which is to say, frequently.

2.2 An Informal Algorithm for Rewriting

Given a collection of equations S and a formula F, we seek to obtain some derived formula G which is obtained by using the equations in S, from left to right, until no rule in S is applicable. A nondeterministic algorithm for obtaining such a maximally reduced G is:

1. START. Let H be the starting formula F.
2. LOOP. If possible, find a subterm T of H and an equation lhs = rhs of S for which there exists a (most general) substitution σ such that lhs_σ is equal to T. If no such term, equation, and substitution can be found, then set G to H and terminate. Otherwise, go to 3.
3. REPLACE. Let H be the result of replacing the occurrence of T with rhs_σ in H. Go to 2.

There are many choices in the strategy for selecting the subterm to work on. For the purposes of this report, we focus on the *leftmost-innermost* strategy.

3 A Simple Algorithm for Rewriting

A simple algorithm for doing leftmost-innermost rewriting is presented in Appendix A. We discuss the algorithm informally in this section. In the next section (4), we will discuss a much faster algorithm, but understanding the slower algorithm is necessary to understanding section 4.

The slower algorithm was developed by the author and J Moore as a simple program that could be quickly transported to different Lisp systems in order to benchmark them with an eye towards determining how the very much larger Boyer-Moore theorem-prover [1] would perform on those systems. This algorithm was tried out on enough systems that Richard Gabriel decided to include it among the benchmark algorithms in his book [2]; times for this "Boyer" benchmark are now among those commonly cited in the advertising for new Lisp systems. This algorithm is certainly far from being the most naive method for rewriting. The fact that the compilation based algorithm to be presented

in the next section is 350 times faster than the simple algorithm was a great surprise to the author!

3.1 The Representation of Terms

In order to describe the algorithm, it is best to start with a description of the representation of terms. We use the simplest kind of Lisp technique to represent terms. The Lisp expression:

```
(EQUAL (QUOTIENT (PLUS X (PLUS X Y))
                 (TWO))
       (PLUS X (QUOTIENT Y (TWO))))
```

is how we represent the equation

```
(x + (x + y)) / 2 = x + (y / 2).
```

A term is either a variable, which is represented by a Common Lisp `SYMBOLP`, or it is the application of a function symbol (also represented by a Common Lisp `SYMBOLP`) to some arguments, and is represented by `CONS`ing the function symbol on to a list of the arguments.

3.2 The Storage of Rules

The algorithm is initialized with a set of rewrite rules (in the global variable `*RULES*`), by invoking `(SETUP)`. `SETUP` places on the property lists of function symbols the rules that are applicable to terms beginning with that function symbol. This segregation of rules is a minor efficiency; it permits us to avoid considering rules about `REMAINDER` when we are trying to rewrite terms that begin with `QUOTIENT`. After `SETUP` has been invoked the property `LEMMAS` of `QUOTIENT` consists of the following list of length two:

```
((EQUAL (QUOTIENT (TIMES Y X) Y)
        (IF (ZEROP Y) (ZERO) (FIX X)))
 (EQUAL (QUOTIENT (PLUS X (PLUS X Y)) (TWO))
        (PLUS X (QUOTIENT Y (TWO)))))).
```

That is, there are just two equations in `*RULES*` with a lefthand side whose outermost function symbol is `QUOTIENT`.

3.3 REWRITE

In the Gabriel Benchmark, the computation that is timed is (TAUTOLOGYP (REWRITE *TERM*) NIL NIL). The overwhelming percentage of the computation time is spent executing the REWRITE call, and we will ignore TAUTOLOGYP for the remainder of this report.

The function REWRITE performs leftmost-innermost simplification of its parameter. If the parameter TERM is a variable, it is simply returned. Otherwise, each of the arguments of TERM is rewritten. Then REWRITE-WITH-LEMMAS is applied to the result of consing the function symbol, say *fn*, of TERM on to the list of the rewritten arguments. REWRITE-WITH-LEMMAS searches the LEMMAS property of *fn*; SETUP has previously arranged that all of the equations whose lefthand sides have *fn* as their outermost function symbol are stored there. One at a time, the equations are examined, and for each such equation lhs = rhs, an attempt is made, using the function ONE-WAY-UNIFY to find a substitution σ such that lhs_σ is equal to the term passed to REWRITE-WITH-LEMMAS. If such a σ is found, then ONE-WAY-UNIFY returns T and the substitution is found in the variable UNIFY-SUBST. In case of a successful unification, REWRITE is called on the result of instantiating rhs with the substitution. (The unification algorithm used is akin to a reasonably efficient version [4] of Robinson's unification algorithm [3].)

4 Compiling the Rewriting Operation

In Appendix B, a rewrite system is presented which performs the same computation as that performed by REWRITE in the Gabriel benchmark but (with both tests run on a Symbolics Lisp Machine) about 350 times faster. The secret of the speed up is simply compilation. This should not be confused with the compilation versus interpretation of Lisp functions such as REWRITE; in all timing tests discussed in this report, we are assuming that all the Lisp functions in sight have been compiled. We are not using the Lisp interpreter at all!

4.1 An Example

Perhaps one reason that the compilation of theorem-proving operations has received so little discussion in the scientific literature is that felicitous descriptions of the idea have not been discovered. Instead of starting with an abstract description of the compilation process, let us instead jump into the middle of things.

After the rewrite rules on *RULES* have been compiled by invoking (RCP-SETUP) there exists a Lisp function RCP:QUOTIENT whose definition is:

```
(DEFUN RCP:QUOTIENT (V1 V2)
  (COND ((AND (CONSP V1)
```

```
      (EQ 'TIMES (CAR V1))
      (EQUAL V2 (CAR (CDR V1))))
 (RCP:IF (RCP:ZEROP (CAR (CDR V1)))
         (RCP:ZERO)
         (RCP:FIX (CAR (CDR (CDR V1))))))
((AND (CONSP V1)
      (EQ 'PLUS (CAR V1))
      (CONSP (CAR (CDR (CDR V1))))
      (EQ 'PLUS (CAR (CAR (CDR (CDR V1)))))
      (EQUAL (CAR (CDR (CAR (CDR (CDR V1)))))
             (CAR (CDR V1)))
      (CONSP V2)
      (EQ 'TWO (CAR V2)))
 (RCP:PLUS (CAR (CDR V1))
           (RCP:QUOTIENT
             (CAR (CDR (CDR (CAR (CDR (CDR V1))))))
             (RCP:TWO))))
(T (CONS 'QUOTIENT (LIST V1 V2)))))
```

First note that many of the symbols in this definition are in a special Common Lisp package RCP of our own creation. Why do we engage in this apparently awkward naming convention? We do it in order to avoid collisions with the normal Lisp functions. For example, although IF is one of the function symbols in our rewrite rules, it would create chaos if we were to redefine IF, so therefore the rewrite compiler defines functions for symbols such as RCP:QUOTIENT and RCP:IF instead of QUOTIENT and IF.

Here again, for the sake of discussion, are the equations about QUOTIENT:

```
((EQUAL (QUOTIENT (TIMES Y X) Y)
        (IF (ZEROP Y) (ZERO) (FIX X)))
 (EQUAL (QUOTIENT (PLUS X (PLUS X Y)) (TWO))
        (PLUS X (QUOTIENT Y (TWO)))))
```

What purpose does the function RCP:QUOTIENT fulfill? Imagine that we are in the middle of rewriting a large expression, that we are currently rewriting an expression that starts with QUOTIENT, and that we have rewritten the arguments. If we invoke RCP:QUOTIENT on the two rewritten arguments, then three conditions of the COND are encountered. The first two conditions correspond exactly to the unifications that would occur with the two rules about QUOTIENT. For example, the first condition checks that the first argument is a TIMES expression and that the second argument is the first argument of the TIMES expression. What happens if the first condition is satisfied? We invoke:

```
(RCP:IF (RCP:ZEROP (CAR (CDR V1)))
        (RCP:ZERO)
        (RCP:FIX (CAR (CDR (CDR V1))))).
```

Under similar assumptions about the RCP package functions for IF, ZEROP, ZERO, and FIX, the result of the invocation will be exactly the same as the result of rewriting the expression (IF (ZEROP Y) (ZERO) (FIX X)) after first substituting in the unifying substitution that would have been produced by ONE-WAY-UNIFY.

The last clause of RCP:QUOTIENT handles the case in which neither equation is applicable; we simply return a term that begins with QUOTIENT and has as arguments the rewritten arguments.

4.2 The Idea

From the foregoing example, the main idea here should be obvious: we seek to avoid the overhead of *interpreting* the rewrite rules with functions such as REWRITE-WITH-LEMMAS and ONE-WAY-UNIFY in exactly the same way that a Lisp compiler permits one to avoid the overhead of interpreting Lisp function definitions.

The objective that RCP-SETUP achieves is to define for each of the function symbols *fn* of the terms involved in the rewriting process a new Lisp function RCP:*fn* such that:

> If *args* is a list of rewritten terms, then applying RCP:*fn* to *args* will return the same thing as calling REWRITE-WITH-LEMMAS on the result of consing *fn* on to *args* will return.

4.3 The Creation of the Compiled Rules

How are these compiled functions computed? The key function is:

```
(DEFUN COMPILE-RULES (FN RULES &AUX ARGS LHS RHS)
  (MATCH! (CAR RULES) (EQUAL (CONS FN ARGS) &))
  (SETQ ARGS (MAKE-VARS (LENGTH ARGS)))
  `(DEFUN ,(RCP-INTERN FN) ,ARGS
     (COND
       ,@(LOOP FOR RULE IN RULES
               DO
           (SETQ *A-LIST* NIL *CONDITIONS* NIL)
               DO
           (MATCH! RULE (EQUAL LHS RHS))
```

```
        DO
    (LOOP FOR TERM IN (CDR LHS) AS V IN ARGS
        DO (CONDITIONS TERM V))
        COLLECT
          '((AND ,@(REVERSE *CONDITIONS*))
            ,(RCP-SUBST *A-LIST* RHS)))
(T (CONS (QUOTE ,FN) (LIST ,@ARGS))))))
```

COMPILE-RULES returns a DEFUN form, which when evaluated will define a function, and that function can then be compiled. COMPILE-RULES takes as its two arguments a function symbol FN and some rules about that function symbol, the same rules that would have been stored under the LEMMAS property of FN. The DEFUN form that COMPILE-RULES will return will be a definition of RCP:*fn*, where *fn* is the value of FN. The body of the DEFUN will be a COND. The last condition of the COND is the trivial case; FN is simply consed on to the arguments.

Each of the other clauses of the COND corresponds to one of the equations in RULES. For each such clause, there is a test and a value; the test is computed by calling CONDITIONS sequentially on the arguments of the lefthand side of the equation. Here is the definition of CONDITIONS:

```
(DEFUN CONDITIONS (TERM NAME &AUX TMP)
  (COND ((ATOM TERM)
         (COND ((SETQ TMP (ASSOC TERM *A-LIST*))
                (PUSH '(EQUAL ,NAME ,(CDR TMP))
                      *CONDITIONS*))
               (T (PUSH (CONS TERM NAME) *A-LIST*))))
        (T (PUSH '(CONSP ,NAME) *CONDITIONS*)
           (PUSH '(EQ (QUOTE ,(CAR TERM)) (CAR ,NAME))
                 *CONDITIONS*)
           (LOOP FOR TM IN (CDR TERM) DO
             (PROGN (SETQ NAME '(CDR ,NAME))
                    (CONDITIONS TM '(CAR ,NAME)))))))
```

CONDITIONS will deposit on the special variable *CONDITIONS* the sequence of tests that are necessary to perform the unification. However, during this compiled unification process, no attempt is made to accumulate a substitution (cf. the setting of UNIFY-SUBST in the execution of ONE-WAY-UNIFY). After all, at compilation time, we do not even know with which terms we are going to be unifying; we only have in hand the equations, not the terms to be rewritten. Rather, whenever a variable is encountered in the lefthand side of the equation, we check whether it has been encountered before. (1) If it has not, we place a note on the a-list *A-LIST* consisting of the pair of (a) the variable and (b)

an expression that, when evaluated at runtime, will return the term with which the variable would have been bound on UNIFY-SUBST. (2) If it has, then there is a note on *A-LIST*. In this case, we lay down a test that requires the EQUALity of the expression we will be encountering here at runtime with the expression we encountered at runtime which would have been bound on UNIFY-SUBST, to the variable in question.

The purpose of accumulating *A-LIST* is seen in the way that COMPILE-RULES forms the value part of the corresponding clause of the COND: we substitute *A-LIST* into the righthand side of the equation, while renaming all of the function symbols of the righthand side to be in the package RCP.

4.4 Further Enhancements

The code produced by the compiler here is far from optimal. Since we are dealing with Pure Lisp, we can certainly rearrange the text to take note of common subexpressions and to eliminate redundant tests. Furthermore, we could use some sort of "indexing," as it is called in the Prolog literature, to dispatch rapidly using hashing techniques when we do get information about function symbols encountered. We could even macro-expand some of the RCP: functions, in effect applying the rewrite rules to themselves at compile time.

5 Results and *Caveats*

In repeated tests on a Symbolics 3640 with 4 megabytes of RAM, 2 140 megabyte disks, and with (GC-ON), we have found that execution of (TIME (PROGN (SETQ *RT* (REWRITE *TERM*)) NIL)) takes about 80 seconds whereas (TIME (PROGN (SETQ *RC-RT* (RCP-REWRITE *TERM*)) NIL)) takes about .23 seconds. The values of *RT* and *RC-RT* returned are, of course, EQUAL. The speed-up obtained is thus about a factor of 350.

In any compilation based system, one has to consider the cost of compiling. Invoking (RCP-SETUP) requires about 20 seconds. Of course, compilation is only necessary when one changes rewrite rules. Furthermore, in the particular strategy of rewrite compilation discussed here, only the function which embodies the rewrite rules about a particular function needs to be redefclbobi ined and recompiled when a rule about that function is changed.

A The "Boyer" Gabriel Benchmark

We have slightly modified the benchmark that appears in Gabriel's book [2] in the following ways: (a) we have introduced the top level variables ***RULES*** and ***TERM*** only in order to make those objects readily accessible for use by the functions in Appendix B and (b) we have cleaned up a few typographic errors in the rules so that all functions everywhere take the same number of arguments and so that integers do not occur where terms are expected. The defined functions are unmodified.

In order to use this code, you need to first invoke **(SETUP)**, which puts the ***RULES*** on property lists.

```
;;;  -*- Syntax: Common-lisp; Package: USER; Base: 10 -*-

(DEFVAR UNIFY-SUBST)

(DEFVAR TEMP-TEMP)

(DEFUN MEMBER-EQUAL (X L)
  (DOLIST (Y L) (COND ((EQUAL X Y)
                       (RETURN-FROM MEMBER-EQUAL T)))))

(DEFUN ADD-LEMMA (TERM)
       (COND ((AND (NOT (ATOM TERM))
                   (EQ (CAR TERM)
                       (QUOTE EQUAL))
                   (NOT (ATOM (CADR TERM))))
              (SETF (GET (CAR (CADR TERM))
                         (QUOTE LEMMAS))
                    (CONS TERM (GET (CAR (CADR TERM))
                                    (QUOTE LEMMAS)))))
             (T (ERROR "ADD-LEMMA-DID-NOT-LIKE-TERM"
                       TERM))))

(DEFUN ADD-LEMMA-LST (LST)
       (COND ((NULL LST)
              T)
             (T (ADD-LEMMA (CAR LST))
                (ADD-LEMMA-LST (CDR LST)))))
```

```
(DEFUN APPLY-SUBST (ALIST TERM)
       (COND ((ATOM TERM)
              (COND ((SETQ TEMP-TEMP (ASSOC TERM ALIST))
                     (CDR TEMP-TEMP))
                    (T TERM)))
             (T (CONS (CAR TERM)
                      (APPLY-SUBST-LST ALIST (CDR TERM))))))


(DEFUN APPLY-SUBST-LST (ALIST LST)
       (COND ((NULL LST)
              NIL)
             (T (CONS (APPLY-SUBST ALIST (CAR LST))
                      (APPLY-SUBST-LST ALIST (CDR LST))))))


(DEFUN FALSEP (X LST)
       (OR (EQUAL X (QUOTE (F)))
           (MEMBER-EQUAL X LST)))


(DEFUN ONE-WAY-UNIFY (TERM1 TERM2)
       (PROGN (SETQ UNIFY-SUBST NIL)
              (ONE-WAY-UNIFY1 TERM1 TERM2)))


(DEFUN ONE-WAY-UNIFY1 (TERM1 TERM2)
       (COND ((ATOM TERM2)
              (COND ((SETQ TEMP-TEMP
                           (ASSOC TERM2 UNIFY-SUBST))
                     (EQUAL TERM1 (CDR TEMP-TEMP)))
                    (T (SETQ UNIFY-SUBST
                             (CONS (CONS TERM2 TERM1)
                                   UNIFY-SUBST))
                       T)))
             ((ATOM TERM1)
              NIL)
             ((EQ (CAR TERM1)
                  (CAR TERM2))
              (ONE-WAY-UNIFY1-LST (CDR TERM1)
                                  (CDR TERM2)))
             (T NIL)))
```

```
(DEFUN ONE-WAY-UNIFY1-LST (LST1 LST2)
       (COND ((NULL LST1)
              T)
             ((ONE-WAY-UNIFY1 (CAR LST1)
                              (CAR LST2))
              (ONE-WAY-UNIFY1-LST (CDR LST1)
                                  (CDR LST2)))
             (T NIL)))


(DEFUN REWRITE (TERM)
       (COND ((ATOM TERM)
              TERM)
             (T (REWRITE-WITH-LEMMAS
                 (CONS (CAR TERM)
                       (REWRITE-ARGS (CDR TERM)))
                 (GET (CAR TERM)
                      (QUOTE LEMMAS))))))

(DEFUN REWRITE-ARGS (LST)
       (COND ((NULL LST)
              NIL)
             (T (CONS (REWRITE (CAR LST))
                      (REWRITE-ARGS (CDR LST))))))

(DEFUN REWRITE-WITH-LEMMAS (TERM LST)
       (COND ((NULL LST)
              TERM)
             ((ONE-WAY-UNIFY TERM (CADR (CAR LST)))
              (REWRITE (APPLY-SUBST UNIFY-SUBST
                                    (CADDR (CAR LST)))))
             (T (REWRITE-WITH-LEMMAS TERM (CDR LST)))))


(DEFUN
  SETUP NIL
  (ADD-LEMMA-LST *RULES*))


(DEFUN TAUTOLOGYP (X TRUE-LST FALSE-LST)
       (COND ((TRUEP X TRUE-LST)
              T)
             ((FALSEP X FALSE-LST)
```

```
            NIL)
           ((ATOM X)
            NIL)
           ((EQ (CAR X)
                (QUOTE IF))
            (COND ((TRUEP (CADR X)
                          TRUE-LST)
                   (TAUTOLOGYP (CADDR X)
                               TRUE-LST FALSE-LST))
                  ((FALSEP (CADR X)
                           FALSE-LST)
                   (TAUTOLOGYP (CADDDR X)
                               TRUE-LST FALSE-LST))
                  (T (AND (TAUTOLOGYP (CADDR X)
                                      (CONS (CADR X)
                                            TRUE-LST)
                                      FALSE-LST)
                          (TAUTOLOGYP (CADDDR X)
                                      TRUE-LST
                                      (CONS
                                       (CADR X)
                                       FALSE-LST))))))
            (T NIL)))

(DEFUN TAUTP (X)
       (TAUTOLOGYP (REWRITE X)
                   NIL NIL))


(DEFUN TRUEP (X LST)
       (OR (EQUAL X (QUOTE (T)))
           (MEMBER-EQUAL X LST)))


(DEFUN TEST NIL
       (TAUTP *TERM*))


(DEFVAR *TERM* (APPLY-SUBST
                (QUOTE ((X G (PLUS (PLUS A B)
                                   (PLUS C (ZERO))))
                        (Y G (TIMES (TIMES A B)
                                    (PLUS C D)))
                        (Z G (REVERSE (APPEND (APPEND A B)
                                              (NIL))))
```

```
                        (U EQUAL (PLUS A B)
                           (DIFFERENCE X Y))
                        (W LESSP (REMAINDER A B)
                           (MEMBER A (LENGTH B)))))
                (QUOTE (IMPLIES (AND (IMPLIES X Y)
                                     (AND
                                      (IMPLIES Y Z)
                                      (AND
                                       (IMPLIES Z U)
                                       (IMPLIES U W))))
                               (IMPLIES X W)))))

(DEFVAR *RULES*
        (QUOTE ((EQUAL (COMPILE FORM)
                       (REVERSE (CODEGEN (OPTIMIZE FORM)
                                         (NIL))))
                (EQUAL (EQP X Y)
                       (EQUAL (FIX X)
                              (FIX Y)))
                (EQUAL (GREATERP X Y)
                       (LESSP Y X))
                (EQUAL (LESSEQP X Y)
                       (NOT (LESSP Y X)))
                (EQUAL (GREATEREQP X Y)
                       (NOT (LESSP X Y)))
                (EQUAL (BOOLEAN X)
                       (OR (EQUAL X (T))
                           (EQUAL X (F))))
                (EQUAL (IFF X Y)
                       (AND (IMPLIES X Y)
                            (IMPLIES Y X)))
                (EQUAL (EVEN1 X)
                       (IF (ZEROP X)
                           (T)
                           (ODD (SUB1 X))))
                (EQUAL (COUNTPS- L PRED)
                       (COUNTPS-LOOP L PRED (ZERO)))
                (EQUAL (FACT- I)
                       (FACT-LOOP I (ONE)))
                (EQUAL (REVERSE- X)
                       (REVERSE-LOOP X (NIL)))
                (EQUAL (DIVIDES X Y)
                       (ZEROP (REMAINDER Y X)))
                (EQUAL (ASSUME-TRUE VAR ALIST)
```

```
       (CONS (CONS VAR (T))
             ALIST))
(EQUAL (ASSUME-FALSE VAR ALIST)
       (CONS (CONS VAR (F))
             ALIST))
(EQUAL (TAUTOLOGY-CHECKER X)
       (TAUTOLOGYP (NORMALIZE X)
                   (NIL)))
(EQUAL (FALSIFY X)
       (FALSIFY1 (NORMALIZE X)
                 (NIL)))
(EQUAL (PRIME X)
       (AND (NOT (ZEROP X))
            (AND (NOT (EQUAL X
                             (ADD1 (ZERO))))
                 (PRIME1 X (SUB1 X)))))
(EQUAL (AND P Q)
       (IF P (IF Q (T)
                   (F))
           (F)))
(EQUAL (OR P Q)
       (IF P (T)
           (IF Q (T)
               (F))))
(EQUAL (NOT P)
       (IF P (F)
           (T)))
(EQUAL (IMPLIES P Q)
       (IF P (IF Q (T)
                   (F))
           (T)))
(EQUAL (FIX X)
       (IF (NUMBERP X)
           X
           (ZERO)))
(EQUAL (IF (IF A B C)
           D E)
       (IF A (IF B D E)
           (IF C D E)))
(EQUAL (ZEROP X)
       (OR (EQUAL X (ZERO))
           (NOT (NUMBERP X))))
(EQUAL (PLUS (PLUS X Y)
             Z)
```

```
       (PLUS X (PLUS Y Z)))
(EQUAL (EQUAL (PLUS A B)
              (ZERO))
       (AND (ZEROP A)
            (ZEROP B)))
(EQUAL (DIFFERENCE X X)
       (ZERO))
(EQUAL (EQUAL (PLUS A B)
              (PLUS A C))
       (EQUAL (FIX B)
              (FIX C)))
(EQUAL (EQUAL (ZERO)
              (DIFFERENCE X Y))
       (NOT (LESSP Y X)))
(EQUAL (EQUAL X (DIFFERENCE X Y))
       (AND (NUMBERP X)
            (OR (EQUAL X (ZERO))
                (ZEROP Y))))
(EQUAL (MEANING (PLUS-TREE (APPEND X Y))
                A)
       (PLUS (MEANING (PLUS-TREE X)
                      A)
             (MEANING (PLUS-TREE Y)
                      A)))
(EQUAL (MEANING (PLUS-TREE (PLUS-FRINGE X))
                A)
       (FIX (MEANING X A)))
(EQUAL (APPEND (APPEND X Y)
               Z)
       (APPEND X (APPEND Y Z)))
(EQUAL (REVERSE (APPEND A B))
       (APPEND (REVERSE B)
               (REVERSE A)))
(EQUAL (TIMES X (PLUS Y Z))
       (PLUS (TIMES X Y)
             (TIMES X Z)))
(EQUAL (TIMES (TIMES X Y)
              Z)
       (TIMES X (TIMES Y Z)))
(EQUAL (EQUAL (TIMES X Y)
              (ZERO))
       (OR (ZEROP X)
           (ZEROP Y)))
(EQUAL (EXEC (APPEND X Y)
```

```
                PDS ENVRN)
        (EXEC Y (EXEC X PDS ENVRN)
                ENVRN))
(EQUAL (MC-FLATTEN X Y)
       (APPEND (FLATTEN X)
               Y))
(EQUAL (MEMBER X (APPEND A B))
       (OR (MEMBER X A)
           (MEMBER X B)))
(EQUAL (MEMBER X (REVERSE Y))
       (MEMBER X Y))
(EQUAL (LENGTH (REVERSE X))
       (LENGTH X))
(EQUAL (MEMBER A (INTERSECT B C))
       (AND (MEMBER A B)
            (MEMBER A C)))
(EQUAL (NTH (ZERO)
            I)
       (ZERO))
(EQUAL (EXP I (PLUS J K))
       (TIMES (EXP I J)
              (EXP I K)))
(EQUAL (EXP I (TIMES J K))
       (EXP (EXP I J)
            K))
(EQUAL (REVERSE-LOOP X Y)
       (APPEND (REVERSE X)
               Y))
(EQUAL (REVERSE-LOOP X (NIL))
       (REVERSE X))
(EQUAL (COUNT-LIST Z (SORT-LP X Y))
       (PLUS (COUNT-LIST Z X)
             (COUNT-LIST Z Y)))
(EQUAL (EQUAL (APPEND A B)
              (APPEND A C))
       (EQUAL B C))
(EQUAL (PLUS (REMAINDER X Y)
             (TIMES Y (QUOTIENT X Y)))
       (FIX X))
(EQUAL (POWER-EVAL (BIG-PLUS1 L I BASE)
                   BASE)
       (PLUS (POWER-EVAL L BASE)
             I))
(EQUAL (POWER-EVAL (BIG-PLUS X Y I BASE)
```

```
                    BASE)
       (PLUS I (PLUS (POWER-EVAL X BASE)
                     (POWER-EVAL Y BASE))))
(EQUAL (REMAINDER Y (ONE))
       (ZERO))
(EQUAL (LESSP (REMAINDER X Y)
              Y)
       (NOT (ZEROP Y)))
(EQUAL (REMAINDER X X)
       (ZERO))
(EQUAL (LESSP (QUOTIENT I J)
              I)
       (AND (NOT (ZEROP I))
            (OR (ZEROP J)
                (NOT (EQUAL J (ONE))))))
(EQUAL (LESSP (REMAINDER X Y)
              X)
       (AND (NOT (ZEROP Y))
            (AND (NOT (ZEROP X))
                 (NOT (LESSP X Y)))))
(EQUAL (POWER-EVAL (POWER-REP I BASE)
                   BASE)
       (FIX I))
(EQUAL (POWER-EVAL
        (BIG-PLUS (POWER-REP I BASE)
                  (POWER-REP J BASE)
                  (ZERO)
                  BASE)
         BASE)
       (PLUS I J))
(EQUAL (GCD X Y)
       (GCD Y X))
(EQUAL (NTH (APPEND A B)
            I)
       (APPEND
        (NTH A I)
        (NTH B (DIFFERENCE I
                           (LENGTH A)))))
(EQUAL (DIFFERENCE (PLUS X Y)
                   X)
       (FIX Y))
(EQUAL (DIFFERENCE (PLUS Y X)
                   X)
       (FIX Y))
```

```
(EQUAL (DIFFERENCE (PLUS X Y)
                   (PLUS X Z))
       (DIFFERENCE Y Z))
(EQUAL (TIMES X (DIFFERENCE C W))
       (DIFFERENCE (TIMES C X)
                   (TIMES W X)))
(EQUAL (REMAINDER (TIMES X Z)
                  Z)
       (ZERO))
(EQUAL (DIFFERENCE (PLUS B (PLUS A C))
                   A)
       (PLUS B C))
(EQUAL (DIFFERENCE (ADD1 (PLUS Y Z))
                   Z)
       (ADD1 Y))
(EQUAL (LESSP (PLUS X Y)
              (PLUS X Z))
       (LESSP Y Z))
(EQUAL (LESSP (TIMES X Z)
              (TIMES Y Z))
       (AND (NOT (ZEROP Z))
            (LESSP X Y)))
(EQUAL (LESSP Y (PLUS X Y))
       (NOT (ZEROP X)))
(EQUAL (GCD (TIMES X Z)
            (TIMES Y Z))
       (TIMES Z (GCD X Y)))
(EQUAL (VALUE (NORMALIZE X)
              A)
       (VALUE X A))
(EQUAL (EQUAL (FLATTEN X)
              (CONS Y (NIL)))
       (AND (NLISTP X)
            (EQUAL X Y)))
(EQUAL (LISTP (GOPHER X))
       (LISTP X))
(EQUAL (SAMEFRINGE X Y)
       (EQUAL (FLATTEN X)
              (FLATTEN Y)))
(EQUAL (EQUAL (GREATEST-FACTOR X Y)
              (ZERO))
       (AND (OR (ZEROP Y)
                (EQUAL Y (ONE)))
            (EQUAL X (ZERO))))
```

```
(EQUAL (EQUAL (GREATEST-FACTOR X Y)
              (ONE))
       (EQUAL X (ONE)))
(EQUAL (NUMBERP (GREATEST-FACTOR X Y))
       (NOT (AND (OR (ZEROP Y)
                     (EQUAL Y (ONE)))
                 (NOT (NUMBERP X)))))
(EQUAL (TIMES-LIST (APPEND X Y))
       (TIMES (TIMES-LIST X)
              (TIMES-LIST Y)))
(EQUAL (PRIME-LIST (APPEND X Y))
       (AND (PRIME-LIST X)
            (PRIME-LIST Y)))
(EQUAL (EQUAL Z (TIMES W Z))
       (AND (NUMBERP Z)
            (OR (EQUAL Z (ZERO))
                (EQUAL W (one)))))
(EQUAL (GREATEREQPR X Y)
       (NOT (LESSP X Y)))
(EQUAL (EQUAL X (TIMES X Y))
       (OR (EQUAL X (ZERO))
           (AND (NUMBERP X)
                (EQUAL Y (ONE)))))
(EQUAL (REMAINDER (TIMES Y X)
                  Y)
       (ZERO))
(EQUAL (EQUAL (TIMES A B)
              (ONE))
       (AND
        (NOT (EQUAL A (ZERO)))
        (AND
         (NOT (EQUAL B (ZERO)))
         (AND
          (NUMBERP A)
          (AND
           (NUMBERP B)
           (AND
            (EQUAL (SUB1 A)
                   (ZERO))
            (EQUAL (SUB1 B)
                   (ZERO)))))))))
(EQUAL (LESSP (LENGTH (DELETE X L))
              (LENGTH L))
       (MEMBER X L))
```

```
(EQUAL (SORT2 (DELETE X L))
       (DELETE X (SORT2 L)))
(EQUAL (DSORT X)
       (SORT2 X))
(EQUAL (LENGTH
        (CONS
         X1
         (CONS
          X2
          (CONS
           X3 (CONS
               X4
               (CONS
                X5
                (CONS X6 X7)))))))
       (PLUS (SIX) (LENGTH X7)))
(EQUAL (DIFFERENCE (ADD1 (ADD1 X))
                   (TWO))
       (FIX X))
(EQUAL (QUOTIENT (PLUS X (PLUS X Y))
                 (TWO))
       (PLUS X (QUOTIENT Y (TWO))))
(EQUAL (SIGMA (ZERO)
              I)
       (QUOTIENT (TIMES I (ADD1 I))
                 (TWO)))
(EQUAL (PLUS X (ADD1 Y))
       (IF (NUMBERP Y)
           (ADD1 (PLUS X Y))
           (ADD1 X)))
(EQUAL (EQUAL (DIFFERENCE X Y)
              (DIFFERENCE Z Y))
       (IF (LESSP X Y)
           (NOT (LESSP Y Z))
           (IF (LESSP Z Y)
               (NOT (LESSP Y X))
               (EQUAL (FIX X)
                      (FIX Z)))))
(EQUAL (MEANING (PLUS-TREE (DELETE X Y))
                A)
       (IF (MEMBER X Y)
           (DIFFERENCE (MEANING
                        (PLUS-TREE Y)
                        A)
```

```
                        (MEANING X A))
              (MEANING (PLUS-TREE Y)
                       A)))
(EQUAL (TIMES X (ADD1 Y))
       (IF (NUMBERP Y)
           (PLUS X (TIMES X Y))
           (FIX X)))
(EQUAL (NTH (NIL)
            I)
       (IF (ZEROP I)
           (NIL)
           (ZERO)))
(EQUAL (LAST (APPEND A B))
       (IF (LISTP B)
           (LAST B)
           (IF (LISTP A)
               (CONS (CAR (LAST A))
                     B)
               B)))
(EQUAL (EQUAL (LESSP X Y)
              Z)
       (IF (LESSP X Y)
           (EQUAL (T) Z)
           (EQUAL (F) Z)))
(EQUAL (ASSIGNMENT X (APPEND A B))
       (IF (ASSIGNEDP X A)
           (ASSIGNMENT X A)
           (ASSIGNMENT X B)))
(EQUAL (CAR (GOPHER X))
       (IF (LISTP X)
           (CAR (FLATTEN X))
           (ZERO)))
(EQUAL (FLATTEN (CDR (GOPHER X)))
       (IF (LISTP X)
           (CDR (FLATTEN X))
           (CONS (ZERO)
                 (NIL))))
(EQUAL (QUOTIENT (TIMES Y X)
                 Y)
       (IF (ZEROP Y)
           (ZERO)
           (FIX X)))
(EQUAL (GET J (SET I VAL MEM))
       (IF (EQP J I)
```

```
VAL
(GET J MEM))))))
```

B A Compiler Based Rewriter

Here is the code that is original with this report. Compilation of the file depends upon (a) Common Lisp, (b) the Maclisp/Zetalisp **LOOP** macro, (c) the code in Appendix A, and (d) the **MATCH** and **MATCH!** macros in Appendix C.

```
(DEFUN RULE-FN (RULE &AUX FN)
  (MATCH! RULE (EQUAL (CONS FN &) &))
  FN)

(DEFUN COMPILE-RULE-LIST (RULE-LIST)
  (LOOP FOR PAIR IN (PARTITION-BY RULE-LIST #'RULE-FN)
        COLLECT
    (COMPILE-RULES (CAR PAIR) (CDR PAIR))))

(DEFUN PARTITION-BY (LST FN &OPTIONAL STASH)
  (COND ((NULL LST) STASH)
        (T (INSERT-ITEM (CAR LST)
                        FN
                        (PARTITION-BY (CDR LST)
                                      FN
                                      STASH)))))

(DEFUN INSERT-ITEM (ITEM FN STASH)
  (LET ((KEY (FUNCALL FN ITEM)))
    (COND ((NULL STASH) (LIST (CONS KEY (LIST ITEM))))
          ((EQUAL (CAAR STASH) KEY)
           (CONS (CONS KEY (CONS ITEM (CDAR STASH)))
                 (CDR STASH)))
          (T (CONS (CAR STASH)
                   (INSERT-ITEM ITEM FN (CDR STASH)))))))

(DEFUN MAKE-VARS (N)
  (LOOP FOR I FROM 1 TO N COLLECT
        (INTERN (STRING-APPEND "V" (PRIN1-TO-STRING I)))))

(DEFVAR *A-LIST*)

(DEFVAR *CONDITIONS*)

(OR (FIND-PACKAGE "REWRITE-COMPILER-PACKAGE")
    (MAKE-PACKAGE "REWRITE-COMPILER-PACKAGE"
                  :NICKNAMES '(RCP) :USE NIL))
```

```
(DEFVAR *REWRITE-COMPILER-PACKAGE*
  (FIND-PACKAGE "REWRITE-COMPILER-PACKAGE"))

(DEFUN RCP-INTERN (SYMBOL &AUX TMP)
  (SETQ TMP (INTERN (SYMBOL-NAME SYMBOL)
                    *REWRITE-COMPILER-PACKAGE*))
  (COND ((NOT (FBOUNDP TMP))
         (EVAL `(DEFUN ,TMP (&REST ARGS)
                  (COND
                   (ARGS
                    (CONS
                     (QUOTE ,SYMBOL)
                     (LOOP FOR ARG IN ARGS COLLECT ARG)))
                   (T (QUOTE (,SYMBOL))))))
         (COMPILE TMP)))
  TMP)

(DEFUN COMPILE-RULES (FN RULES &AUX ARGS LHS RHS)
  (MATCH! (CAR RULES) (EQUAL (CONS FN ARGS) &))
  (SETQ ARGS (MAKE-VARS (LENGTH ARGS)))
  `(DEFUN ,(RCP-INTERN FN) ,ARGS
     (COND
       ,@(LOOP FOR RULE IN RULES
               DO
           (SETQ *A-LIST* NIL *CONDITIONS* NIL)
               DO
           (MATCH! RULE (EQUAL LHS RHS))
               DO
           (LOOP FOR TERM IN (CDR LHS) AS V IN ARGS
               DO (CONDITIONS TERM V))
               COLLECT
                 `((AND ,@(REVERSE *CONDITIONS*))
                   ,(RCP-SUBST *A-LIST* RHS)))
       (T (CONS (QUOTE ,FN) (LIST ,@ARGS))))))

(DEFUN RCP-SUBST (L TERM &AUX TMP)
  (COND ((ATOM TERM)
         (SETQ TMP (ASSOC TERM L))
         (COND ((NULL TMP) (ERROR "Variable unbound!")))
         (CDR TMP))
        (T (CONS (RCP-INTERN (CAR TERM))
                 (LOOP FOR ARG IN (CDR TERM)
                       COLLECT (RCP-SUBST L ARG))))))
```

```
(DEFUN CONDITIONS (TERM NAME &AUX TMP)
  (COND ((ATOM TERM)
         (COND ((SETQ TMP (ASSOC TERM *A-LIST*))
                (PUSH '(EQUAL ,NAME ,(CDR TMP))
                      *CONDITIONS*))
               (T (PUSH (CONS TERM NAME) *A-LIST*))))
        (T (PUSH '(CONSP ,NAME) *CONDITIONS*)
           (PUSH '(EQ (QUOTE ,(CAR TERM)) (CAR ,NAME))
                 *CONDITIONS*)
           (LOOP FOR TM IN (CDR TERM) DO
             (PROGN (SETQ NAME '(CDR ,NAME))
                    (CONDITIONS TM '(CAR ,NAME)))))))


(DEFUN RCP-REWRITE (TERM)
  (COND ((ATOM TERM) TERM)
        (T (APPLY (RCP-INTERN (CAR TERM))
                  (LOOP FOR ARG IN (CDR TERM)
                        COLLECT (RCP-REWRITE ARG))))))


(DEFVAR *FNS*)

(RCP-INTERN 'G)



(DEFUN RCP-SETUP (&AUX FN ARGS BODY)
  (SETQ *FNS* (LOOP FOR X IN
                    (COMPILE-RULE-LIST (REVERSE *RULES*))
                    ;;  We reverse the rules only in order to
                    ;;  keep the
                    ;;  order in which the rules are
                    ;;  applied the same
                    ;;  as for the Gabriel benchmark.
                    DO (MATCH! X (DEFUN FN ARGS BODY))
                    DO (SETF (GET FN (QUOTE DEF))
                             '(LAMBDA ,ARGS ,BODY))
                    DO (EVAL X)
                    COLLECT FN))
  (LOOP FOR FN IN *FNS* DO (COMPILE FN)))
```

C Match

The following macros are used by the code in the previous appendix. This code is taken from the Boyer-Moore theorem-prover. **MATCH** is a macro that has been used in that system since about 1975. It is very similar to what has been called a "destructuring let" in the Lisp community for over a decade. In a certain sense, the idea of **MATCH** is very similar to that of the compilation of rewrite rules discussed in this note.

```
;;;  -*- Syntax: Common-lisp; Package: USER; Base: 10 -*-

;;;  This is the Common Lisp-ification of the Match stuff
;;;  from the Boyer-Moore theorem-prover.

(DEFVAR *SETQ-LST*)

(DEFVAR *TEST-LST*)

(DEFUN NCONC1 (X Y) (NCONC X (LIST Y)))

(DEFMACRO MATCH (TERM PAT) (MATCH-MACRO TERM PAT))

(DEFMACRO MATCH! (TERM PAT) (MATCH!-MACRO TERM PAT))

(DEFUN MATCH-MACRO (TERM PAT)
  (COND ((CONSP TERM)
         '(LET ((MATCH-TEMP ,TERM))
            ,(MATCH1-MACRO (QUOTE MATCH-TEMP) PAT)))
        (T (MATCH1-MACRO TERM PAT))))

(DEFUN MATCH!-MACRO (TERM PAT)
    (LIST (QUOTE OR)
          (MATCH-MACRO TERM PAT)))

(DEFUN MATCH1-MACRO (TERM PAT)
    (LET (*TEST-LST* *SETQ-LST*)
      (MATCH2-MACRO TERM PAT)
      (LIST (QUOTE COND)
            (CONS
             (CONS (QUOTE AND) *TEST-LST*)
             (NCONC1 *SETQ-LST* T)))))
```

```
(DEFUN MATCH2-MACRO (TERM PAT)
  (COND ((ATOM PAT)
         (COND ((EQ PAT (QUOTE &)) NIL)
               ((SYMBOLP PAT)
                (SETQ *SETQ-LST*
                      (NCONC1
                       *SETQ-LST*
                       (LIST (QUOTE SETQ) PAT TERM))))
               (T (SETQ *TEST-LST*
                        (NCONC1 *TEST-LST*
                                (LIST (QUOTE EQUAL)
                                      PAT TERM))))))
        ((EQ (QUOTE CONS) (CAR PAT))
         (SETQ *TEST-LST*
               (NCONC1 *TEST-LST*
                       (LIST (QUOTE CONSP) TERM)))
         (MATCH2-MACRO (LIST (QUOTE CAR) TERM) (CADR PAT))
         (MATCH2-MACRO (LIST (QUOTE CDR) TERM) (CADDR PAT)))
        ((EQ (QUOTE QUOTE) (CAR PAT))
         (COND ((SYMBOLP (CADR PAT))
                (SETQ *TEST-LST*
                      (NCONC1 *TEST-LST*
                              (LIST (QUOTE EQ)
                                    (LIST (QUOTE QUOTE)
                                          (CADR PAT))
                                    TERM))))
               (T (SETQ *TEST-LST*
                        (NCONC1 *TEST-LST*
                                (LIST (QUOTE EQUAL)
                                      (LIST (QUOTE QUOTE)
                                            (CADR PAT))
                                      TERM))))))
        (T (COND ((NEQ (CAR PAT)
                       (QUOTE LIST))
                  (SETQ PAT (CONS (QUOTE LIST)
                                  (CONS (LIST (QUOTE QUOTE)
                                              (CAR PAT))
                                        (CDR PAT))))))
           (LOOP FOR SUBPAT IN (CDR PAT)
                 DO (SETQ *TEST-LST*
                          (NCONC1 *TEST-LST*
                                  (LIST (QUOTE CONSP)
                                        TERM)))
                 (MATCH2-MACRO (LIST (QUOTE CAR)
```

```
                                TERM)
                            SUBPAT)
        (SETQ TERM (LIST (QUOTE CDR)
                         TERM)))
(SETQ *TEST-LST*
      (NCONC1 *TEST-LST* (LIST (QUOTE EQ)
                               TERM NIL))))))
```

D Index of Lisp Functions, Macros, and Variables

```
TAUTOLOGYP 11
TAUTP 12
TEMP-TEMP 9
TEST 12
TRUEP 12
UNIFY-SUBST 9
```

References

[1] Robert S. Boyer and J Strother Moore. *A Computational Logic.* Academic Press, 1971.

[2] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems.* MIT Press, 1985.

[3] J. A. Robinson. *A Machine-oriented Logic Based on the Resolution Principle.* JACM 12:1. pp. 23-41. 1965.

[4] J. A. Robinson. *Computational Logic: The unification algorithm.* in *Machine Intelligence 6*, eds. B. Meltzer and D. Michie, pp. 63-72, Edinburgh University Press, 1971.