On Automatically Generating and Using Examples in a Computational Logic System

@shortTitle(Generating and Using Examples)

@authorbox(Myung Won Kim)

@reportnumbox(Technical Report #57 March 1987)

The contents of this technical report originally appeared as the author's dissertation.

foo

Throw this page away. It’s here solely for the numbering.

1. Introduction

Examples are important in Artificial Intelligence. They are, in particular, critical to automated reasoning and machine learning - they illustrate abstract notions and complicated procedures, provide us with a source of information on which ideas and concepts are developed, and serve as a tool with which hypotheses can be validated. It has also been shown that examples can be effectively used in theorem proving. Examples are used in guiding proof search by pruning unpromising reasoning paths. Early work has demonstrated that using examples in this way substantially reduces the proof search space.

This thesis describes research on automatic example generation and applications of examples in a formal domain, the Boyer-Moore theory. We have implemented a system which automatically generates examples for a constraint stated in the theory and we have experimented with the system in the Boyer-Moore theorem prover.

1.1 An Overview

Examples are, in general, a very useful tool in Artificial Intelligence. Many machine learning systems [52, 33, 10, 14, 32] use examples for the tasks of generating concepts and conjectures. For instance, Winston's system [52] learns structural descriptions from examples. His system is presented with training instances - positive examples and near misses - of a structure (concept) to be learned such as an "arch" and it generates a description of the structure. A training instance is a line drawing of a certain structure and it is converted by the system into a semantic-network representation. Such representations of training instances are used to modify the current representation of the concept in such a way that the new representation accepts the positive examples but rejects the near misses.

Lenat's AM [33] automatically discovers concepts and conjectures in number theory from some concepts of elementary set theory. Discovery in AM is guided by heuristic rules; examples play an important role in AM. Being guided by some of the heuristic rules, examples are generated when a new concept is generated and new concepts and conjectures are generated using examples. For instance, AM discovers the concept "prime" as an interesting extreme case of the previously generated concept "divisors-of". After the concept divisors-of is created, AM attempts to fill in examples of the concept; some of them would be like divisors-of(2) = {1 2}, divisors-of(4) = {1 2 4}, divisors-of(5) = {1 5}, divisors-of(6) = {1 2 3 6}, and divisors-of(12) = {1 2 3 4 6 12}. AM finds that numbers which have very small sets of divisors-of are interesting. No numbers are found to have 0 divisors; only one number is found to have 1 divisor; several numbers are found to have two divisors and this case develops into the discovery of the concept "prime". In AM what constitutes an example is not clearly defined but they are represented in terms of LISP objects such as numbers, literal atoms, and lists.

Early work has also shown that examples can be effectively used in theorem proving [19, 20, 40, 2, 4]. Gelernter's geometry machine [19, 20] used examples in automatically proving theorems in Euclidean geometry[1]. Gelernter has experimented with the idea that humans often draw diagrams when trying to prove geometry theorems. Given a theorem to be proved the geometry machine draws a diagram representing the theorem statement. For example, suppose we are given the following theorem to be

[1]Implemented is an ad hoc formal system of Euclidean geometry as opposed to those axiomatized systems of Tarski or Hilbert. It has been reported that the machine produced the proofs of more than fifty geometry theorems comparable to those of a high school student.

proved:

```
Two vertices of a triangle are equidistant from the median
to the sides determined by those vertices [19, 20].
```

The geometry machine draws a diagram[2] which pictorially looks like figure 1-1. Now the system

Figure 1-1: A Geometry Diagram

generates several (or) subgoals for the top-goal of proving the theorem. The generated subgoals would be[3]:

1. triangle ABD is congruent to triangle ACE;
2. triangle ABD is congruent to triangle CME;
3. triangle BMD is congruent to triangle ACE;
4. triangle BMD is congruent to triangle CME.

Among those generated subgoals the first three will be rejected as false because they are not valid in the diagram. By backward reasoning with the last subgoal as the only plausible one, more new subgoals would be generated and each of them is to be checked against the diagram. Again those subgoals that are not valid in the diagram are rejected. This process continues until any of the (or) subgoals is immediately inferred from the premises of the theorem being proved and axioms. It has been reported that the use of diagrams in this way reduces the proof search space by many orders of magnitude [19, 20].

1.2 Examples in a Formal Domain

In the above we have seen that examples are represented in different forms and they are used in different ways. Now consider the domain of (natural) numbers and literal atoms, and lists. We represent numbers, literal atoms, and lists as is usually done in LISP. For instance, 0, 2, 5, 8, 11, and 119 are examples of numbers; 'A, 'ABC, 'E, and 'NIL are examples of literal atoms; '(A), '((B) E), '(A B C), '(NIL

[2]The diagram is internally represented in the form of a list of coordinates representing points named in the theorem. Coordinates are chosen at random to reflect the maximum possible generality of the diagram.

[3]The actual number of the generated subgoals will be far more than four because for each triangle in the figure has many different internal representations, for example, triangle ABD, triangle BDA, and triangle DAB refer to the same triangle in the figure.

NIL NIL), and '((D) E ((F)) 3 . A) are examples of lists. We assume that for each of the object types there is an appropriate recognizer which determines whether a given object is of the corresponding type of the recognizer; the recognizers for numbers, literal atoms, and lists are NUMBERP, LITATOM, and LISTP, respectively. (NUMBERP 5) evaluates to T whereas (NUMBERP 'A) evaluates to F; (LITATOM 'A) evaluates to T but (LITATOM '(A)) evaluate to F; (LISTP '(A B C)) evaluates to T, while (LISTP 3) and (LISTP 'ABC) both evaluate to F.

Now consider the functions LENGTH, APPEND, and REVERSE defined as follows:

```
(LENGTH L) =
  (IF (LISTP L)
      (ADD1 (LENGTH (CDR L)))
      0),

(APPEND X Y) =
  (IF (LISTP X)
      (CONS (CAR X) (APPEND (CDR X) Y))
      Y), and

(REVERSE L) =
  (IF (LISTP L)
      (APPEND (REVERSE (CDR L)) (CONS (CAR L) NIL))
      NIL).
```

The function LENGTH computes the length of its argument - if the argument is not a list, the length of the argument is 0; if the argument is a list, then the length of the argument is one larger than the length of the tail of the argument. The function APPEND concatenates two arguments - if the first argument is not a list, APPEND returns the second argument; if the first argument is a list, APPEND CONSes the head of the first argument to the concatenation of the tail of the first argument and the second argument. The function REVERSE reverses its argument - if the argument is not a list, REVERSE returns NIL; if the argument is a list, REVERSE concatenates the REVERSE of the tail of the argument and the single list containing the head of the argument. Now we can have

```
(LENGTH 'A) = 0,
(LENGTH '(B (D) . E)) = 2,
(LENGTH '(A B C)) = 3, and
(LENGTH '(2 (3) ((4)) 5)) = 4;

(APPEND 'A 'B) = 'B,
(APPEND '(A B) '(C)) = '(A B C), and
(APPEND '(B D . C) '(E . A)) = '(B D E . A);

(REVERSE 'D) = NIL,
(REVERSE '(C B)) = '(B C),
(REVERSE '(C (D) E . B)) = '(E (D) C), and
(REVERSE '(A B A)) = '(A B A).
```

(LENGTH '(A B C)) = 3 indicates that '(A B C) is an example of a list of length 3; (REVERSE '(A B A)) = '(A B A) indicates that '(A B A) is an example of a list which reverses to itself. Suppose now we generate examples of lists of length 3 which reverse to themselves. The constraint that the examples need to satisfy can be stated as:

(LENGTH L) = 3 and **(REVERSE L) = L.**

Appropriate assignment of constant values to the variable L which cause the constraint to evaluate to T can be considered examples. Now we will describe how we can generate examples satisfying the constraint.

First, we test some known examples of REVERSE and LENGTH such as those we have shown in the above. For instance, the equality (REVERSE '(A B A)) = '(A B A) suggests that the assignment of '(A B A) to the variable L satisfies both of the conditions in the constraint, thus L = '(A B A) is an example of the constraint. The assignment of '(A B C) to L satisfies the first condition but not the second condition whereas the assignment of NIL satisfies the second condition but not the first condition, thus neither of them are examples of the constraint.

Also, we can create examples using the definitions of the concepts involved. Consider the first condition (LENGTH L) = 3 and the definition of LENGTH. We expand the definition of LENGTH in the constraint to have

(LISTP L), (ADD1 (LENGTH (CDR L))) = 3 and **(REVERSE L) = L,**

which can be simplified using the fact that (ADD1 x) = y implies x = (SUB1 y) where x and y denote numbers to

(LISTP L), (LENGTH (CDR L)) = 2 and **(REVERSE L) = L.**

From (LISTP L) we can safely rewrite L to be (CONS U V) where U and V are new variables. We substitute (CONS U V) for L in the constraint to obtain a new constraint:

```
(LISTP (CONS U V)),
(LENGTH (CDR (CONS U V))) = 2, and
(REVERSE (CONS U V)) = (CONS U V).
```

This can be further simplified to:

```
(LENGTH V) = 2 and
(APPEND (REVERSE V) (CONS U NIL)) = (CONS U V),
```

using the facts that (LISTP (CONS U V)) is T and (CDR (CONS U V)) = V, and using the definition of REVERSE[4]. Repeating the same process - expanding LENGTH and substituting (CONS Z W) for V - we get:

```
(LENGTH W) = 1 and
(APPEND (APPEND (REVERSE W) (CONS Z NIL))
        (CONS U NIL))
  = (CONS U (CONS Z W)).
```

Another definition expansion gives

```
(LENGTH N) = 0 and
(APPEND (APPEND (APPEND (REVERSE N) (CONS M NIL))
                (CONS Z NIL))
        (CONS U NIL))
  = (CONS U (CONS Z (CONS M N))),
```

[4]We could instead test the known examples of REVERSE and LENGTH again. The example '(B D) of LENGTH satisfies the first condition immediately - V is assigned the value '(B D) - and we have the new constraint (APPEND '(D B) (CONS U NIL)) = (CONS U '(B D)). This can be rewritten to (CONS 'D (CONS 'B (CONS U NIL))) = (CONS U '(B D)), which yields the equation U = 'D. Now L is constructed to be '(D B D) from the equations L = (CONS U V), V = '(B D), and U = 'D and the assignment L = '(D B D) is an example of the original constraint.

with the substitution W = (CONS M N). Next we rewrite the first condition into (NOT (LISTP N)) from the definition of LENGTH and (REVERSE N) into NIL from the definition of REVERSE and the condition that (NOT (LISTP N)). We thus have the new constraint

```
(NOT (LISTP N)) and
(CONS M (CONS Z (CONS U NIL)))
  = (CONS U (CONS Z (CONS M N))).
```

This constraint can be further rewritten into

```
(NOT (LISTP N)),
M = U, and
N = NIL.
```

From this and our previous substitutions L = (CONS U V), V = (CONS Z W), and W = (CONS M N) we can easily construct examples such as

```
L = '(C B C),
L = '(3 5 3), and
L = '((A) D (A)),
```

by appropriately instantiating the free variables.

We have illustrated that we can generate examples for a given constraint 1) by testing known examples and 2) expanding the definitions of involved concepts. The method of testing known examples is usually more efficient, but the method of expanding definitions is more general. It seems necessary to combine these methods appropriately to achieve efficiency and generality of an example generation system.

In the following we explain how examples can be used in the domain of numbers and lists. Gelernter's work clearly suggests how examples can be used in theorem proving even in this domain. Proving theorems about numbers and lists is generally carried out by rewriting formulas into simpler forms. One of the powerful strategies of rewriting formulas is applying axioms and previously proven theorems (lemmas). Suppose we rewrite the term (APPEND X Y) under a certain assumption. This term can be rewritten in different ways by applying the known facts such as:

1. (NOT (LISTP X)) implies (APPEND X Y) = Y;
2. (PLISTP X)[5] and Y = NIL implies (APPEND X Y) = X.

For the first fact to be applied, we need to establish the hypothesis that X is not a list; for the second fact, we need to establish that ; X is PLISTP and Y is NIL. If we find any counter-examples of the hypotheses, then we should not attempt to apply the corresponding fact to rewrite the term because the hypotheses can not be established. In many theorem provers, rewriting formulas by applying lemmas is critical because:

1. As the number of lemmas grows, the proof search space explodes exponentially.
2. To establish a hypothesis it may be required to establish another hypotheses and so on; this backward chaining might run indefinitely long.

Examples can be effectively used to prune unpromising backward chainings during the search for proofs.

Our research has been motivated by Gelernter's work on using diagrams in proving geometry theorems. We have adopted the Boyer-Moore theory as the domain formalism and have implemented an

[5]PLISTP is defined as (PLISTP L) = (IF (LISTP L) (PLISTP (CDR L)) (EQUAL L NIL)).

example generation system called EGS. Experiments have been performed by using the EGS system to control backward chaining in the Boyer-Moore theorem prover. It has been found that irrelevant literals in conjectures, which are often introduced by the induction strategy of the theorem prover, prohibit effective use of examples in pruning unpromising backward chainings.

1.3 Organization of the Thesis

This thesis is organized into nine chapters and three appendices. Chapter 2 sketches the Boyer-Moore theorem prover. Chapter 3 introduces example generation in the Boyer-Moore theory - we define what examples are and define some terms that we will use throughout this thesis. We also illustrate how examples are generated by EGS. In chapter 4 we describe the implementation of EGS and present several cases of EGS-generated examples and in chapter 5 we describe an experiment with EGS for the use of examples in the Boyer-Moore theorem prover and present some statistics and observations. Chapter 6 discusses the weaknesses and limitations of EGS and chapter 7 describes related work on example generation and applications of examples. In chapter 8 we propose some possible improvements to EGS and potential applications of examples in the Boyer-Moore theorem prover. In chapter 9 we summarize our accomplishments and conclude with a remark on examples and Artificial Intelligence. Appendix A lists the definitions of some functions in the Boyer-Moore theory which are referenced in this thesis; appendix B discusses the incompleteness of example generation in the Boyer-Moore theory and the soundness of EGS example generation; appendix C presents some results of our experiment with EGS in the Boyer-Moore theorem prover.

2. The Boyer-Moore Theory and Theorem Prover

We have adopted the Boyer-Moore theory as our domain formalism. The Boyer-Moore theory is a theory of inductively constructed objects and recursive functions; it resembles pure LISP. It is well-defined and is well supported by its mechanized theorem prover. Several advantages should be mentioned:

- The Boyer-Moore theory provides a general and expressively powerful formal language.
- The problems of example representation and evaluation can be easily solved[6].
- EGS can share with the Boyer-Moore theorem prover a substantial portion of its reasoning capability and knowledge base.
- The theorem proving environment serves as a rich source of problems for the application of examples [19, 40, 47, 29, 30].

We assume that the reader is familiar with the Boyer-Moore theory; those who are not familiar to the theory should refer to Boyer and Moore [5, 6]. In this chapter we sketch the Boyer-Moore theorem prover; our example generation system shares several facilities with the theorem prover and our application of examples is closely tied with its structure. We finally make some notes on the Boyer-Moore theory and theorem prover as related to our example generation system and the application of examples.

2.1 The Boyer-Moore Theorem Prover

The Boyer-Moore theorem prover implements the Boyer-Moore logic. The theorem prover includes automated facilities which implement various principles and notational conventions of the Boyer-Moore logic. When a new shell type is introduced, the theorem prover automatically generates axioms which are necessary to axiomatize the shell type. When a new function is defined, it generates some forms of relevant information about the function; they are effectively used during theorem proving. The theorem prover also performs various checks to maintain the consistency of the corresponding theory.

The theorem prover proves a conjecture by successively rewriting it until it reduces to T. The theorem prover employs various heuristics for rewriting formulas. Lemma-based simplification and invention of the induction schema are important features of the theorem prover. When a theorem is proved, it can be stored as a lemma to be used in proving other theorems.

2.1.1 The Shell Mechanism

The shell mechanism is an implementation of the shell principle. It is invoked when new object types (shells) are introduced. The mechanism syntactically generates consistent axioms which axiomatize the new shell. It can be considered as analogous to the data abstraction mechanism in conventional programming languages. For example, suppose we define a new object type "list" by the shell mechanism as follows:

```
add the shell CONS of two arguments
with recognizer LISTP,
     accessors CAR and CDR,
     default values 0 and 0.
```

[6]This will be further discussed in chapter 3.

Some of the important axioms that are automatically generated by the mechanism are the following (with annotations):

(LISTP (CONS X Y))
- *a CONS of two objects is always a list*;

(EQUAL (CAR (CONS X Y)) X)
- *definition of the accessor function CAR*;

(IMPLIES (LISTP X)
(LESSP (COUNT (CAR X)) (COUNT X)))
- *a measure property of accessor functions*;

(EQUAL (EQUAL (CONS X Y) (CONS U V))
(AND (EQUAL X U) (EQUAL Y V)))
- *two CONSes are equal if and only if their components are equal*;

(IMPLIES (LISTP X)
(EQUAL (CONS (CAR X) (CDR X)) X))
- *any LISTP can be represented as a CONS*.

Those axioms are stored in the system's knowledge base for future references for proving theorems.

2.1.2 Defining Functions

When a function is defined, the theorem prover first checks whether the function is total; only total functions are admitted. The totality of a function is guaranteed by establishing an appropriate relation-measure, <r, m> where r is a well-founded relation and m is a function. This relation-measure specifies that at each recursive call of the function in the evaluation with any given actual arguments, m of the new arguments is r-smaller than m of the old arguments. If a function is admitted, the theorem prover generates the corresponding LISP code which implements the function in the LISP environment running the theorem prover. The theorem prover also generates various forms of information about the function being defined and such information is used by components of the theorem prover while proving theorems. Such information includes level number, type prescription, induction machine, etc. and it is also used in example generation. More details will be given in section 4.8.

2.1.3 The Theorem Prover

The Boyer-Moore theorem prover proves that a formula is a theorem by continually rewriting the formula until it is reduced to T. Such rewriting is divided into several different phases: simplification, elimination of destructors, cross-fertilization, generalization, elimination of irrelevance, and induction. Each phase rewrites the input formula. If the formula is rewritten into a (or set of) different formula(s), those formulas are sent back to the very first phase of rewriting, namely, simplification and the process is repeated. If not changed, control goes to the next phase with the formula. Each rewriting phase is explained in more detail below.

2.1.3.1 Simplification

Type Checking: The type checking strategy for simplification makes use of information about the range of values that a term can have. For example, the formula (EQUAL (PLUS X Y) (APPEND U (CONS V W))) simplifies to F because the range of values that the left hand-side term can have is disjoint with that of the right hand-side term - one is NUMBERP while the other is LISTP. The theorem prover

employs various heuristics to generate information about the type of a function value. Such information is created by examining the body of the function definition when a function is defined, and is stored in the knowledge base of that function. For example, from the definition of APPEND, the type information that the term (APPEND X Y) is LISTP or is equal to Y, is generated. Using this information the theorem prover further computes that the type of the term (APPEND U (CONS V W)) is LISTP.

Applying Lemmas: Lemma application is one of the most powerful simplification strategies in the Boyer-Moore theorem prover. It rewrites a term into a simpler term by applying "rewrite" type lemmas. Rewrite lemmas are generally of the form (IMPLIES hyps (EQUAL lhs rhs)). An instance of lhs will be rewritten into the corresponding instance of rhs provided that the corresponding instance of hyps can be established by recursive rewriting. Applying lemmas in this way causes problems such as infinite looping and backward chaining. To avoid some forms of infinite looping the theorem prover adopts an ad hoc strategy that "permutative lemmas" are only applied when the instantiated terms are ordered in a certain way. The backward chaining problem is worth mentioning here in detail because the problem is critical and examples can be used to guide backward chaining in proof attempts. The backward chaining problem arises when the theorem prover tries to establish the hypothesis part of the rewrite lemma in an attempt to simplify the formula using lemmas. Suppose the rewrite lemmas (IMPLIES hyps (EQUAL lhs rhs)) is applied to simplify a formula. To rewrite an instance of lhs into the corresponding instance of rhs it is required to prove that the corresponding instance of hyps is true. Let the instance of hyps be hyps'. In order to prove hyps' the theorem prover may try to apply lemmas and these generate more hypotheses to be established. Those new hypotheses cause yet more hypotheses to be shown true and hence the backward chaining.

Expanding definitions: A substantial part of the power of the Boyer-Moore theorem prover is due to its expansion of definitions when it simplifies formulas. For example, the formula (MEMBER X (CONS U (CONS X L))) simplies to T by opening up the definition of MEMBER and then further applying other simplification methods. The capability of expanding definitions enables the theorem prover to appear very thoughtful. The theorem prover does not limit the depth to which the definition expansion dives. It causes significantly reduced efficiency because the theorem prover puts a great deal of effort in simplifying the much inflated formula. Expanding definitions trades efficiency for power.

2.1.3.2 Eliminating destructors

A plausible strategy is to trade "bad" terms for "good" terms. The theorem prover rewrites a formula by trading some functions for others. For example, the terms (SUB1 X) and X can be traded with Y and (ADD1 Y), respectively when X is known to be a number other than 0. The justification of the elimination of SUB1 in favor of ADD1 is as follows. Suppose we have a formula of the form

```
(p (SUB1 x) x)      *1
```

where x is a variable and the term (SUB1 x) actually occurs in the formula. To prove the formula *1 it is desirable to consider two cases separately:

case 1 x is not a number or 0;

case 2 x is a number and not 0.

For case 2 the formula *1 can be rewritten into

```
(IMPLIES (NUMBERP y)
         (p y (ADD1 y))),
```

trading (SUB1 x) and x for y and (ADD1 y), respectively where y is a new variable.

Another example is the case where a formula involving (QUOTIENT X Y) and (REMAINDER X Y) can be reformulated by trading the terms (QUOTIENT X Y), (REMAINDER X Y), and X for J, I, and (PLUS I (TIMES Y J)), respectively where I and J are new variables. This trading eliminates the function symbols QUOTIENT and REMAINDER in favor of the function symbols PLUS and TIMES. This enables us to induct on J, namely, the number of times Y divides X. Furthermore, the functions PLUS and TIMES are simpler than QUOTIENT and REMAINDER and in fact there are a lot of lemmas about them.

In the Boyer-Moore theorem prover, information that justifies such trading of terms is stored in the form of lemmas labelled as "elimination" type. Some of them are axioms generated by the system when new shells are introduced while some are designated as such by the user after they are proved. Examples of elimination lemmas are

```
Axiom SUB1.ELIM:

   (IMPLIES (AND (NUMBERP X)
                 (NOT (EQUAL X 0)))
            (EQUAL (ADD1 (SUB1 X)) X));

Theorem REMAINDER.QUOTIENT.ELIM:

   (IMPLIES (AND (NOT (ZEROP Y))
                 (NUMBERP X))
            (EQUAL (PLUS (REMAINDER X Y)
                         (TIMES Y (QUOTIENT X Y)))
                   X)).
```

2.1.3.3 Cross Fertilization

The theorem prover uses an equality hypothesis such as (EQUAL s t) by substituting s for some occurrences of t elsewhere in the formula and removing the equality from the formula. This cross fertilization heuristic is closely related to the way that induction arguments are formulated; it is designed to allow use of the induction hypothesis. For further details the reader should refer to Boyer and Moore [5].

For example, consider the theorem

```
(IMPLIES (PLISTP L)
         (EQUAL (REVERSE (REVERSE L)) L)).
```

Simplification of the induction formula yields

```
(IMPLIES (AND (PLISTP L)
              (EQUAL (REVERSE (REVERSE L)) L))
         (EQUAL (REVERSE (APPEND (REVERSE L)
                                 (CONS A NIL)))
                (CONS A L))).
```

Cross-fertilization results in

```
(IMPLIES (PLISTP L)
         (EQUAL (REVERSE (APPEND (REVERSE L)
                                 (CONS A NIL)))
                (CONS A (REVERSE (REVERSE L))))).
```

The induction heuristic fundamentally tries to arrange terms so that in an induction step of the form

(IMPLIES (p t') (p t)), simplification will be able to reduce the conclusion (p t) to some expression containing t', so that the induction hypothesis can be used. Suppose a theorem of the form (EQUAL s t) is to be proved. If the induction heuristic succeeds we will, after simplification, have a conjecture of the form

```
(IMPLIES (EQUAL s' t')
         (EQUAL s (h t'))).
```

In particular, t' will occur in the conclusion because the induction is performed in a way to make it do so.

2.1.3.4 Generalization

In the Boyer-Moore theorem prover generalization is performed by replacing common subterms of the formula with variables. Such replacement often yields a more general conjecture than the original. However, in proofs by induction it is often easier to prove a theorem that is more general than the one which is specific to some particular application. Once the more general theorem is proved, the more specific theorem can be easily proved because then it is a mere instance of the more general theorem. It is also the case that replacing multiple occurrences of the same subterm with a variable causes a significant reduction of the expense of the repeated simplification with those term occurrences.

However, overgeneralization can possibly generate a non-theorem as the new goal and may lead to the failure of the proof. To prevent overgeneralization from occurring or reduce the risk of the generation of a non-theorem, it is necessary to restrict the generalization variable introduced from generalizing the conjecture. Suppose that the common subterm (f x_1 ... x_n) is replaced by a new variable y and (f x_1 ... x_n) is known to be numeric. It might be good to require that y be numeric. For the above REVERSE example,

```
(IMPLIES (PLISTP L)
         (EQUAL (REVERSE (APPEND (REVERSE L)
                                 (CONS A NIL)))
                (CONS A (REVERSE (REVERSE L))))),
```

we can replace the common subterm (REVERSE L) with a new variable B producing

```
(IMPLIES (PLISTP L)
         (EQUAL (REVERSE (APPEND B (CONS A NIL)))
                (CONS A (REVERSE B)))).
```

Even with the usefulness of the generalization strategy of the theorem prover, one must be careful in locating common subterms to be generalized and restricting the generalization variable. In the theorem prover the generalization is guided by lemmas labelled as "generalization" type. Such lemmas are proved by the theorem prover and classified as such by the user. Boyer and Moore discusses various heuristics for generalization.

2.1.3.5 Eliminating Irrelevance

Irrelevant hypotheses are eliminated from the conjecture. This is another way of obtaining a more general conjecture to prove and simplifies the task of finding an appropriate induction. For the REVERSE example, we can eliminate the irrelevant hypothesis (PLISTP L) from the conjecture resulting from generalization producing

```
(EQUAL (REVERSE (APPEND B (CONS A NIL)))
       (CONS A (REVERSE B))).
```

However, it generally requires a deep understanding of the problem at hand to recognize that a

hypothesis is irrelevant to the truth of a conjecture. The Boyer-Moore theorem prover has adopted a few simple heuristics for identifying irrelevant hypotheses. The reader should refer Boyer and Moore [5] for details.

2.1.3.6 Induction

The theorem prover automatically generates induction arguments employing various heuristics. An important fact about the Boyer-Moore logic is that objects are constructed inductively. Its induction principle allows us to prove theorems by inducting on the construction of these objects. This is called "structural induction".

It should be noticed that the induction principle is closely related to the definition principle. Functions in the logic are recursively defined and their termination is established by arguing that a certain measure of the arguments to each recursive call of the function being defined is smaller in some well-founded sense. Eventually the arguments get to the point where no further recursion is possible and the function value starts to build up.

To formulate an induction argument for a conjecture it is important to select appropriate variables to induct on. Selecting induction variables is based on the heuristic evaluation of different cases obtained by considering variables occurring in the conjecture or appropriate combinations of them. The selection is mainly affected by the way that recursion occurs in the definitions of functions involved in the conjecture and the way that variables occur in the conjecture.

Continuing with the REVERSE example, the theorem prover creates an induction argument to prove

```
(EQUAL (REVERSE (APPEND B (CONS A NIL)))
       (CONS A (REVERSE B))).
```

It determines the induction schemes for REVERSE and APPEND; since both functions recurse on the CDR of B, their schemes are merged to create the unique induction scheme

```
(AND (IMPLIES (NOT (LISTP B)) (p B A))
     (IMPLIES (AND (LISTP X) (p (CDR X) Y))
              (p X Y))).
```

According to the scheme the induction argument produced is

```
(AND (IMPLIES (NOT (LISTP B))
              (EQUAL (REVERSE (APPEND B (CONS A NIL)))
                     (CONS A (REVERSE B))))
     (IMPLIES (AND (LISTP B)
                   (EQUAL (REVERSE (APPEND (CDR B)
                                           (CONS A NIL)))
                          (CONS A (REVERSE (CDR B)))))
              (EQUAL (REVERSE (APPEND B (CONS A NIL)))
                     (CONS A (REVERSE B))))).
```

Information used in formulating induction arguments is collected when functions are defined by analysis of their definitions. The "induction machine" is very important. The induction machine represents the way that a recursive function recurses and is used to build induction schemes. It is a table which lists the branches that lead to recursions in the definition, giving the governing tests and the corresponding recursive calls. For example, the induction machine of APPEND is

```
((TEST-AND-CASES ((LISTP X))
                 (((CDR X) Y)))
```

```
            (TEST-AND-CASES ((NOT (LISTP X)))
                            NIL)).
```

It is interpreted that the function APPEND recurses on the CDR of the first argument and the second argument unchanged if the first argument is a list, otherwise no recursion occurs.

Various heuristics are employed for formulating induction schemes, manipulating those schemes and finally producing the induction formula. Boyer and Moore [5] discussed them in detail.

2.2 An Example of Theorem Proving by the Boyer-Moore Theorem Prover

The following is an example of actually running the Boyer-Moore theorem prover - the user input to the prover is underlined. The textual explanation is automatically produced by the theorem prover.

```
(DEFN APPEND (X Y)
     (IF (LISTP X)
         (CONS (CAR X) (APPEND (CDR X) Y))
         Y))

     Linear arithmetic and the lemma CDR-LESSP establish that the
measure (COUNT X) decreases according to the well-founded relation
LESSP in each recursive call.  Hence, APPEND is accepted under the
definitional principle.  Observe that (OR (LISTP (APPEND X Y))
(EQUAL (APPEND X Y) Y)) is a theorem.

[ 0.3 0.0 0.1 ]
APPEND


(DEFN REVERSE (X)
     (IF (LISTP X)
         (APPEND (REVERSE (CDR X)) (CONS (CAR X) NIL))
         NIL))

     Linear arithmetic and the lemma CDR-LESSP establish that the
measure (COUNT X) decreases according to the well-founded relation
LESSP in each recursive call.  Hence, REVERSE is accepted under
the principle of definition.  Note that (OR (LITATOM (REVERSE X))
(LISTP (REVERSE X))) is a theorem.

[ 0.3 0.0 0.1 ]
REVERSE


(DEFN PLISTP (L)
     (IF (LISTP L)
         (PLISTP (CDR L))
         (EQUAL L NIL)))

     Linear arithmetic and the lemma CDR-LESSP establish that the
measure (COUNT L) decreases according to the well-founded relation
LESSP in each recursive call.  Hence, PLISTP is accepted under the
definitional principle.  Note that (OR (FALSEP (PLISTP L))
(TRUEP (PLISTP L))) is a theorem.
```

```
[ 0.3 0.0 0.1 ]
PLISTP


(PROVE '(IMPLIES (PLISTP L)
                 (EQUAL (REVERSE (REVERSE L)) L)))

     Give the conjecture the name *1.


     We will try to prove it by induction.  There are two plausible
inductions.  However, they merge into one likely candidate induction.
We will induct according to the following scheme:
      (AND (IMPLIES (AND (LISTP L) (P (CDR L)))
                    (P L))
           (IMPLIES (NOT (LISTP L)) (P L))).
Linear arithmetic and the lemma CDR-LESSP can be used to prove that
the measure (COUNT L) decreases according to the well-founded
relation LESSP in each induction step of the scheme.  The above
induction scheme leads to three new conjectures:

Case 3. (IMPLIES (AND (LISTP L)
                      (NOT (PLISTP (CDR L)))
                      (PLISTP L))
                 (EQUAL (REVERSE (REVERSE L)) L)),

  which we simplify, unfolding PLISTP, to:

        T.

Case 2. (IMPLIES (AND (LISTP L)
                      (EQUAL (REVERSE (REVERSE (CDR L)))
                             (CDR L))
                      (PLISTP L))
                 (EQUAL (REVERSE (REVERSE L)) L)).

  This simplifies, expanding the functions PLISTP and REVERSE, to:

        (IMPLIES (AND (LISTP L)
                      (EQUAL (REVERSE (REVERSE (CDR L)))
                             (CDR L))
                      (PLISTP (CDR L)))
                 (EQUAL (REVERSE (APPEND (REVERSE (CDR L))
                                         (LIST (CAR L))))
                        L)).

  Appealing to the lemma CAR-CDR-ELIM, replace L by (CONS Z X) to
  eliminate (CDR L) and (CAR L).  The result is:

        (IMPLIES (AND (EQUAL (REVERSE (REVERSE X)) X)
                      (PLISTP X))
                 (EQUAL (REVERSE (APPEND (REVERSE X) (LIST Z)))
                        (CONS Z X))).

  We use the above equality hypothesis by cross-fertilizing
```

```
  (REVERSE (REVERSE X)) for X and throwing away the equality.
  We would thus like to prove:

        (IMPLIES (PLISTP X)
                 (EQUAL (REVERSE (APPEND (REVERSE X) (LIST Z)))
                        (CONS Z (REVERSE (REVERSE X))))).

  We will try to prove the above formula by generalizing it,
  replacing (REVERSE X) by Y.  This produces:

        (IMPLIES (PLISTP X)
                 (EQUAL (REVERSE (APPEND Y (LIST Z)))
                        (CONS Z (REVERSE Y)))),

  which has an irrelevant term in it.  By eliminating the term we
  get:

        (EQUAL (REVERSE (APPEND Y (LIST Z)))
               (CONS Z (REVERSE Y))),

  which we will finally name *1.1.

Case 1. (IMPLIES (AND (NOT (LISTP L)) (PLISTP L))
                 (EQUAL (REVERSE (REVERSE L)) L)),

  which simplifies, opening up the definitions of PLISTP, REVERSE,
  and EQUAL, to:

        T.


     So next consider:

      (EQUAL (REVERSE (APPEND Y (LIST Z)))
             (CONS Z (REVERSE Y))),

which is formula *1.1 above.  We will appeal to induction.  There
are two plausible inductions.  However, they merge into one likely
candidate induction.  We will induct according to the following
scheme:
      (AND (IMPLIES (AND (LISTP Y) (P (CDR Y) Z))
                    (P Y Z))
           (IMPLIES (NOT (LISTP Y)) (P Y Z))).
Linear arithmetic and the lemma CDR-LESSP inform us that the measure
(COUNT Y) decreases according to the well-founded relation LESSP in
each induction step of the scheme.  The above induction scheme
generates two new conjectures:

Case 2. (IMPLIES (AND (LISTP Y)
                      (EQUAL (REVERSE (APPEND (CDR Y) (LIST Z)))
                             (CONS Z (REVERSE (CDR Y)))))
                 (EQUAL (REVERSE (APPEND Y (LIST Z)))
                        (CONS Z (REVERSE Y)))),

  which we simplify, rewriting with CAR-CONS and CDR-CONS, and
  unfolding the definitions of APPEND and REVERSE, to:
```

```
        T.


Case 1. (IMPLIES (NOT (LISTP Y))
                 (EQUAL (REVERSE (APPEND Y (LIST Z)))
                        (CONS Z (REVERSE Y)))).

  This simplifies, rewriting with CAR-CONS and CDR-CONS, and
  expanding the functions APPEND, LISTP, and REVERSE, to:

        T.


     That finishes the proof of *1.1, which, in turn, finishes the
proof of *1.
Q.E.D.
```

2.2.1 A Note on the Introduction of New Variables

The use of examples to control backward chaining is complicated by the introduction of new variables into the conjectures during the proofs. New variables are introduced by destructor elimination and generalization.[7] Since our strategy is that we generate examples once and for all for a given theorem and try to use those examples later on, it is necessary to keep the examples up-to-date. When new variables are introduced during destructor elimination or generalization in the theorem prover, we must generate examples for those variables. But the new variables are introduced by replacing existing subterms. We compute appropriate values for the new variables based on the terms being replaced and the current examples associated with the conjecture. The interface of EGS to the theorem prover carefully keeps track of the introduction of new variables and updates the examples associated with the conjecture appropriately.

2.3 Some Notes on the Boyer-Moore Theory

2.3.1 QUOTEd Constants

In the Boyer-Moore theory certain terms are abbreviated by QUOTE forms. For instance,

```
     (QUOTE 3) abbreviates (ADD1 (ADD1 (ADD1 (ZERO))));

     (QUOTE ABC) abbreviates
         (PACK (CONS (QUOTE 65)
                     (CONS (QUOTE 66)
                           (CONS (QUOTE 67)
                                 (QUOTE 0)))))
         where 65, 66, and 67 are the ASCII code corresponding to
         A, B, and C, respectively;

     (QUOTE (A 1 2)) abbreviates
```

[7] It is also possible that new variables are introduced to the conjecture in simplifying by applying rewrite lemmas because lemmas might have free variables occurring in their conclusion parts. However, such cases are regarded rather as pathological and they are ignored in integrating EGS into the theorem prover.

```
(CONS (QUOTE A)
      (CONS (QUOTE 1)
            (CONS (QUOTE 2) (QUOTE NIL)))).
```

The terms that can be written in QUOTE forms are "explicit values" - variable free expressions composed of shell constructors and bottom objects. Many recursively defined functions are "explicit value preserving"; it is possible to reduce to an explicit value any call on explicit values. (APPEND (QUOTE (A B)) (QUOTE (C))) reduces to (QUOTE (A B C)) and (PLUS (QUOTE 3) (QUOTE 5)) reduces to (QUOTE 8). If a term is variable free and every function symbol occurring in it is explicit value preserving, then the term is "reducible" to a QUOTEd constant. Boyer and Moore have an efficient implementation of reduction that compiles logical functions into LISP[8].

Reduction of terms to QUOTEd constants is important in our example generation. It enables us to determine whether a certain assignments of values (QUOTEd constants) to variables satisfies a constraint formula. For instance, the assignment of the constant (QUOTE (A B C)) to the variable L causes the constraint formula (AND (LISTP L) (EQUAL (LENGTH L) (QUOTE 3))) to reduce to T (more precisely, (QUOTE *1*TRUE)) whereas it causes the formula (EQUAL (REVERSE L) L) to reduce to F (more precisely, (QUOTE *1*FALSE))[9]. Thus the assignment is an example of the first constraint but not an example (but a counter-example) of the second constraint.

The set of all possible QUOTEd constants together with the reduction mechanism corresponds to a formal model of the Boyer-Moore theory. For details the reader should refer to Boyer and Moore [6].

2.3.2 The Constructiveness of the Boyer-Moore Logic

An important characteristic of the Boyer-Moore logic, especially, related to example generation, is its constructiveness. By being constructive, we mean that the existence of an object (or objects) which has certain properties is stated by explicitly describing the way that the object is constructed or found[10]. For example, suppose that one defines the function which computes the remainder of two numbers. In the first-order number theory this can be defined as:

```
rem(x, y) = r if there exist numbers q and r such that
              r < y and x = q*y + r.
```

As we can see it is not clear how one computes the remainder of two given numbers. However, in the Boyer-Moore logic one can define the function as

```
(REMAINDER X Y) =
  (IF (ZEROP Y)
      (FIX X)
      (IF (LESSP X Y)
          (FIX X)
          (REMAINDER (DIFFERENCE X Y) Y))).
```

---

[8]When a new function is defined the theorem prover automatically generates the corresponding LISP code. If a call of the function is evaluated with constant arguments, the LISP code is executed to compute the corresponding value of the function call.

[9]The terms (TRUE) and (FALSE) are abbreviated by their QUOTE forms (QUOTE *1*TRUE) and (QUOTE *1*FALSE), respectively. However, T and F are more often used as the abbreviations of (TRUE) and (FALSE), respectively.

[10]For constructive logic one should refer to Skolem and Goodstein [46, 24]. They presented an idea of using recursion as an alternative to the existential quantification and using induction as a proof procedure in number theory.

The above definition of REMAINDER resembles a (functional) program. This definition actually specifies the way to compute the remainder of any given two numbers.

The constructive nature of the Boyer-Moore logic is due to lack of quantification and the use of recursion as an alternative. In the logic, a logical formula can be interpreted as a computational procedure which evaluates the formula. The constructive nature provides the Boyer-Moore theory with a close resemblance to a programming language and it allows one to build a simple yet efficient interpreter for the Boyer-Moore theory. The interpreter reduces a formula of the theory to the corresponding value with respect to an environment - an assignment of values to variables.

3. Example Generation in the Boyer-Moore Theory

This chapter introduces our system for example generation in the Boyer-Moore theory. We describe how examples are represented and how they are evaluated to see if they satisfy the given constraint. We also illustrate how examples are generated given a constraint formula.

A problem of our example generation is specified in terms of a **constraint**, a well-formed formula in the Boyer-Moore theory[11] . A constraint is generally a conjunction of **conditions** which must be satisfied by examples simultaneously and such conditions interact with each other by sharing variables. Variables occurring in a constraint are called **constraint variables**.

3.1 Representation and Evaluation of Examples

An **example** is defined to be an assignment of QUOTEd constant values to variables - a list of pairs each of which is of the form (var . val) where var denotes a constraint variable and val denotes the assigned constant value in its "QUOTEd" form. The assignment causes the constraint formula to evaluate to some non-F value. A **counter-example** is similarly defined except that the assignment causes the constraint formula to evaluate to F. For instance, the assignment ((X . (QUOTE (A B)))) is an example of (LISTP X), whereas ((X . (QUOTE 3))) is a counter-example. However, the assignment ((X . (CONS U V))) is neither an example nor a counter-example of (LISTP X) since X is assigned a non-constant. Our interpreter would evaluate the formula (LISTP X) to T with the assignment ((X . (QUOTE (A B)))), whereas it would evaluate it to F with the assignment ((X . (QUOTE 3))).

For a more interesting example, the constraint

```
(AND (MEMBER X L)
     (MEMBER Y L)
     (NOT (EQUAL X Y))
     (EQUAL (REVERSE L) L))
```

may be read as:

Find examples of list-palindromes containing at least two different members.

Some examples satisfying the constraint would be

```
((L . (QUOTE (A B A)))
 (X . (QUOTE A))
 (Y . (QUOTE B)))

((L . (QUOTE (C (A) C)))
 (X . (QUOTE (A)))
 (Y . (QUOTE C)))

((L . (QUOTE (1 NIL NIL 1)))
 (X . (QUOTE 1))
 (Y . (QUOTE NIL))).
```

Our definition of examples is first-order in the sense that the variables range over the set of shell

[11] A constraint may not be a well-formed formula because of the abbreviation conventions, however, it can be translated to a well-formed formula.

objects as opposed to functions or sets[12].

An evaluation scheme defines the method which determines whether candidate examples satisfy the given constraint. We adopt interpretation (reduction) as our evaluation scheme[13]. Interpretation reduces a formula[14] to a QUOTEd constant when all variables occurring in the formula are assigned constant values. Boyer and Moore have an efficient implementation of interpretation that compiles logical functions into LISP. With the assignments ((X . (QUOTE (A B)))) and ((X . (QUOTE 3))) the interpreter reduces the formula (LISTP X) to T and F, respectively. Whereas with the assignment ((X . (CONS U V))) it fails to reduce the formula.

3.2 An Overview of Example Generation in EGS

In EGS examples are generated by refinement. The original problem is divided into (or) subproblems and each of those subproblems is further refined to produce new subproblems. When no further refinement is applicable to a subproblem, we construct examples from information collected in the subproblem. The refinement and example construction process continues until the sufficient number of examples have been generated or the allocated resources such as time have been exhausted.

In EGS a subproblem is internally represented in the form of a **context** which consists of a **hard-list** and **soft-list**. The hard-list contains the constraint in the form of a list of conditions assumed conjoined; the soft-list contains the **substitution equalities** represented in the form (lhs . rhs), where lhs is a variable and rhs is a term which does not contain lhs as a variable. In this paper we use the terms *constraint*, *context*, and *subproblem constraint* interchangeably when there is no danger of confusion. For the list-palindrome example the original problem is represented as the context:

```
hard-list: ((MEMBER X L)
            (MEMBER Y L)
            (NOT (EQUAL X Y))
            (EQUAL (REVERSE L) L))

soft-list: NIL.
```

An intermediate subproblem that might be generated during example generation can be in the form:

```
hard-list: ((MEMBER X (QUOTE (A B A)))
            (MEMBER Y (QUOTE (A B A)))
            (NOT (EQUAL X Y)))

soft-list: ((L . (QUOTE (A B A)))).
```

Example generation in EGS can be viewed as a sequence of transformations from the constraint (hard-list) to a list of substitution equalities (soft-list). When the hard-list becomes empty, an example (or examples) is constructed directly from the information collected in the soft-list.

For a given subproblem represented by a context several operations can be applied to further refine

---

[12]Ballantyne and Bledsoe have discussed higher order example generation in functional analysis and topology [2, 4].

[13]In his theorem prover, Aubin has employed simplification in evaluating "structures" assigned to variables for checking the non-provability of generalized conjectures [1]. A similar approach has been adopted by Cohen in LTP [12].

[14]A formula must be reducible. For detail refer to Boyer and Moore [6].

the subproblem. A refinement operation associated with a subproblem is called a **task**; performing a task means the actual execution of the attached operation with the corresponding context. Generally, performing a task produces multiple subproblems derived from the context representing the current problem. Each of them is more specific and easier to handle than its parent problem. New tasks are created for these new subproblems. Each time a task is created, its plausibility score is computed. The plausibility score measures the "worth" of the task for generating examples. A task may also contain information that is useful in performing the task. For example, the task

```
task score: 84
task operator: TEST
hard-list: ((MEMBER X L)
            (MEMBER Y L)
            (NOT (EQUAL X Y))
            (EQUAL (REVERSE L) L))
soft-list: NIL
info-list: (CLUE (REVERSE L) FLG NIL)
```

represents a task which tests (the typical and boundary) examples of REVERSE against the constraint (hard-list). Performing a task results in contexts representing new subproblems derived from the problem corresponding to the context of the task. Each of these new contexts is simplified and apparently unsatisfiable contexts are eliminated from further consideration. Suppose that as the result of performing a task we have a context representing a subproblem created whose hard-list is empty. With the context only the CONSTRUCT operator attached to it to produce a CONSTRUCT task. Then examples are actually generated by performing the CONSTRUCT task. Each of the generated examples is finally checked to see if it actually satisfies the original constraint. The following is an example of the CONSTRUCT task:

```
task score: 1000
task operator: CONSTRUCT
hard-list: NIL
soft-list: ((L . (QUOTE (A B A)))
            (X . (QUOTE A))
            (Y . (QUOTE B)))
info-list: (TEMPLET (L X Y)).
```

All tasks generated are maintained in the data structure called the **task agenda**. The task agenda keeps the tasks in the order of the plausibility score; the task on the top the task agenda is considered the currently most plausible one.

3.3 A Simple Illustration of How EGS Generates Examples

In this section we will illustrate how EGS generates examples. For convenience we will explain the illustration informally without strictly following our representational convention. Let x <- t denote an assignment of the term t to variable x. We concentrate on those paths of problem solving which lead to successful generation of examples, largely ignoring many other possibilities.

Suppose we have the constraint stated by the following well-formed formula:

```
constraint: (AND (MEMBER X L)
                 (MEMBER Y L)
                 (NOT (EQUAL X Y))
                 (EQUAL (REVERSE L) L))
```

EGS translates the constraint formula into the constraint form; a list of conditions each of which needs to be satisfied. The original problem would be represented in the form of the context:

```
hard-list: ((MEMBER X L)
            (MEMBER Y L)
            (NOT (EQUAL X Y))
            (EQUAL (REVERSE L) L))
soft-list: NIL.
```

EGS first tests known examples against the constraint. Terms occurring in the constraint provide clues about examples to be tested. For example, the term (REVERSE L) suggests that the stored examples under the function symbol REVERSE could be retrieved and tested. Suppose that the stored examples of REVERSE are:

```
Typical examples[15]:
  ((QUOTE (A B C))), ((QUOTE (A B A))), ((QUOTE (C B . A))),
  ((QUOTE (H G G H))), ((QUOTE (E F))),
  ((QUOTE (D (D (D))))), ((QUOTE (C))),
  ((QUOTE (5 3 2))), ((QUOTE (3 5 3))), ((QUOTE (3 3))),
  ((QUOTE (3 7 . 3))), ((QUOTE (1 2 3 4)));

Boundary examples:
  ((QUOTE A)), ((QUOTE NIL)), ((QUOTE (A . A))), ((QUOTE 0)).
```

The typical examples and the boundary examples of REVERSE are tested against the constraint. For each of the examples the testing is performed by substituting the variables in the constraint with the corresponding values and evaluating those conditions having no variables in the formula.

At the same time the terms (MEMBER X L), (MEMBER Y L), and (EQUAL X Y) can also give clues for examples to test. Selection of a clue to process is heuristically determined. Suppose the term (REVERSE L) is chosen and the known example ((QUOTE (A B C))) of REVERSE is being tested by instantiating the variable L with (QUOTE (A B C)) in the constraint. It fails to satisfy the condition (EQUAL (REVERSE L) L), therefore it is discarded. This example testing process continues until EGS has found some number[16] of examples which satisfy the constraint or all the retrieved known examples have been tested. In this case, we have two typical examples ((QUOTE (A B A))) and ((QUOTE (H G G H))) and a boundary example ((QUOTE NIL)) that have passed the test because they do not explicitly falsify any of the conditions in the constraint. For each of those successful examples, new subproblem constraints are generated:

```
case L <- (QUOTE (A B A)):
  hard-list: ((MEMBER X (QUOTE (A B A)))
              (MEMBER Y (QUOTE (A B A)))
              (NOT (EQUAL X Y)))
  soft-list: ((L . (QUOTE (A B A)))),

case L <- (QUOTE (H G G H)):
  hard-list: ((MEMBER X (QUOTE (H G G H)))
              (MEMBER Y (QUOTE (H G G H)))
              (NOT (EQUAL X Y)))
```

[15]The stored example is in the form of an ordered list of constant values whose i-th element corresponds to the i-th argument of the function. Each constant in an example is internally represented unQUOTEd.

[16]The number is set by an EGS system parameter.

```
    soft-list: ((L . (QUOTE (H G G H)))),
```
and

```
  case L <- (QUOTE NIL):
    hard-list: ((MEMBER X (QUOTE NIL))
                (MEMBER Y (QUOTE NIL))
                (NOT (EQUAL X Y)))
    soft-list: ((L . (QUOTE NIL))).
```

However, the last case will be simplified to be false; the first two cases can be further processed. For the first case, the condition (MEMBER X (QUOTE (A B A))) is processed and split into two cases (EQUAL X (QUOTE A)) and (EQUAL X (QUOTE B))[17]. For these two cases we have the new constraints:

```
  case L <- (QUOTE (A B A)), X <- (QUOTE A):
    hard-list: ((MEMBER Y (QUOTE (A B A)))
                (NOT (EQUAL (QUOTE A) Y)))
    soft-list: ((X . (QUOTE A))
                (L . (QUOTE (A B A))))
```
and

```
  case L <- (QUOTE A B A), X <- (QUOTE B):
    hard-list: ((MEMBER Y (QUOTE (A B A)))
                (NOT (EQUAL (QUOTE B) Y)))
    soft-list: ((X . (QUOTE B))
                (L . (QUOTE (A B A)))).
```

Similarly, for both cases, the first condition of each constraint is selected and processed to produce the following cases:

```
  case L <- (QUOTE (A B A)), X <- (QUOTE A), Y <- (QUOTE B):
    hard-list: NIL
    soft-list: ((Y . (QUOTE B))
                (X . (QUOTE A))
                (L . (QUOTE (A B A))))
```
and

```
  case L <- (QUOTE (A B A)), X <- (QUOTE B), Y <- (QUOTE A):
    hard-list: NIL
    soft-list: ((Y . (QUOTE A))
                (X . (QUOTE B))
                (L . (QUOTE (A B A)))),
```
respectively.

For both cases no constraints are left to be processed in the hard-lists and the value assignments in the soft-lists become examples.

So far we have considered the cases derived from successful testing of the REVERSE examples. Now, a question may be raised:

> What if no known examples have passed the testing?
> For instance, what if REVERSE does not have examples like
> ((QUOTE (A B A))) and ((QUOTE (H G G H))) as stored examples?

In such cases EGS would appeal to a more fundamental method, namely, definition expansion. With the original conjecture, the definition of REVERSE is expanded and simplification yields:

```
  case:
    hard-list: ((LISTP L)
```

[17]This case spilt is performed by one of the solvers associated with MEMBER and this will explained in chapter 4.

```
                (MEMBER X L)
                (MEMBER Y L)
                (NOT (EQUAL X Y))
                (EQUAL (APPEND (REVERSE (CDR L))
                               (CONS (CAR L) NIL))
                       L))
  soft-list: NIL.
```

For this case again the testing would fail. However, the condition (LISTP L) suggests that L can be replaced by (CONS U V) where U and V are new variables not occurring in the constraint. The replacement will produce:

```
case L <- (CONS U V):
  hard-list: ((MEMBER Y V)
              (NOT (EQUAL U Y))
              (EQUAL (APPEND (REVERSE V) (CONS U NIL))
                     (CONS U V)))
  soft-list: ((X . U)
              (L . (CONS U V))),

case L <- (CONS U V):
  hard-list: ((MEMBER X V)
              (NOT (EQUAL X U))
              (EQUAL (APPEND (REVERSE V) (CONS X NIL))
                     (CONS U V)))
  soft-list: ((Y . U)
              (L . (CONS U V))), and

case L <- (CONS U V):
  hard-list: ((NOT (EQUAL X U))
              (NOT (EQUAL Y U))
              (MEMBER X V)
              (MEMBER Y V)
              (NOT (EQUAL X Y))
              (EQUAL (APPEND (REVERSE V) (CONS X NIL))
                     (CONS U V)))
  soft-list: ((L . (CONS U V))).
```

Suppose we are to process the first case. Testing a stored example of MEMBER - suggested by the condition (MEMBER Y V), for instance, ((QUOTE A) (QUOTE (A B))), and simplifying the constraint produces a new subproblem:

```
case L <- (CONS U V), Y <- (QUOTE A), V <- (QUOTE (A B)):
  hard-list: ((NOT (EQUAL U (QUOTE A)))
              (EQUAL (CONS (QUOTE B)
                           (CONS (QUOTE A) (CONS U NIL)))
                     (CONS U (QUOTE (A B)))))
  soft-list: ((V . (QUOTE (A B)))
              (Y . (QUOTE A))
              (X . U)
              (L . (CONS U V))).
```

The above subproblem is further simplified to generate an example:

```
case L <- (CONS U V), Y <- (QUOTE A), V <- (QUOTE (A B)):
  hard-list: NIL
  soft-list: ((U . (QUOTE B))
              (V . (QUOTE (A B)))
```

```
                (Y . (QUOTE A))
                (X . U)
                (L . (CONS U V))),
```

which produces the example:

```
          ((L . (QUOTE (B A B)))
           (X . (QUOTE B))
           (Y . (QUOTE A))).
```

Suppose again we do not have such appropriate stored examples of MEMBER like ((QUOTE A) (QUOTE (A B))). In such cases the repeated expansion of the definition of REVERSE and simplification will produce several subproblems and one of them would be:

```
case L <- (CONS U V), V <- (CONS Z W), W <- (CONS M N):
  hard-list: ((NOT (LISTP N))
              (NOT (EQUAL U Z))
              (EQUAL (APPEND
                       (APPEND
                         (APPEND (REVERSE N) (CONS M NIL))
                         (CONS Z NIL))
                       (CONS U NIL))
                     (CONS U (CONS Z (CONS M N)))))
  soft-list: ((W . (CONS M N))
              (Y . Z)
              (V . (CONS Z W))
              (X . U)
              (L . (CONS U V))),
```

which will be further simplified, using the definitions of REVERSE and APPEND and the fact that (EQUAL (CONS X Y) (CONS U V)) implies that (EQUAL X U) and (EQUAL Y V), to:

```
case L <- (CONS U V), V <- (CONS Z W), W <- (CONS M N):
  hard-list: ((NOT (EQUAL U Z)))
  soft-list: ((N . NIL)
              (M . U)
              (W . (CONS M N))
              (Y . Z)
              (V . (CONS Z W))
              (X . U)
              (L . (CONS U V))),
```

which, in turn, causes examples to be easily created from the known examples of EQUAL and CONS. Some of the generated examples would be:

```
          ((L . (QUOTE (C (E) C)))
           (X . (QUOTE C))
           (Y . (QUOTE (E))))
```

and

```
          ((L . (QUOTE (3 7 3)))
           (X . (QUOTE 7))
           (Y . (QUOTE 3))).
```

4. EGS: Implementation

EGS was originally written in INTERLISP running on a DEC-2060 under TOPS-20 and later translated into ZETALISP. It runs on top of the Boyer-Moore theorem prover on a Symbolics LISP machine. In this chapter we describe the implementation of EGS in detail. We begin with the description of the agenda-centered architecture of EGS and then explain task generation and task performing, the major part of EGS. Finally, we list several cases of constraints together with their EGS-generated examples.

4.1 The Architecture of EGS

EGS is composed of six major functional components: Executive, Preprocessor, Task Generator, Task Performer, Simplifier, and Evaluator. The global data structure Task Agenda maintains tasks in order and the Knowledge Base contains various forms of knowledge that are used in generating tasks and performing tasks. Also, EGS has Linear Solver as a subcomponent of the task performer. Figure 4-1 depicts the EGS architecture.

Figure 4-1: EGS Architecture

Executive: The executive component controls the other components of EGS in generating examples. It implements the control structure of EGS as shown in Figure 4-2. The control is mainly iterating the sequence of task generation and task performing. The executive also initializes the system status, managing resources such as time, and checking whether the current system status satisfies the

termination conditions.

Preprocessor: The preprocessor implements some aspects of the user-interface of EGS. This component also performs high-level analysis such as resource allocation - how much time is assigned and how many examples for a specific constraint are to be generated? Currently, the preprocessor simply translates the user's abbreviated input constraint into a logically well-formed formula. For example, the user input formula (AND (NUMBERP X) (LESSP X 8) (EQUAL X (CADR L))) translates into (AND (NUMBERP X) (AND (LESSP X (QUOTE 8)) (EQUAL X (CAR (CDR L))))).

Task Generator: The task generator generates tasks from the current subproblem constraint (produced by performing a task) by associating with it an appropriate task operator and information that is relevant to performing the task. Currently, seven task operators are available: START, TEST, SOLVE, ANALYZE, EXPAND, CONSTRUCT, and RECALL. Each operator is associated with a "specialist" for the corresponding task. The specialists checks certain conditions required to generate the corresponding type of tasks and collect the relevant information from the knowledge base. For a given subproblem constraint, multiple different types of tasks can be generated and several tasks of the same type with different relevant information can be generated. Each task is assigned a score which measures its plausibility.

Task Performer: This component performs a task by invoking the task performing specialist corresponding to the task operator. Performing a task involves retrieving relevant information from the knowledge base and applying the specified task operator to the constraint of the task. The application of a task operator generally produces multiple refined subproblem constraints, each of which is subject to further processing to generate new tasks.

Simplifier: The simplifier simplifies subproblem constraints produced from performing a task; it rewrites a constraint to be simpler and easier to handle. The simplification is global in the sense that when a constraint is simplified each of the conditions in the constraint is rewritten by taking into consideration all the other conditions in the constraint. The simplification is based on rewrite rules which are axioms and lemmas.

Evaluator: This component evaluates whether or how well the proposed examples satisfy the initial constraint. The function of evaluation would be important for the system to be able to learn or self-organize from experience. Currently it simply determines whether candidate examples actually satisfy the initial constraint.

Task Agenda: The task agenda is a global data structure which maintains the generated tasks. Each generated task is scored with its plausibility; the task agenda keeps the tasks in the order of plausibility.

Knowledge Base: The knowledge base contains various forms of information: function definitions, lemmas, known examples, procedures for rewriting special forms of formulas, etc. Such information is used by the various components of EGS.

4.2 Control Structure of EGS

The essence of the control structure of EGS is the iteration of the sequence of task generation and task performing. A problem of example generation is specified in the form of a user input constraint. It is translated into the corresponding well-formed formula by the preprocessor. For the constraint formula, tasks are generated and stored in the task agenda. The task on the top of the task agenda is performed. Performing a task and the subsequent global simplification produces new subproblem constraints. For each of the new constraints, the task generator generates new tasks and the new tasks are scored and stored in the task agenda. The task on the top of the task agenda will be performed next, and so on. The alternation of task generation and task performing continues until some termination condition is satisfied. Examples are actually generated when a CONSTRUCT task is performed. A CONSTRUCT task has an empty hard-list; thus examples can be directly constructed from information in the soft-list. Figure 4-2 shows the control structure of EGS.

Figure 4-2: EGS Control Structure

4.3 Task Generation

A task, in general, specifies an application of a refinement scheme to a subproblem constraint. Some types of tasks, such as START and RECALL, do not correspond to any refinement; they simply serve

EGS to control generating and performing other tasks. A task is described by a **task descriptor**[18] and the descriptor consists of five entities: **task score**, **task operator**, **hard-list**, **soft-list**, and **info-list**. Each entity carries some information relevant to performing the task: task score contains the plausibility score of the task; task operator specifies which refinement scheme (or control scheme) is applied; hard-list contains the unresolved part of constraint; soft-list contains a collection of substitution equalities; info-list contains information that is used in performing the task. Each task contains all the necessary information for generating examples; a task can be regarded as a top-level goal in its own right. Given a subproblem constraint, the task generator invokes the task generation specialists to generate tasks of the constraint. In the following we will describe each specialist for task generation.

4.3.1 START Task

This task is generated for initiating the iteration of the task generation and task performing sequence. The START task is actually generated by the preprocessor of EGS and put into the task agenda. The START task is generated only once for a given example generation session.

4.3.2 TEST Task

This is the most important type of task. The corresponding refinement scheme of the task is to (partially) instantiate variables with constants obtained from stored examples. For example, suppose we have the following constraint:

```
((PRIME P)
 (DIVIDES P X)
 (MEMBER X (CDR L))).
```

A TEST task can be generated which would retrieve the stored examples of DIVIDES and instantiate the argument variables P and X with the corresponding components of an individual example. Suppose we have the stored example ((QUOTE 3) (QUOTE 12)) of DIVIDES. Now the variable P is assigned the constant (QUOTE 3) and X is assigned (QUOTE 12). With this assignment each condition formula is evaluated to true (non-F) except the last one, (MEMBER (QUOTE 3) (CDR L)), and the resulting constraint will be further processed.

To generate a TEST task it is necessary to compute a pattern, which would key to retrieving stored examples. Such a pattern is called a **test clue**. It must be related to the scheme in which known examples are classified and stored. In the current implementation a simple strategy has been adopted; a function symbol keys to the stored examples. A test clue is a subterm in the constraint of the form (f t_1 ... t_n), where f is a function symbol called the **primary function symbol**. The primary function symbol f is used as a key to retrieving stored examples. For instance, a condition (MEMBER X (APPEND U V)) in the constraint may have a test clue (APPEND U V). This clue suggests retrieving the stored (typical and boundary) examples of APPEND and instantiating the variables U and V with them. Test clues are generated by extracting appropriate subterms in the constraint. For a given constraint there could be many test clues generated. Test clues may widely differ in their plausibility. For examples, the clue (PRIME X) is considered more plausible than (NUMBERP X) and (MEMBER X L) is more plausible than (MEMBER X (APPEND U V)). Generated test clues are evaluated in their plausibility. Plausibility scores

[18]Later on we use "task" instead of "task descriptor" when there is no danger of confusion.

for test clues are taken into consideration when the overall task scores are computed. The following are important factors in evaluating test clues.

- The level number[19] of the primary function symbol of the clue. The level number of a function symbol roughly indicates the complexity of the function;
- The coverage which measures the extent to which the clue variables - variables being assigned constants when an example is instantiated - covers the constraint. For a given formula, it is the ratio of the number of the different variables occurring in the test clue over the number of the different variables occurring in the constraint;
- The complexity of the clue. This measures the difficulty of the equations corresponding to mapping each component of an example to the corresponding argument of the clue. For instance, the test clue (MEMBER X (CDR L)) is considered more complex than the test clue (MEMBER X L). When we map an example of MEMBER to these test clues, the corresponding equations are more difficult to solve for the first clue than the second clue.

The resulting test clues are further processed so that the similar clues are merged into one and negligible clues are discarded. Furthermore, the test clues whose plausibility scores are quite low compared to those of the rest or to the average score are discarded. Humans seem to consider even trivial clues when few clues are available while ignoring unplausible clues when many plausible clues are available. By behaving analogously EGS is able to avoid exploring cases which would be likely to result in examples with only minor differences. Computing appropriate test clues is important to avoid biased example generation.

For each of the resulting test clues a TEST task is generated. The info-list of the task contains the corresponding test clue and a flag indicating whether the typical and boundary stored examples are to be retrieved or the counter examples are to be retrieved. For example, the constraint

```
((PRIME P)
 (NOT (DIVIDES P X))
 (MEMBER X (CDR L)))
```

produces the following set of test clues:

```
{(PRIME P) (NOT (DIVIDES P X)) (MEMBER X (CDR L))}.
```

For the first two clues we will have the following tasks generated.

```
task score: 72
task operator: TEST
hard-list: ((PRIME P)
            (NOT (DIVIDES P X))
            (MEMBER X (CDR L)))
soft-list: NIL
info-list: (CLUE (PRIME P) FLG NIL)
```

and

```
task score: 78
task operator: TEST
hard-list: ((PRIME P)
            (NOT (DIVIDES P X))
            (MEMBER X (CDR L)))
soft-list: NIL
```

[19]See section 4.8.2.4.

info-list: (CLUE (DIVIDES P X) FLG T).

4.3.3 SOLVE Task

SOLVE is a refinement scheme which corresponds to generalized equation solving. Suppose the following condition formula appears in a constraint

(EQUAL (REVERSE L) '(C B A)).

Computing L for the condition would be costly if we were to use a general method of repeated expansion of the definition of REVERSE and simplification. It would be more efficient to directly solve the equation by applying some procedural knowledge which computes an answer. In this case, we would have (EQUAL L '(A B C)) as an answer. In solving equations generally there could be more than one solution. In such cases it is necessary that the procedure for solving equation should be coded in a way to select a unique answer from among many possibilities. However, equation solving may result in so specialized an answer that the answer may fail to satisfy some of other conditions in the same constraint. For our example, the answer (EQUAL L '(A B C)) does not satisfy the condition (NOT (PLISTP L)), which might be in the constraint. In such cases one can employ a general solver which produces a general solution. Even with such limitations, however, it seems that the SOLVE transformation scheme as a heuristic for solving equations is desirable.

The equation solving scheme can be more generalized so that more than one equation can be solved simultaneously; currently, solving of linear equations/inequalities has been implemented as one form of general equation solving scheme. For solving equations, it is required that the user provide EGS with solvers, production rule-like knowledge for solving equations. Equations are generally condition formulas in a constraint in the form of (EQUAL lhs rhs). A predicate term of the form (p t_1 ... t_n) can be interpreted as an equation (EQUAL (p t_1 ... t_n) T). The following will describe the generation of two different types of SOLVE tasks: linear SOLVE task and simple SOLVE task.

4.3.3.1 Linear SOLVE Task

A solver for linear arithmetic equations and inequalities, called "linear solver", has been built into EGS. The linear solver simultaneously solves a set of linear equations and inequalities whose unknowns are variables. Each condition formula corresponding to a linear arithmetic equation/inequality is translated into a polynomial equation/inequality, which is internally represented as a triple (rel alist const). Here rel, alist, and const represent the polynomial relation, the list of pairs of unknown and coefficient for a term in the polynomial, and the constant term of the polynomial, respectively. Each inequality is normalized to a less-than-equal-to (LEQ) relation by adjusting the constant term and transferring unknown terms around appropriately. For example, the condition formula (LESSP (PLUS X 4) (TIMES 3 Y)) would be internally represented as (LEQ ((X . 1) (Y . -3)) -5).

For generating a linear SOLVE task the task generator scans the current constraint and translates linear arithmetic equations/inequalities[20] into their internal representations and collects them. To distinguish a linear solving task from other solving task we assign the value "linear" to the MODE attribute in the info-list of a SOLVE task. The info-list also contains the list of polynomial forms of

[20](NOT (EQUAL lhs rhs)) is not translated because in example generation formulas of that form are most unrestrictive and often can be ignored. Also, the translation assumes that all unknown variables range over the natural numbers.

equations/inequalities. For example, the constraint

```
((NUMBERP Z)
 (LESSP X (PLUS Y (QUOTE 8)))
 (NOT (LESSP X (TIMES 3 Y)))
 (NOT (EQUAL Y Z)))
```

would yield the linear SOLVE task:

```
task score: 82
task operator: SOLVE
hard-list: ((NUMBERP Z)
            (LESSP X (PLUS Y (QUOTE 8)))
            (NOT (LESSP X (TIMES 3 Y)))
            (NOT (EQUAL Y Z)))
soft-list: NIL
info-list: (MODE LINEAR
            POLY-LST
            ((LEQ ((X . 1) (Y . -1)) 7)
             (LEQ ((X . -1) (Y . 3)) 0))).
```

4.3.3.2 Other SOLVE Tasks

Other SOLVE tasks are generally concerned with solving a single condition formula. Each condition formula in a constraint may be considered an equation of the form (EQUAL lhs rhs). A condition of the form (p t_1 t_2 ... t_n) where p is a predicate is equivalent to (EQUAL (p t_1 t_2 ... t_n) T). SOLVE tasks often solve such equations with respect to the principle functions of lhs[21].

The task generator computes an appropriate primary function for each condition formula of the constraint and checks whether the function symbol has any associated solver. If there are associated solvers, then for each of those solvers a SOLVE task is generated. The info-list of the SOLVE task would include information such as the name of the solver and the TUPLE information which is a list of rhs followed by the arguments of lhs in the equation.

The constraint

```
((MEMBER X L)
 (EQUAL (REVERSE L) '(C B A))),
```

for example, would yield the SOLVE task

```
task score: 81
task operator: SOLVE
hard-list: ((MEMBER X L)
            (EQUAL (REVERSE L) '(C B A)))
soft-list: NIL
info-list: (SOLVERS (F-COND-1 REVERSE-SOL-1)
            TUPLE ((QUOTE (C B A)) L)).
```

[21]Very often rhs is constant; however, this is not necessarily the case.

4.3.4 ANALYZE Task

The ANALYZE refinement scheme analyzes a certain term in the constraint by cases. It refines the constraint by adding to the current constraint new conditions suggested by subterms in the constraint. Case analysis can be carried out when a function definition is expanded. The ANALYZE refinement scheme generally does case analysis on terms of function symbols having no definitions, such as shell primitives. The ANALYZE scheme can also apply to cases where expanding functions is not desirable because it causes many uninteresting subgoal splits. For such cases the user might suggest subcases which he wants to consider separately. Such knowledge is encoded in the form of a procedure called an analyzer by the user and stored in the knowledge base. For example, consider the constraint

```
((NUMBERP (CAR X))
 (NOT (NUMBERP (CDR X)))).
```

In this case one might want to replace X with (CONS U V) where U and V are new variables. The subterms (CAR X) and (CDR X) suggest that two cases for X, namely, (NOT (LISTP X)) and (LISTP X), can be considered separately. The first case yields an unsatisfiable subgoal but the second yields a much simpler subgoal constraint

```
((NUMBERP U)
 (NOT (NUMBERP V))),
```

with X replaced by (CONS U V).

Currently, the ANALYZE scheme applies only to shell accessors such as CAR and CDR. Case-analyze clues are computed and sorted by variables occurring in them. For a given term a case-analyze clue is a list of the form (v f_1 f_2 ... f_n) where v denotes a variable, the f_i's denote accessor functions which have case analysis knowledge associated, and the term (f_1 v) occurs in the term. If more than one case-analyze clues are generated for a given constraint, similar ones will be merged. Finally, for each of the resulting clues an ANALYZE task is generated. For the above constraint a case analyze clue would be (X CAR CDR) and the following task would be generated.

```
task score: 63
task operator: ANALYZE
hard-list: ((NUMBERP (CAR X))
            (NOT (NUMBERP (CDR X))))
soft-list: NIL
info-list: (CLUE (X CAR CDR)).
```

4.3.5 EXPAND Task

The EXPAND refinement scheme expands the symbolic definition of a function in the current constraint. An EXPAND task is generated only after a RECALL task is performed. We only apply the EXPAND scheme to certain subproblem constraints - those produced by performing RECALL tasks. This is done because, lacking an effective method for detecting unsatisfiable or unpromising constraints, for certain subproblem constraints, especially unsatisfiable constraints, expanding definitions may run forever.[22]. To avoid this danger we apply the EXPAND scheme to general forms of constraints rather than to specific ones and such general forms of constraints are retained by RECALL tasks. No particular

[22]For instance, consider the subproblem constraint (hard-list) ((EQUAL (APPEND X '(B)) '(C B A))). This constraint is apparently unsatisfiable. Attempts to solve the constraint by expanding the definition of APPEND may not terminate. In such cases we may want to apply the EXPAND scheme to the original constraint.

information needs to be saved in the info-list of an EXPAND task.

4.3.6 CONSTRUCT Task

A CONSTRUCT task is generated when the hard-list of a subproblem constraint is empty. The info-list contains the TEMPLATE information, a list of variables occurring in the original constraint. This scheme constructs a specific example from information in the soft-list. Examples must be value assignments to the variables in TEMPLATE; other variables created during example generation will be ignored. Usually CONSTRUCT tasks are given the highest plausibility score. EXPAND tasks are generally given a low plausibility score.

4.3.7 RECALL Task

RECALL tasks are generated after an EXPAND task is performed. Performing an EXPAND task followed by simplification splits the current constraint into subproblem constraints. For each of the subproblem constraints a RECALL task is generated. A RECALL task simply saves a subproblem constraint so that it can be retrieved and processed as needed. For RECALL tasks, no particular information needs to be saved in the info-list. They are usually given low plausibility score.

4.4 Plausibility Score Computation

Computing the plausibility score is important; it actually determines the pattern of search in example generation. For our example generation the search space is vast because for any given constraint there are many (maybe an infinite number of) different possibilities to consider. Also, there are generally many potential solutions (examples); we need to select some good examples from among many possibilities. We have designed the score computation mechanism to be flexible enough to allow the user to control the plausibility score computation instead of adhering to a certain fixed algorithm. Such controllability allows us to keep EGS in as general a form as possible so that it can be used in various different applications.

In computing a plausibility score of a task three important factors are considered. They are task operation, complexity of the current constraint formula, and extra credit. We will explain each of these in more detail.

Task Operation: Operations of various tasks correspond to different refinement schemes and are given different weights. These weights indicate the relative importance of task operations. For example, the CONSTRUCT operation is given the highest weight and the EXPAND operation is given a much lower score. One may want a higher weight for the TEST operation than the EXPAND operation. Since it is generally the case that the CONSTRUCT tasks must always be given the highest score, we need to assign a weight to the CONSTRUCT operation such that no other tasks are assigned higher plausibility scores than those of the CONSTRUCT tasks.

The Complexity of the Constraint: For each task the hard-list specifies the current constraint. The syntactic complexity of the constraint formula is measured and is used in computing the plausibility score. The syntactic complexity indicates the restrictiveness of the current constraint, measured by considering the level of difficulty of involved functions, the appearance of variables and constants in the constraint, and the occurrence of special terms in the constraint. The difficulty of a function is approximated by its

"level number", which is computed by the theorem prover when the function is defined For example, the level number of PLUS is 1 whereas the level number of TIMES is 2. The appearance of variables and constants is important to determining the syntactic complexity of the constraint. Multiple occurrences of the same variable symbol within a condition or over conditions in the constraint is considered restricting; it is more difficult to find examples satisfying such a constraint. More frequent occurrences of constants in the constraint is considered more restrictive than the case of less frequent occurrences of constants. Some special patterns of formulas are so important that they need to be treated separately. For example, an equality of the form (EQUAL lhs rhs)[23] is generally considered quite restrictive, whereas its negation i.e. (NOT (EQUAL lhs rhs)) is considered far less restrictive. The quantitative measures of these aspects of the syntax of the formula are combined to yield the overall syntactic complexity of the current constraint.

Extra Credit: Extra credit for a task is determined by considering the availability and plausibility of relevant information to performing the task. Such information is stored in the info-list of the task. For examples, a TEST task with a more plausible test clue may get more extra credit than one with a less plausible test clue; the plausibility of test clues is largely determined by syntax. For SOLVE tasks those having a "quick" solver can be given more credit than those having a "general" solver. Extra credit primarily allows us to order those tasks with the same task operation and the same constraint formula but different relevant information.

In the above we have described three important factors which determine the plausibility score of a task. The quantities representing each of the factors are combined to produce the overall plausibility score of a task. We adopt a simple formula to compute the overall score. Let op-score, s-score, and credit denote the operation, constraint complexity, and extra credit scores of a task, respectively. Then the overall plausibility score will be given by the formula:

$$\textbf{Plausibility Score} = C_1 * \textbf{op-score} + C_2 * \textbf{s-score} + C_3 * \textbf{credit}$$

where C_1, C_2, and C_3 denote the corresponding weight coefficients.

The algorithm for computing plausibility score can be arbitrary. Many decisions are made heuristically and largely based on the syntactic form of the relevant information. It is questionable, however, whether the plausibility of tasks can be well represented in such a formula. Humans seems to employ a much more complicated mechanism to evaluate the plausibility of tasks. This mechanism is largely based on a thorough understanding of the domain and a good deal of problem solving experience with the domain. Our algorithm is relatively simple and seems to approximate reasonably well the plausibility of tasks.

4.5 Task Agenda

The task agenda is the global data structure of EGS which maintains the generated tasks in the order of plausibility. When tasks are generated, they are scored in plausibility and stored in the task agenda in the order of the plausibility score. Each time a task is performed, the most plausible task (the task on the top of the task-agenda) is selected to be performed. The example generation problem in EGS is viewed as search. The original problem is divided into subproblems and those subproblems again produce other

[23]The substitution equality is not included here because the substitution equality is always transferred from the hard-list to the soft-list when the subproblem constraint is simplified.

subproblems. At any point of search, there are many alternatives to explore. The agenda scheme together with the plausibility score mechanism, as a way of implementing a search method, is suitable for a problem which requires a vast search space in which search is to be carried out in a complicated way.

Search for example generation in EGS is difficult; the important factors that cause the difficulties:

- the system is required to be efficient;
- the search space for example generation is vast and the solutions are many: we need to find good ones from among them;
- the example generation problem (at least of EGS) is undecidable.

It is often the case that an example generation system is used as a component of a larger system to increase the efficiency of the larger system. In such cases, it is important that the example generator be efficient enough that it does not detract from the performance of the larger system. For a given example generation problem, there are many different ways to generate examples. However, very often it is not desirable to enumerate all these possible examples. We need to generate examples selectively. We are most concerned with generating good examples, but the characterization of good examples is rather vague and may differ among applications. Furthermore, it is difficult to evaluate intermediate states of example generation in terms of how close they are to the goal state. For an example generation system it is desirable that the search carried out by the system be controllable by a simple mechanism and the user be able to define or modify the mechanism. The agenda scheme together with the plausibility score mechanism provides EGS with considerable controllability to search in EGS.

4.6 Task Performing

Performing a task is, in general, applying the corresponding refinement scheme to the current constraint (the hard-list of the task). As the result, the current constraint is refined into (multiple) subproblem constraints, each of which is simplified and is further processed to create new tasks. Some types of tasks such as START tasks and RECALL tasks do not correspond to any refinement schemes but are simply for controlling the task generation and task performing process of EGS. After a task is performed, the task performer tells the task generator specifically which types of tasks are allowed to be generated from the result of performing the current task. It gives us more freedom in controlling the task generation and task performing of EGS. For instance, performing a RECALL task only allows EXPAND tasks to be generated and after an EXPAND task is performed RECALL tasks are generated for each of the subproblem constraints produced by performing the EXPAND task. This allows the EXPAND scheme to be applied only to those subproblem constraints which have been produced by performing an EXPAND task.

4.6.1 START Task

This task initiates the iteration of the sequence of task generation and task performing. Performing a START task simply returns the current constraint. After a START task is performed a RECALL task is generated with the current constraint and the RECALL task allows EGS to go back to the original constraint if needed.

4.6.2 TEST Task

Performing a TEST task retrieves appropriate stored examples and tests each of them against the corresponding constraint of the task. The info-list of the task contains information such as the test clue and what type of examples should be retrieved to be tested. The types of examples to be retrieved are divided into two cases: 1) typical and boundary examples and 2) counter examples. The corresponding types of examples of the primary function symbol of the test clue are retrieved. For instance, the TEST task:

```
task score: 84
task operator: TEST
hard-list: ((MEMBER X L)
            (MEMBER Y L)
            (NOT (EQUAL X Y))
            (EQUAL (REVERSE L) L))
soft-list: NIL
info-list: (CLUE (REVERSE L) FLG NIL)
```

is interpreted to mean that the typical and boundary examples of REVERSE should be retrieved from the knowledge base and tested. The value of FLG (in this case NIL) indicates which type of examples to be retrieved.

When an example is tested, we need to instantiate the arguments of the test clue with the corresponding components of the example. Suppose that the test clue term is of the form (f t_1 ... t_n) where f is a function symbol of arity n. We have several cases to consider; the following three cases are important and need to be mentioned.

1. When all t_i's are variables and no two of them are the same[24] each variable t_i is bound to the i-th component of the example.
2. When all t_i's are variables and some of them are the same we randomly select one from among those components of the example corresponding to the arguments of the same variable and bind the variable to the selected value.
3. When some t_i's are non-variable terms we instantiate the non-variable term to the corresponding component of the example after simple checking that the value is compatible with the term. In the case that the same terms occurs more than one place in the arguments of the test clue, we do as in the case 2. If the instantiated non-variable term is in a simple form involving functions such as ADD1 or SUB1, then the corresponding equation is solved with respect to the variable occurring in the term.

If some of t_i's are constants, then the corresponding components of the example are ignored while the instantiations of the other arguments are made as usual. The binding of arguments (a variable or a term) to the corresponding values creates an environment for evaluation, called a "test environment". When the instantiation is completed, the constraint of the task is evaluated in the test environment. A modified version of the Boyer-Moore interpreter is invoked to evaluate the constraint. Only the conditions of the constraint containing the bound variables are evaluated. If any of them evaluates to F, then the example being tested is discarded and the next example is tested. This testing process continues until some number of examples[25] have passed the testing. When both of the typical and the boundary examples are

[24]This means that two variables are syntactically the same.

[25]The actual number is set by a system parameter; the number differs by the type of examples. To some extent this number controls the even-distributedness of the examples generated by EGS

tested at the same time, each type is tested separately and the specified number of the successful typical examples and the boundary examples are computed.

For each of the successful examples, we generate a new subproblem constraint as following:

1. delete from the hard-list those conditions which have evaluated to non-F;
2. for each of the non-variable terms t in the test environment and its corresponding value val add to the hard-list new conditions of the form (EQUAL t val);
3. for each of the variables v in the test environment and its corresponding value val substitute the value for all occurrence of the variable and add to the soft-list the pair of the form (v val).

In performing the above TEST task, the stored REVERSE examples ((QUOTE (A B A))), ((QUOTE (A B B A))), and ((QUOTE NIL)) pass the testing and these cases produce the following new subproblem constraints, respectively.

```
hard-list: ((MEMBER X (QUOTE (A B A)))
            (MEMBER Y (QUOTE (A B A)))
            (NOT (EQUAL X Y)))
soft-list: ((L (QUOTE (A B A)))),

hard-list: ((MEMBER X (QUOTE (A B B A)))
            (MEMBER Y (QUOTE (A B B A)))
            (NOT (EQUAL X Y)))
soft-list: ((L (QUOTE (A B B A)))), and

hard-list: ((MEMBER X (QUOTE NIL))
            (MEMBER Y (QUOTE NIL))
            (NOT (EQUAL X Y)))
soft-list: ((L (QUOTE NIL))).
```

4.6.3 SOLVE Task

There are two different cases to consider in performing a SOLVE task: 1) solving by application of solvers and 2) solving linear inequalities.

4.6.3.1 Applying Solver

A solver is user-given knowledge about some procedure to transform a particular formula into a simpler form. The solver transforms a formula of the form (EQUAL lhs rhs)[26] into a formula or formulas which are easier to handle. The solver is in production rule-like form: a pair of IF-part and THEN-part. For example, one of the solvers of MEMBER could be:

```
IF-part:
   rhs is T,
   lhs is of the form (MEMBER x l) where l is a constant of list
   and x is a variable;

THEN-part:
   for each different element e in l produce a new context by
   replacing the condition lhs in the current constraint with
   (EQUAL x e).
```

[26]The predicate formula of the form (p t_1 ... t_n) is also considered as an equation whose lhs is the predicate form and rhs is "T" while for (NOT (p t_1 ... t_n)) rhs is "F".

The application of this solver is equivalent to expanding the definition of MEMBER and simplifying the resulting formula. However, in many cases applying solver knowledge is more efficient. Details on solver is described in section 4.8.2.6.

4.6.3.2 Solving Linear Equations/Inequalities

If the SOLVE task indicates that the task is to solve linear inequalities, the info-list of the task contains the polynomial representation of the linear arithmetic conditions in the constraint. Consider the previous linear solving task example:

```
task score: 82
task operator: SOLVE
hard-list: ((LESSP X (PLUS Y (QUOTE 8)))
            (NOT (LESSP X (TIMES 3 Y)))
            (NOT (EQUAL X Y)))
soft-list: NIL
info-list: (MODE LINEAR
            POLY-LST
            ((LEQ ((X . 1) (Y . -1)) 7)
             (LEQ ((X . -1) (Y . 3)) 0))).
```

The polynomial representation can be rewritten in the arithmetical form:

$$X - Y \leq 7 \quad ..... (1)$$
$$-X + 3Y \leq 0 \quad ..... (2)$$

These simultaneous inequalities are simplified by the method of variable elimination [3, 8]. For example, (1) + (2) gives us $2Y \leq 7$ which is further simplified, considering the fact that Y is a natural number, to yield $0 \leq Y \leq 3$. We also have $0 \leq X \leq 10$, by substituting the value range for Y. We choose the most restrictive variable and assign it appropriate numbers within the value range. The numbers of the range are scanned in random order and the value assignment is tested with the constraint. New contexts are generated for only the first specified number of successful cases.

The linear inequality solver employs various techniques to simplify inequalities and reduce the value range of variables with the assumption that variables range over natural numbers. For example, suppose we have an inequality

$$3X + 5Y \leq 10.$$

From this we can restrict the value range of X and Y to $0 \leq X \leq 3$ and $0 \leq Y \leq 2$, respectively. The inequality $3X + 6Y \leq 14$ can be rewritten into the equivalent form $X + 2Y \leq 4$, given that we are restricted in the domain of natural numbers. The linear solver parameterizes certain variables occurring in linear equations to restrict their value ranges. For example, suppose we have the equation

$$3X - 5Y = 10.$$

The variables X and Y are parameterized as $X = 5P + 5$ and $Y = 3P + 1$ with $0 \leq P$[27]. The variables X

[27]Parameterization applies only to an equation of the form $Ax + By = C$ where x and y are variables, A, B, and C are integers, and A and B are mutually prime. If A and B are mutually prime, we can always find integers u and v such that $Au + Bv = 1$ (Euclid's theorem). From this we can have $ACu + BCv = C$. Using the equality we can parameterize the variables x and y as follows: $x = Bt + Cu$ and $y = -At + Cv$ where t is a parameter ranging over integers. This parameterization satisfies the original equation. For our example, we may have the parameterization $X = 5p + 20$ and $Y = 3p + 10$ where p is a parameter ranging over integers. However, we need to adjust p so that X and Y range over natural numbers. From $X = 5p + 20$ we have $-4 \leq p$; from $Y = 3p + 10$ we have $-3 \leq p$. Now we have $-3 \leq p$ satisfying the two conditions simultaneously. We further set $p = P - 3$ ($0 \leq P$) and we substitute the equation in the parameterization to obtain $X = 5P + 5$ and $Y = 3P + 1$ where $0 \leq P$.

and Y occurring in other equations/inequalities are substituted by the parameterized forms. Parameterization increase the power of the linear solver.

4.6.4 ANALYZE Task

The ANALYZE task performs case analysis with the current constraint based on the information collected from certain types of subterms occurring in the constraint. Among such information, the case analysis clue is important. Even though we can further generalize this refinement scheme, currently we simply do case analysis on variables. The case analysis clue is of the form (v f_1 ... f_n) where v is a variable symbol and the f_i's are function symbols which suggest the case analysis. When the ANALYZE task is performed, each of the case analyzers of the f_i's is executed. The case analyzer produces the conditions to be added to the corresponding constraint of the task. For each f_i such conditions are generated and are added to the current constraint to obtain new and more refined constraints. For our example of the ANALYZE task

```
task score: 63
task operator: ANALYZE
hard-list: ((NUMBERP (CAR X))
            (NOT (NUMBERP (CDR X))))
soft-list: NIL
info-list: (CLUE (X CAR CDR)),
```

we retrieve the case analyzers[28] of CAR and CDR (in this case they are the same) and apply them to the associated variable to produce the conditions (LISTP X)[29] and (NOT (LISTP X)). After performing the task and simplification we would have the new contexts generated:

```
hard-list: ((NUMBERP U))
            (NOT (NUMBERP V))))
soft-list: ((X . (CONS U V))).
```

For the second case of (NOT (LISTP X)) the constraint is simplified to F because, in the Boyer-Moore theory, if (NOT (LISTP X)) is true, then (CAR X) and (CDR X) are equal to 0.

4.6.5 EXPAND Task

This task expands the definition of a function in the current constraint of the task. Definition expansion followed by simplification corresponds to the case analysis of the constraint with respect to the function being expanded. Which function to expand is determined heuristically so that the expansion of the definition results in an appropriate refinement of the current constraint. Subterms of the form (f t_1 ... t_n) in the constraint, where f is a defined function symbol, are evaluated in their plausibility for definition expansion. Heuristic information used in the evaluation includes the level number of f, the complexity of the subterm, and the occurrence of t_i's in other subterms. The last is considered important because, if cases are split on t_i's as the result of the definition expansion, then other terms with the same t_i's as arguments would also affected by the case split. Once a term has been determined to expand, every occurrence of the term in the constraint will be replaced with the equivalent instantiation of the definition of the primary function. Only RECALL tasks are generated after an EXPAND task is performed; this

[28]For an example see section 4.8.2.7.

[29]For this case the case analyzer actually generates the condition (EQUAL X (CONS U V)), where U and V are the new variables, to be more efficient.

allows EGS to "recall" the subproblem constraints produced by performing the EXPAND task as needed.

4.6.6 CONSTRUCT Task

A CONSTRUCT task constructs examples from the information in the soft-list. A CONSTRUCT task is generated only when the hard-list of the current context is empty; by that time the soft-list contains sufficient information for constructing examples. The soft-list is a list of pairs of the form (v t) where v and t represent a variable and a term, respectively. This pair is interpreted to mean that the variable denoted by v is equal to the term denoted by t. It should be noticed that for each such pair, v does not occur in t. From the way that the soft-list is built we also know that the soft-list is consistent - no variables occur more than once in the left hand side of pairs in the soft-list. This fact is important in reconstructing examples from the soft-list. This provides the DAG (Directed Acyclic Graph) property of the order of the variable instantiation for example construction. Suppose that we have the soft-list of a CONSTRUCT task

```
((L . (CONS U V)) (U . (CONS X (QUOTE (C))))
 (V . (APPEND X Y))).
```

The following DAG graph represents the order of the variable instantiation.

Figure 4-3: DAG for Variable Instantiation

The graph can be interpreted to mean that we need to instantiate the variable X before U, X and Y before U and V, and the values of U and V need to be determined before L.

When variables are instantiated with values, arbitrary values can be assigned to the tip node variables in the graph. However, EGS does the instantiation in a smarter way. Suppose we instantiate the tip node variables X and Y. To do that we would refer to the term (APPEND X Y), choose an appropriate example from among the stored examples of APPEND, and assign the corresponding components of the example to the variables. In this way EGS can generate more natural and "user-tuned" examples. For various special cases, such as when the term has constant values in some of argument positions, we instantiate in a way similar to "testing stored examples". When the variable being instantiated occurs in more than one term or when more than one function symbol suggest the instantiation, we instantiate the variable with one example from each of those supporting functions to obtain multiple instantiations.

When a "supporting" function has no stored examples, the CONSTRUCT task performer tries to acquire support from those major functions occurring in the symbolic definition body of the function. In this case it also tries to match an arbitrary example chosen from the pool of "free examples" for the variable being instantiated.

4.6.7 RECALL Task

The role of the RECALL task is simply to retain certain subproblem constraints; usually those produced from performing an EXPAND task. Performing a RECALL task simply produces the same subproblem constraint as the constraint of the RECALL task. An EXPAND task, together with other types of tasks, is generated after a RECALL task is performed.

4.7 Simplification

It is important to keep the constraint formula in as simple a form as possible. Since examples are generated by successively transforming constraint formulas into the forms of substitution equalities, a great deal of formula manipulation, especially simplification, needs to be carried out. A constraint is simplified after it is produced. The major functions of the simplifier are:

1. to transfer sequentially each substitution equality in the hard-list onto soft-list, after carrying out the indicated substitution on the formulas of the hard-list;
2. to rewrite condition formulas on the hard-list to simpler forms.

Consider the subproblem constraint

```
hard-list: ((EQUAL L (CONS U V))
            (NUMBERP (CAR L))
            (MEMBER Y (CDR L)))
soft-list: NIL.
```

This would be simplified to

```
hard-list: ((NUMBERP U)
            (MEMBER Y V))
soft-list: ((L . (CONS U V))).
```

The following are some advantages of simplification.

1. By rewriting formulas into simpler and normalized forms, the simplifier allows the task generator to be able to easily handle constraints to generate tasks.
2. By eliminating apparently unsatisfiable subgoals, the simplifier enables EGS to focus on more promising subgoals.
3. The fact that the simplifier makes use of rewrite lemmas provides EGS with some degree of extensibility.

Even with these advantages, it is necessary to employ simplification carefully because it is generally expensive. Simplification in example generation differs from that in theorem proving because:

1. it is not necessary that formulas be fully simplified;
2. it is desirable to avoid too many case splits generated by simplification.

Example generation in EGS has a good chance of success by testing known examples and testing known examples is cheap. We postpone the application of more expensive methods until they are needed. Also, in our example generation it is necessary to avoid the generation of biased examples such as examples with minor differences or many examples generated by considering only a few subcases.

The EGS simplifier shares a substantial portion of its implementation with the simplifier of the Boyer-Moore theorem prover; it behaves very much like the Boyer-Moore simplifier, however, it rewrites formulas in a limited way. The EGS simplifier rewrites formulas using type information, lemmas and

function definitions. Important differences between the EGS simplifier and the Boyer-Moore simplifier are as follows:

1. EGS simplifies formulas with **shallow definition expansion**. It only allows expanding definition up to a limited number of levels; currently, it expands definitions only one level deep. In EGS it is not necessary to dive deep into definitions to simplify formulas because eventually EGS will appeal to the strategy of expanding definitions in case other strategies fail to generate examples. The shallow definition expansion in the EGS simplification tends to be efficient, particularly when the formulas being simplified involve many high level functions[30].
2. The EGS simplifier avoids unnecessary case splits: when it expands the definition of a function, the simplifier checks the governing conditions corresponding to the cases to split from the definition expansion to see if any of the conditions is satisfied. Only when there is a governing condition that is (explicitly) satisfied, the simplifier attempts to expand the definition of the function.[31].
3. The simplification in EGS is single-pass in the sense that it does not iterate until no further simplification is possible.

4.8 Knowledge Base

Knowledge in its various forms plays an important role in EGS example generation. Most of the knowledge is used in transforming constraints into (hopefully) simpler and easier to handle forms. Such knowledge includes stored examples, symbolic function definitions, procedures which solve equations and perform case analysis, axioms and proved theorems, and various kinds of information about functions used in making heuristic decisions.

Each refinement scheme corresponding to a task operation is simply an application of a particular type of knowledge to the current constraint in order to obtain more refined subproblem constraints. The simplifier, a global transformation scheme, employs various heuristic information and axioms and theorems already proved in rewriting formulas.

Knowledge in EGS is divided into two groups: the knowledge which is specific to EGS and knowledge which it shares with the Boyer-Moore theorem prover. The shared knowledge is originally created by the Boyer-Moore theorem prover, whereas the EGS specific knowledge is given by the user and only devoted to example generation by EGS.

It is necessary for EGS to provide a rich knowledge base and a mechanism which supports easy manipulation of such knowledge. This section will explain the kinds of knowledge that are employed in the EGS example generation, how such knowledge is structured, acquired, and used.

---

[30]For example, suppose the formula (PRIME X) is simplified. The Boyer-Moore simplifier would dive into the definitions of PRIME, PRIME1, REMAINDER, and down to DIFFERENCE. The shallow simplification of EGS only dives one level into the definition of PRIME.

[31]For instance, expanding the definition of MEMBER in (MEMBER A B) would yield three cases to split: 1) if (NOT (LISTP B)), (MEMBER A B) can be rewritten into F; 2) if (LISTP B) and (EQUAL A (CAR B)), (MEMBER A B) can be rewritten into T; 3) if (LISTP B) and (NOT (EQUAL A (CAR B))), (MEMBER A B) can be rewritten into (MEMBER A (CDR B)). Consider the formula (MEMBER X (CONS U (CONS V (CONS W NIL)))) is simplified by expanding the definition of MEMBER. None of the governing conditions corresponding to the above three cases are not satisfied, therefore the definition expansion is abandoned. In this case the Boyer-Moore simplifier would rewrite it into three cases: (EQUAL X Y), (EQUAL X U), or (EQUAL X V).

4.8.1 Knowledge Structure

Most of the shared knowledge available to EGS is automatically created by the theorem prover at the time a new function is defined. Such knowledge includes symbolic definitions, LISP code, the induction machine, type prescriptions, controller pockets, level number, etc. Axioms are also automatically generated and stored by the theorem prover when a new object type (shell type) is introduced to the underlying theory. Lemmas are previously proved theorems formulated by the user and proved by the theorem prover. The user also specifies how each lemma is to be used. The EGS specific knowledge is mostly given by the user. It includes stored examples, equation solving procedures, case analysis procedures, and information used in controlling the opening up the function definitions. Knowledge is sorted by functions (concepts) and the function symbol keys to the associated knowledge. The knowledge associated with a function is represented in frame-like structure - a collection of slot and value pairs. Figure 4-4 illustrates the EGS knowledge structure for the concept MEMBER.

4.8.2 Types of Knowledge

4.8.2.1 Symbolic Definition

Each defined function has a unique symbolic definition. Each time a function is defined the definition is carefully analyzed to justify that the function always terminates; the Boyer-Moore logic only admits total functions according to the principle of definition. For each admitted function, the translated version of the definition is stored in the knowledge base. The symbolic definition is important in the Boyer-Moore theorem prover because various other information is also derived from the symbolic definition.

The symbolic definition is used in three different ways in EGS:

1. when an EXPAND task is formed - the occurrence of a term in the constraint is replaced with the corresponding instantiation of the symbolic definition of the primary function of the term;
2. when EGS simplifies formulas - given a term the EGS simplifier (recursively) dives into the definitions of the involved functions and attempt to simplify the term by simplifying the appropriate instances of the function definitions in a limited (shallow) way;
3. when EGS generates the examples of the function being defined - EGS generates the examples of the defined functions by referring the symbolic definition. The details are explained in a later section.

4.8.2.2 LISP Code

When a new function is defined, the Boyer-Moore system automatically generates the corresponding LISP code, which implements the defined function in the running LISP environment. The code is invoked when a term is evaluated. The LISP code is heavily used when the TEST tasks are performed and the proposed examples are finally evaluated by the evaluator component of EGS. For some primitive functions such as EQUAL and IF the LISP code has been hand-coded. It may also be the case that some LISP code is hand-coded for efficiency, such as the PRIME function. The details of the LISP code is described in [6].

```
TYPE-PRESCRIPTION-LST ((MEMBER 3 NIL NIL))

LEMMAS
   (REWRITE-RULE MEMBER-DELETE
                 ((MEMBER X (DELETE U V)))
                 (MEMBER X V)
                 NIL)
   (REWRITE-RULE MEMBER-APPEND NIL
                 (EQUAL (MEMBER X (APPEND A B))
                        (IF (MEMBER X A) T (MEMBER X B)))
                 NIL)

   ;;; More lemmas could be shown here.
   ;;; Lemmas are shown in the reverse order.

SDEFN
   (LAMBDA (X LST)
     (IF (LISTP LST)
         (IF (EQUAL X (CAR LST))
             T
             (MEMBER X (CDR LST)))
         F))

LISP-CODE *1*MEMBER

INDUCTION-MACHINE
   (TESTS-AND-CASES ((NLISTP LST)) NIL)
   (TESTS-AND-CASES ((NOT (NLISTP LST))
                     (EQUAL X (CAR LST)))
                    NIL)
   (TESTS-AND-CASES ((NOT (NLISTP LST))
                     (NOT (EQUAL X (CAR LST))))
                    ((X (CDR LST)))))
LEVEL-NO 1

CONTROLLER-POCKETS (2)

EXAMPLES
   (TYPICAL ((A) (A (A))) (A (A B)) (2 (5 3 2))
            (3 (3 5)) (B (A B C)))
   (BOUNDARY (NIL (NIL)) (A (A)) (A (A . B)))
   (COUNTER (B (A (B) ((C)))) (1 (3 2 . 1)) (A NIL)
            ((A) (A)) (A B) (D (A B C)) (A (B C))
            (1 (2 3)) (2 (1 . 2))))

SOLVERS (QUICK (IF PSOL-COND1 THEN MEMBER-SOL1)
               (IF MEMBER-COND2 THEN MEMBER-SOL2)))))
```

Figure 4-4: The Knowledge Structure for MEMBER

4.8.2.3 Induction Machine

The induction machine for a function is a table-like structure of entries each of which consists of recursive calls of the function and conditions governing those calls in the body of its definition.[32] It is used when the theorem prover formulates induction schemes in an attempt to prove theorems. For example, the induction machine for MEMBER is

```
            conditions                  calls

1      (NOT (LISTP L))                    -

2        (LISTP L) &
       (EQUAL X (CAR L))                  -

3        (LISTP L) &
    (NOT (EQUAL X (CAR L)))     (MEMBER X (CDR L))
```

EGS uses this information when it simplifies a term. During the simplification it may be necessary to unfold a function definition because doing so results in a term that is simpler or easier for EGS to handle. Before the simplifier actually unfolds a definition, it evaluates whether the result would be better. The simplifier checks if any tests of the function's induction machine can be fully satisfied and, if any, it further checks if the corresponding recursive calls can be rewritten to be simpler. For example, the constraint,

```
((NOT (LISTP L))
 (MEMBER X (CONS Y L))
 (NOT (EQUAL X Y)))
```

is simplified to F. When the second condition (MEMBER X (CONS Y L)) is being simplified under the assumption of the first and the last ones, the simplifier examines MEMBER's induction machine instead of working through the definition body. The third entry of the induction machine justifies MEMBER being expanded because the tests are fully established and the corresponding recursive call gets simpler. However, for the constraint,

```
((NUMBERP X) (MEMBER X L)),
```

when the simplifier works with (MEMBER X L) in order to find out whether it is desirable to open up the definition of MEMBER, it simply checks with the first entry of the induction machine of MEMBER, whose tests are not explicitly satisfied and stops there concluding that opening up the definition is not desirable. The reason that it stops even on the first entry is that no following entries can have the explicit establishment of their test parts. Using the induction machine this way when diving into the definition results in a substantial improvement in the efficiency of the simplifier.

4.8.2.4 Level Number

The level number roughly represents how difficult the function is. The level number of a function is determined as follows. The level numbers of some primitive functions and the shell functions (shell constructor, recognizer, and accessors) are 0. For a recursively defined function its level number is one plus the highest level of the functions occurring in its definition, except the function being defined. The level number of a non-recursive function is equal to the highest level of those functions occurring in its definition. For example, the level numbers of PLUS and APPEND are 1, those of TIMES and REVERSE are 2 while PRIME is of the level number 3. The level number of a function provides some very useful

[32] Induction machine also includes those cases leading to no recursion.

heuristic information for computing the plausibility score of tasks and selecting terms to expand for the EXPAND task. The information of type prescription and controller pocket is also used in simplification. For details the reader should refer to Boyer and Moore [5].

4.8.2.5 Stored Examples

Three different types of known examples - typical, boundary, and counter examples - are associated with each defined function[33]. These known examples of a function are classified by the user and stored as indexed by the function name in the knowledge base. There are no stringent rules for classifying such examples into typical examples and boundary examples; the criteria could be arbitrary. However, we classify them based on our understanding about the function involved and our rather vague notion of being typical and being exceptional. It seems that this criterion is obtained from experience. Generally, boundary examples correspond to the base case of a function definition or require only one recursive call of the function when the function is evaluated with those examples. Typical examples seem to require more than one recursive call in the evaluation. The reason for this classification into typical and boundary examples is that EGS is required to be general enough to be used for a variety of applications. If we try to illustrate a certain concept (function), we may want to give some typical examples of the concept which clearly demonstrate the important characteristics of the concept and also would like to show some unusual cases. Biased examples do not show the various aspects of the concept being demonstrated and often cause misunderstanding. In this case we need to generate examples in such a way that both types of examples appear evenly distributed or in a certain ratio, for instance one boundary example for three generated examples. Suppose we have a certain conjecture to be disproved. It is often the case that such a conjecture is valid for the more usual cases; the counter-examples often occur in unusual cases. In this case boundary examples are suited for constructing counter examples. We control the distribution of generated examples, to some extent, at the time stored examples are tested. We select to a certain ratio typical examples and boundary examples that have passed the test. Without such classification, it is difficult and inappropriate to judge mechanically which examples are typical or unusual with respect to a constraint during example generation.

The definition of an example of a function is slightly different from our usual notion of an example of a constraint. An example of a function f of arity n is an n-tuple of constants each component of which corresponds to a formal argument of the function definition. Typical and boundary examples cause the corresponding function to evaluate to non-F, while counter examples causes the function to evaluate to F. Typical examples are generally those which exhibit well the properties of the function, whereas boundary examples are special and extreme ones that rarely exhibit the important properties of the function. Currently, counter examples are only legitimate for Boolean functions.

Each time a TEST task is performed, the appropriate types of the stored examples of a function are retrieved to be tested against the current constraint. Only some number of examples which have passed the test are selected and processed to obtain the refinements of the current constraint.

Stored examples play the role of building blocks for example generation in EGS. The examples generated by EGS are often simply stored examples or combinations of them. Stored examples provide:

---

[33]Currently, for each function, on the average, 8 typical examples, 5 boundary examples, and 8 counter-examples in case the function is Boolean are given.

- a substantial increase in the efficiency of EGS example generation;
- for generated examples being more natural and user-tuned;
- for potential machine learning capability of EGS.

4.8.2.6 Solver

The solving refinement scheme models an important aspect of human example generation: applying highly proceduralized problem-specific knowledge to transform formulas into ones which are more refined and easier to handle. Such a procedure for transforming formulas is called a "solver" and requires deep understanding of the involved concepts, an extended body of the related knowledge, and powerful reasoning capability. For solver knowledge, EGS relies on the user.

The general form of a solver should be able to arbitrarily manipulate formulas in a constraint. Currently only a limited form of solving scheme has been implemented; an application of a solver yields a list of subgoal constraints.

Generally solver knowledge can recognize an arbitrary particular pattern of formulas in a constraint and transform them into some more desirable form. For example, the linear equations/inequalities solver which has been built in EGS recognizes linear arithmetic equations and inequalities in the constraint and converts them into the internal representations. It then processes them, attempting to rewrite the original equations and inequalities into the more refined and easier form. For efficiency reasons, the linear solver has a considerable amount of built-in knowledge about natural numbers, linear operations such as addition, constant multiplications, integer division, and parameterization.

While the linear solver manipulates formulas globally in the constraint, most solvers carry out local formula transformations in the sense that the extent of formula manipulation is limited. Usually a single equation formula in a constraint is processed at a time. Application of a solver transforms a formula of the form (EQUAL (f t_1 ... t_n) rhs) - predicate formulas are also interpreted as equations - in the constraint into new constraints. Solver knowledge is like production rules; it consists of two parts, namely, the IF (premise) part and the THEN (action) part. It is interpreted as "if the IF part is true then do the THEN part". The IF part checks if the formula being solved is in a form appropriate to apply the THEN part. The THEN part produces new constraints representing new refined subproblems. Also when a TEST task is performed, the IF part of a solver is often referenced in an attempt to find out if a certain term is solvable by assuming that a value is assigned to it. One of the solvers of REMAINDER which solves an equation of the form (EQUAL (REMAINDER x y) z) is:

```
IF-part:
   z is a number and
   x is a number and
   x is not less than z;

THEN-part:
   delete the equation from the constraint and
   if x is equal to z
      then
        create a new constraint by adding to the resulting
        constraint a new condition of the form  (ZEROP y),
      else
        for each of the divisors d of the difference (x - z)
        that are less than z, create a new constraint by
```

adding to the resulting constraint a new condition of the form (EQUAL y d).

The application of this solver with the equation (EQUAL (REMAINDER 12 Y) 4) will effectively rewrite it into (EQUAL Y 8).

Both parts of a solver are coded currently in LISP by the user. It may be desirable to define a language in which one can easily code such procedural knowledge. It is not necessarily the case that the solver knowledge is sound: the solver knowledge can be heuristic and specific to a particular application. In many cases of solving equations there exist multiple solutions. However one may not want to consider all solution cases for some reason, for example, because it will be split into too many cases and eventually cause the biased distribution of generated examples. In such cases, one can define a solver so that it handles the situation appropriately.

There are two different types of solvers: "quick" solvers and "general" solvers. The quick solver quickly solves an equation to produce a few common solutions, while the general solver tries to obtain more general solutions. For example, consider the solvers of REVERSE which solve the equation (EQUAL (REVERSE X) (QUOTE (A B C))). The quick solver might quickly generate the case that (EQUAL X (QUOTE (C B A))). However, the general solver may generate the case of the conjunction of (EQUAL X (APPEND (QUOTE (C B A)) W)) and (NOT (LISTP W)), where W is a new variable. Generally quick solvers are given higher plausibility than general ones.

A function can be associated with many solvers. When a SOLVE task is being generated, for an equation of the form (EQUAL (f t_1 ... t_n) rhs) the task generator retrieves the solvers associated with the function f and tries to check whether the IF part of each solver is satisfied. If so, a SOLVE task for the corresponding solver is generated. Even in its current limited form the solver knowledge is very useful and considerably increases the efficiency and power of EGS. It is desirable that this solving scheme be further extended to deal with more complicated formulas. One important problem is how to organize and index solver knowledge such that efficient retrieval can be achieved.

4.8.2.7 Case Analyzer

The application of a case-analyzer splits the current constraint into new refined subproblem constraints. Case analysis differs from equation solving in that certain particular subterms in the constraint suggest different cases to be separately considered.

Just as with solver knowledge, the case analysis knowledge is in production rule form, coded in LISP, and given by the user. Currently we do case analysis on variables. However, this refinement scheme can be extended to include arbitrary patterns of terms.

To generate examples it is very often useful to split a constraint into different cases and consider each of them separately. In EGS such case spliting occurs when a constraint is simplified, especially after an EXPAND task is performed. During simplification the case split is carried out usually when a nested IF formula or a term containing IF terms is normalized. When a (recursive) function definition is expanded in a constraint the constraint can be split into different cases such as the base case and the recursion case. The definition expansion effectively performs case analysis. However, case analysis as a refinement scheme of EGS example generation is useful in the situation where no such functions can be found in the current constraint. Also case analysis can be proposed even by terms containing the defined functions in

them because it is more appropriate to case split heuristically than logically, as in definition expansion.

Currently the case analysis scheme is incorporated to analyze by cases shell primitive terms - terms consisting of variables, and shell functions, especially accessor functions. This is because in the Boyer-Moore system shell functions are defined in terms of axioms instead of explicit definitions. As the result shell functions have no symbolic definitions to expand. For example, suppose we have the constraint

```
((NUMBERP (CAR L))
 (NOT (LISTP (CDR L)))).
```

One might be tempted to replace the variable L with the term (CONS U V) where U and V are new variables. The idea of such a replacement is suggested by the terms (CAR L) and (CDR L) occurring in the constraint. Actually two cases must be considered: the case of (EQUAL L (CONS U V)) and the case of (NOT (LISTP L)). The case analyzer of CAR (or CDR) which is invoked by the term (CAR x) would be

```
IF-part:
   x is a variable;

THEN-part:
   create new constraints by adding to the original constraint
   each of the new conditions (EQUAL L (CONS U V)) where U
   and V are new variable and (NOT (LISTP L)), respectively.
```

When the ANALYZE task corresponding to the above case analyzer is performed, we have two new constraints

```
((LISTP (CONS U V))
 (NUMBERP (CAR (CONS U V)))
 (NOT (LISTP (CDR (CONS U V))))) and

((NOT (LISTP L))
 (NUMBERP (CAR L))
 (NOT (LISTP (CDR L)))).
```

The first case would be further simplified to

```
((NUMBERP U)
 (NOT (LISTP V)))
```

whereas the second case would be simplified to

```
((NOT (LISTP L))).
```

In both cases we now have more refined and simpler constraints.

4.8.2.8 Control Information

Control information is used to selectively control expanding of function definitions when formulas are simplified or an EXPAND task is performed. It is our general strategy that when a formula is simplified, we expand the definitions of non-recursive functions; for recursive functions we dive recursively into the definitions, attempting to further simplify the formula. In EGS example generation it sometimes may not be desirable to open up function definitions. The function PRIME, for example, is defined non-recursively in terms of PRIME1, which is defined recursively. When a formula containing PRIME is simplified, we do not expand the definition of PRIME. We favor PRIME over PRIME1 because we have more knowledge about PRIME than about PRIME1. There may be many lemmas, good stored examples, and various kinds of procedural knowledge associated with PRIME around. EGS can take advantages of PRIME instead of PRIME1. The FOLD control information under the name PRIME indicates that PRIME is not

opened up during simplification. For the same reason certain functions should not be expanded when the EXPAND task is performed.

Currently two types of the control information are used: FOLD and LOCK. The control information is a procedure which check conditions under which the corresponding control is applied. The FOLD information checks the condition under which the corresponding function should not be opened up; the LOCK information checks the condition for not expanding the definition of the corresponding function when an EXPAND task is performed. For instance, the FOLD information T for PRIME means that PRIME is never opened up when simplified.

One might consider this control scheme to be ad hoc. However, in an extensible system it may be desirable to devise a mechanism which selectively blocks particular unfavorable formulas from occurring during the process. As knowledge about particular concepts grows, the system would favor those concepts over others. Such a well designed control mechanism would enable the system to work at different levels of abstraction.

4.9 Definition Time Example Generation

EGS has a limited capability to generate examples of a function at the time the function is defined. When a new function is being defined EGS generates examples of the function, classifies them appropriately, and stores them under the function symbol in the knowledge base.

The following briefly describes how EGS generates examples at definition time. Each time a new function is admitted by the Boyer-Moore system EGS attempts to generate examples based on its symbolic definition and the known examples of the related functions. For generating examples at definition time, 1) we test various known examples and 2) we generate examples by considering the symbolic definition as a constraint. When we test known examples it is important which known examples are selected. We first heuristically compute the typical type of each of the formal arguments of the defined function. For example, the typical types of the arguments of MEMBER are "anything" for the first argument and a list for the the second argument. This means that typically the second argument of MEMBER is a list whereas the first argument is anything. Then we test the known examples of those types with the definition. We also test the known examples of those important functions whose function symbols occur in the definition. For the second case we generate examples for the constraints corresponding to the base case and recursion case of a recursive function.

The classification of generated examples is done heuristically. For instance, examples consisting of known boundary examples and examples for which the function evaluates to one of the boundary examples of the typical type of the function values are classified as boundary. We do classify typical examples similarly. Those examples generated from the base case of the definition should be classified as boundary whereas those generated from the recursion cases of the definition are classified as typical. Counter examples can be generated by taking the negation of the definition as a constraint or by collecting those examples which evaluate to F in the above process. If too many examples have been generated, we select some of them randomly. Figure 4-5 shows the examples generated by EGS at definition time for MEMBER and CRYPT. EGS employs various heuristics to find appropriate known examples to test and classify the generated examples into appropriate types. However, those heuristics are often found not very useful, especially, when the defined functions are of high level and their recursion structures are complicated. Also, the classification of examples largely depends on applications

```
(MEMBER X L) =
  (IF (LISTP L)
      (IF (EQUAL X (CAR L)) T (MEMBER X (CDR L)))
      F)
```

typical examples	*boundary examples*	*counter examples*
(3 ((3 8) D 3 5))	(D (D C . D))	(0 (2 3 1))
(A ((E) D A B))	(2 (2 3 5))	(8 (3 2 . 5))
(A ((3 8) 5 (3 8) A))	(A (A . B))	(7 B)
(A (5 5 (E) A B))	(A (A (A (A))))	(A (B C . A))
(NIL ((E) B NIL))		(6 (3 2 . 5))
(C (A B C))		((A) 5)
((A (A)) (C A (A (A))))		(NIL (A))
(1 (2 3 1))		(D (B C))
(3 (2 3 5))		(A (C))
(C (B C . A))		(A A)
((A (A)) (A (A (A))))		(2 B)
((A) ((3 8) C D A (A)))		(5 B)
(5 (2 3 5))		(0 (3 2 . 5))
(2 ((3 8) 1 2 . 3))		(1 (B C))
(3 ((E) (E) 2 3 5))		(A (2 3 1))

```
(CRYPT M E N) =
  (IF (ZEROP E)
      1
      (IF (EVEN E)
          (REMAINDER (SQUARE (CRYPT M (QUOTIENT E 2) N)) N)
          (REMAINDER
            (TIMES M
                   (REMAINDER (SQUARE (CRYPT M (QUOTIENT E 2) N))
                              N))
            N)))
```

typical examples	*boundary examples*
(0 7 0)	(A 2 5)
(3 5 2)	((3 2) NIL *1*TRUE)
(4 8 15)	(5 0 4)
(1 4 4)	(A 3 A)
(9 1 3)	((A B) C (A B))
(3 7 2)	((1 2 3) (A) *1*TRUE)
(3 3 2)	(0 2 (A B C))
(1 3 1)	(A 2 (A B))
(4 7 3)	(2 1 NIL))))
(5 5 7)	
(3 1 2)	
(4 7 15)	
(5 4 7)	
(2 5 5)	
(4 9 15)	

Figure 4-5: Examples Generated by EGS at Definition Time

and is hard to predict. The example generation at definition time is limited; the user still needs to provide good examples.

4.10 Some EGS-Generated Examples

In the following we list several cases of the EGS-generated examples. Each case is a constraint formula followed by a list of examples generated by EGS, the time in seconds EGS spent on generating the examples, and the cause of termination. For each case we limit EGS to generate (at most) 10 examples or spend 20 seconds. Most of the constraint formulas have been taken from the hypotheses of theorems about numbers and lists, which have been proved by the Boyer-Moore theorem prover. Many examples have been generated by testing known examples. However, some examples are generated by applying solvers, solving linear equations/inequalities, and expanding definitions. It may be interesting to compare the performance of EGS varying the EGS knowledge base. If we eliminate knowledge such as known examples and solvers associated with certain functions, how well EGS performs? Suppose we generate examples for (PRIME X) without the known examples of PRIME. EGS attempts to expand PRIME in (PRIME X) and the resulting formula involves lower level functions such as PRIME1, SUB1, and NUMBERP. EGS tests the known examples of those functions and generates several examples easily. EGS still generates examples for (PRIME X) relatively quickly without the known examples and solvers associated with functions such as PRIME1, REMAINDER, and QUOTIENT. In this case EGS tests the known examples of NUMBERP, SUB1, etc. In an experiment EGS generates four examples for (PRIME X) in 98 seconds without the known examples and solvers associated with the functions REMAINDER, QUOTIENT, TIMES, and NUMBERP. If we almost completely block knowledge from being used, EGS finds two examples - 2 and 3 - for (PRIME X) relatively easily, and then it seems to get lost. Case 3 in the following well demonstrates that EGS can still generate examples by expanding definitions without the known examples or solvers associated with some functions. It may be interesting to investigate the dependency of the performance of EGS on its knowledge base; no attempts have been made to do that in this research.

Case 1:
```
(NOT (EQUAL (IMPLIES P (IMPLIES Q R))
            (IMPLIES (IMPLIES P Q) R)))

((P . (QUOTE *1*FALSE))
 (Q . (QUOTE *1*TRUE))
 (R . (QUOTE *1*FALSE)))

0.1 EMPTY-AGENDA
```

Case 2:
```
(AND (ORDERED L)
     (NOT (EQUAL (SORT L) L)))

 ((L . (QUOTE (A B C))))

 ((L . (QUOTE A)))

 ((L . (QUOTE (1 2 3 . 0))))

 0.2 EMPTY-AGENDA
```

Case 3:
```
(AND (MEMBER 'P L)
     (MEMBER 'Q L)
     (EQUAL (REVERSE L) L))
```

```
((L . (QUOTE (Q P Q))))
((L . (QUOTE (P Q P))))
((L . (QUOTE (Q P P Q))))
((L . (QUOTE (P Q Q P))))
20.1 TIME-LIMIT
```

Case 4: (MEMBER X (DELETE U V))

```
((U . (QUOTE D)) (V . (QUOTE (A B C))) (X . (QUOTE B)))
((U . (QUOTE D)) (V . (QUOTE (A B C))) (X . (QUOTE A)))
((U . (QUOTE C)) (V . (QUOTE (A B C))) (X . (QUOTE A)))
((U . (QUOTE C)) (V . (QUOTE (A B C))) (X . (QUOTE B)))
((U . (QUOTE A)) (V . (QUOTE (A B C))) (X . (QUOTE B)))
((U . (QUOTE NIL)) (V . (QUOTE (NIL NIL))) (X . (QUOTE NIL)))
((U . (QUOTE 2)) (V . (QUOTE (5 3 2 2))) (X . (QUOTE 2)))
((U . (QUOTE A)) (V . (QUOTE (A A B C))) (X . (QUOTE B)))
0.5 EMPTY-AGENDA
```

Case 5: (SUBBAGP X (DELETE U Y))

```
((U . (QUOTE A)) (X . (QUOTE B)) (Y . (QUOTE (A))))
((U . (QUOTE A)) (X . (QUOTE A)) (Y . (QUOTE (A B C))))
((U . (QUOTE A)) (X . (QUOTE (B . D))) (Y . (QUOTE (A B C))))
((U . (QUOTE C)) (X . (QUOTE B)) (Y . (QUOTE (A B . C))))
((U . (QUOTE C)) (X . (QUOTE (B . D))) (Y . (QUOTE (A B . C))))
((U . (QUOTE C)) (X . (QUOTE (A B))) (Y . (QUOTE (A B . C))))
((U . (QUOTE (3 2))) (X . (QUOTE A)) (Y . (QUOTE B)))
((U . (QUOTE B)) (X . (QUOTE (A B C))) (Y . (QUOTE (B B C A))))
((U . (QUOTE B)) (X . (QUOTE (B B))) (Y . (QUOTE (A B B B))))
((U . (QUOTE A)) (X . (QUOTE NIL)) (Y . (QUOTE (A))))
0.5 COUNT-LIMIT
```

Case 6: `(ORDERED (APPEND A B))`

```
((A . (QUOTE A)) (B . (QUOTE (B C))))

((A . (QUOTE (A . B))) (B . (QUOTE (B . C))))

((A . (QUOTE (1 2))) (B . (QUOTE (3))))

((A . (QUOTE (U V 0 1 2))) (B . (QUOTE NIL)))

((A . (QUOTE (2 3))) (B . (QUOTE (4 5))))

((A . (QUOTE (2))) (B . (QUOTE (2 2))))

0.3  EMPTY-AGENDA
```

Case 7:
```
(AND (ORDERED X)
     (NUMBER-LISTP X)
     (NUMBERP I)
     (LEQ I (CAR X)))

((I . (QUOTE 3)) (X . (QUOTE (3 5 5))))

((I . (QUOTE 3)) (X . (QUOTE (3 5 7 11))))

((I . (QUOTE 3)) (X . (QUOTE (3 5 7))))

((I . (QUOTE 0)) (X . (QUOTE NIL)))

((I . (QUOTE 0)) (X . (QUOTE (1 2 3))))

((I . (QUOTE 1)) (X . (QUOTE (1 2 3))))

((I . (QUOTE 0)) (X . (QUOTE (2 7))))

((I . (QUOTE 2)) (X . (QUOTE (2 7))))

((I . (QUOTE 1)) (X . (QUOTE (2 7))))

((I . (QUOTE 1)) (X . (QUOTE (5 7))))

1.0 COUNT-LIMIT
```

Case 8:
```
(AND (EQUAL (REMAINDER X Z) 0)
     (EQUAL (REMAINDER Y Z) 0))

((X . (QUOTE 0)) (Y . (QUOTE 0)) (Z . (QUOTE 2)))

((X . (QUOTE 0)) (Y . (QUOTE 2)) (Z . (QUOTE 2)))

((X . (QUOTE 0)) (Y . (QUOTE 4)) (Z . (QUOTE 2)))

((X . (QUOTE 3)) (Y . (QUOTE A)) (Z . (QUOTE 3)))

((X . (QUOTE 3)) (Y . (QUOTE 9)) (Z . (QUOTE 3)))
```

```
             ((X . (QUOTE 3)) (Y . (QUOTE 3)) (Z . (QUOTE 3)))

             ((X . (QUOTE 8)) (Y . (QUOTE 0)) (Z . (QUOTE 4)))

             ((X . (QUOTE 8)) (Y . (QUOTE 4)) (Z . (QUOTE 4)))

             ((X . (QUOTE 8)) (Y . (QUOTE 8)) (Z . (QUOTE 4)))

             ((X . (QUOTE 0)) (Y . (QUOTE 10)) (Z . (QUOTE 2)))

             0.6 COUNT-LIMIT
```

Case 9:

```
(AND (NOT (ZEROP X))
     (NOT (ZEROP Y))
     (DIVIDES Z X)
     (DIVIDES Z Y))

     ((X . (QUOTE 4)) (Y . (QUOTE 2)) (Z . (QUOTE 2)))

     ((X . (QUOTE 4)) (Y . (QUOTE 4)) (Z . (QUOTE 2)))

     ((X . (QUOTE 4)) (Y . (QUOTE 8)) (Z . (QUOTE 2)))

     ((X . (QUOTE 8)) (Y . (QUOTE 4)) (Z . (QUOTE 4)))

     ((X . (QUOTE 8)) (Y . (QUOTE 8)) (Z . (QUOTE 4)))

     ((X . (QUOTE 4)) (Y . (QUOTE 6)) (Z . (QUOTE 2)))

     ((X . (QUOTE 4)) (Y . (QUOTE 16)) (Z . (QUOTE 2)))

     ((X . (QUOTE 4)) (Y . (QUOTE 10)) (Z . (QUOTE 2)))

     ((X . (QUOTE 8)) (Y . (QUOTE 28)) (Z . (QUOTE 4)))

     ((X . (QUOTE 8)) (Y . (QUOTE 32)) (Z . (QUOTE 4)))

     5.6 COUNT-LIMIT
```

Case 10:

```
(AND (LESSP Y X)
     (NOT (PRIME1 X Y))
     (NOT (ZEROP X))
     (NOT (EQUAL (SUB1 X) 0))
     (NOT (ZEROP Y)))

     ((X . (QUOTE 6)) (Y . (QUOTE 4)))

     ((X . (QUOTE 4)) (Y . (QUOTE 3)))

     ((X . (QUOTE 4)) (Y . (QUOTE 2)))

     1.7 EMPTY-AGENDA
```

Case 11:

```
(AND (NUMBERP Y)
     (NUMBERP X)
     (NOT (EQUAL X 0))
     (DIVIDES X Y)
```

```
((X . (QUOTE 2)) (Y . (QUOTE 0)))

((X . (QUOTE 3)) (Y . (QUOTE 3)))

((X . (QUOTE 2)) (Y . (QUOTE 4)))

((X . (QUOTE 1)) (Y . (QUOTE 4)))

((X . (QUOTE 3)) (Y . (QUOTE 15)))

((X . (QUOTE 5)) (Y . (QUOTE 5)))

((X . (QUOTE 6)) (Y . (QUOTE 0)))

((X . (QUOTE 4)) (Y . (QUOTE 0)))

((X . (QUOTE 4)) (Y . (QUOTE 8)))

((X . (QUOTE 4)) (Y . (QUOTE 4)))

1.0 COUNT-LIMIT
```

Case 12:

```
(AND (NOT (EQUAL Z 1))
     (NOT (EQUAL Z X))
     (NOT (ZEROP X))
     (NOT (EQUAL X 1))
     (DIVIDES Z X))

((X . (QUOTE 8)) (Z . (QUOTE 4)))

((X . (QUOTE 9)) (Z . (QUOTE 3)))

((X . (QUOTE 6)) (Z . (QUOTE 2)))

((X . (QUOTE 6)) (Z . (QUOTE 3)))

((X . (QUOTE 12)) (Z . (QUOTE 3)))

((X . (QUOTE 15)) (Z . (QUOTE 3)))

((X . (QUOTE 20)) (Z . (QUOTE 5)))

((X . (QUOTE 10)) (Z . (QUOTE 5)))

((X . (QUOTE 35)) (Z . (QUOTE 5)))

((X . (QUOTE 12)) (Z . (QUOTE 4)))

2.7 COUNT-LIMIT
```

Case 13:

```
(AND (NOT (DIVIDES X A))
     (EQUAL A (GCD (TIMES X A) (TIMES B A))))

((A . (QUOTE 5)) (B . (QUOTE 1)) (X . (QUOTE NIL)))

((A . (QUOTE 2)) (B . (QUOTE 5)) (X . (QUOTE 3)))
```

```
            ((A . (QUOTE 2)) (B . (QUOTE 2)) (X . (QUOTE 3)))

            ((A . (QUOTE 3)) (B . (QUOTE 3)) (X . (QUOTE 2)))

            ((A . (QUOTE 3)) (B . (QUOTE 5)) (X . (QUOTE 2)))

            ((A . (QUOTE 8)) (B . (QUOTE 4)) (X . (QUOTE 3)))

            ((A . (QUOTE 8)) (B . (QUOTE 5)) (X . (QUOTE 3)))

            ((A . (QUOTE 2)) (B . (QUOTE 1)) (X . (QUOTE NIL)))

            ((A . (QUOTE 1)) (B . (QUOTE 3)) (X . (QUOTE 5)))

            ((A . (QUOTE 1)) (B . (QUOTE 18)) (X . (QUOTE 5)))

            2.0 COUNT-LIMIT
```

Case 14:

```
(AND (NOT (DIVIDES X B))
     (NOT (ZEROP X))
     (NOT (EQUAL (SUB1 X) 0))
     (PRIME1 X (SUB1 X)))

     ((B . (QUOTE 5)) (X . (QUOTE 3)))

     ((B . (QUOTE 7)) (X . (QUOTE 5)))

     ((B . (QUOTE 2)) (X . (QUOTE 5)))

     ((B . (QUOTE 3)) (X . (QUOTE 5)))

     ((B . (QUOTE 8)) (X . (QUOTE 5)))

     ((B . (QUOTE 2)) (X . (QUOTE 13)))

     ((B . (QUOTE 8)) (X . (QUOTE 13)))

     ((B . (QUOTE 2)) (X . (QUOTE 11)))

     ((B . (QUOTE 8)) (X . (QUOTE 11)))

     ((B . (QUOTE 3)) (X . (QUOTE 11)))

     1.2 COUNT-LIMIT
```

Case 15:

```
(AND (NUMBERP Z)
     (PRIME X)
     (NOT (DIVIDES X Z))
     (NOT (DIVIDES X B)))

  ((B . (QUOTE 2)) (X . (QUOTE 3)) (Z . (QUOTE 5)))

  ((B . (QUOTE 4)) (X . (QUOTE 3)) (Z . (QUOTE 5)))

  ((B . (QUOTE 5)) (X . (QUOTE 3)) (Z . (QUOTE 5)))

  ((B . (QUOTE 2)) (X . (QUOTE 5)) (Z . (QUOTE 7)))
```

```
((B . (QUOTE 9)) (X . (QUOTE 5)) (Z . (QUOTE 7)))

((B . (QUOTE 4)) (X . (QUOTE 5)) (Z . (QUOTE 7)))

((B . (QUOTE 2)) (X . (QUOTE 7)) (Z . (QUOTE 2)))

((B . (QUOTE 8)) (X . (QUOTE 7)) (Z . (QUOTE 2)))

((B . (QUOTE 5)) (X . (QUOTE 7)) (Z . (QUOTE 2)))

((B . (QUOTE 3)) (X . (QUOTE 7)) (Z . (QUOTE 2)))

1.8 COUNT-LIMIT
```

Case 16:

```
(AND (PRIME X)
     (NOT (EQUAL Y 1))
     (NOT (EQUAL X Y)))

     ((X . (QUOTE 5)) (Y . (QUOTE 2)))

     ((X . (QUOTE 2)) (Y . (QUOTE 3)))

     ((X . (QUOTE 23)) (Y . (QUOTE 0)))

     ((X . (QUOTE 37)) (Y . (QUOTE 0)))

     ((X . (QUOTE 17)) (Y . (QUOTE 2)))

     ((X . (QUOTE 19)) (Y . (QUOTE 2)))

     ((X . (QUOTE 7)) (Y . (QUOTE 8)))

     ((X . (QUOTE 31)) (Y . (QUOTE 8)))

     ((X . (QUOTE 3)) (Y . (QUOTE 0)))

     ((X . (QUOTE 5)) (Y . (QUOTE 0)))

     0.6 COUNT-LIMIT
```

Case 17:

```
(AND (PRIME C)
     (PRIME-LIST L2)
     (NOT (MEMBER C L2))

     ((C . (QUOTE 2)) (L2 . (QUOTE *1*FALSE)))

     ((C . (QUOTE 2)) (L2 . (QUOTE 1)))

     ((C . (QUOTE 37)) (L2 . (QUOTE *1*TRUE)))

     ((C . (QUOTE 37)) (L2 . (QUOTE 3)))

     ((C . (QUOTE 7)) (L2 . (QUOTE 1)))

     ((C . (QUOTE 11)) (L2 . (QUOTE 1)))
```

```
              ((C . (QUOTE 13)) (L2 . (QUOTE *1*FALSE)))

              ((C . (QUOTE 2)) (L2 . (QUOTE (5 . A))))

              ((C . (QUOTE 2)) (L2 . (QUOTE (5 . 5))))

              ((C . (QUOTE 2)) (L2 . (QUOTE (5 5 . A))))

              3.7 COUNT-LIMIT
```

Case 18:

```
(AND (EQUAL (TIMES C (TIMES-LIST L1))
            (TIMES-LIST L2))
     (PRIME C)
     (PRIME-LIST L2))

((C . (QUOTE 7))
 (L1 . (QUOTE 1))
 (L2 . (QUOTE (7))))

((C . (QUOTE 5))
 (L1 . (QUOTE (7 2)))
 (L2 . (QUOTE (7 5 2))))

((C . (QUOTE 5))
 (L1 . (QUOTE 5))
 (L2 . (QUOTE (5))))

((C . (QUOTE 7))
 (L1 . (QUOTE C))
 (L2 . (QUOTE (7))))

((C . (QUOTE 2))
 (L1 . (QUOTE (3 5 1)))
 (L2 . (QUOTE (5 3 2))))

((C . (QUOTE 5))
 (L1 . (QUOTE NIL))
 (L2 . (QUOTE (5))))

((C . (QUOTE 5))
 (L1 . (QUOTE (2 7)))
 (L2 . (QUOTE (7 5 2))))

((C . (QUOTE 5))
 (L1 . (QUOTE (1 7 2)))
 (L2 . (QUOTE (7 5 2))))

((C . (QUOTE 5))
 (L1 . (QUOTE (14 . 5)))
 (L2 . (QUOTE (7 5 2))))

((C . (QUOTE 5))
 (L1 . (QUOTE (14 . A)))
 (L2 . (QUOTE (7 5 2))))

5.8 COUNT-LIMIT
```

```
Case 19:  (AND (PRIME-LIST L1)
               (PRIME-LIST L2)
               (EQUAL (TIMES-LIST L1)
                      (TIMES-LIST L2)))

               ((L1 . (QUOTE NIL)) (L2 . (QUOTE B)))

               ((L1 . (QUOTE (2 5 7))) (L2 . (QUOTE (2 5 7))))

               ((L1 . (QUOTE (2 5 7))) (L2 . (QUOTE (7 5 2))))

               ((L1 . (QUOTE (7))) (L2 . (QUOTE (7))))

               ((L1 . (QUOTE NIL)) (L2 . (QUOTE 1)))

               ((L1 . (QUOTE (2 5 7))) (L2 . (QUOTE (5 2 7))))

               ((L1 . (QUOTE NIL)) (L2 . (QUOTE 0)))

               ((L1 . (QUOTE (3 3))) (L2 . (QUOTE (3 3))))

               ((L1 . (QUOTE NIL)) (L2 . (QUOTE 2)))

               ((L1 . (QUOTE (5))) (L2 . (QUOTE (5))))

               1.0 COUNT-LIMIT

Case 20:  (AND (PLISTP Y)
               (ORDERED2 Y)
               (NOT (EQUAL X V)))

      ((V . (QUOTE 2)) (X . (QUOTE 5)) (Y . (QUOTE NIL)))

      ((V . (QUOTE 2)) (X . (QUOTE 5)) (Y . (QUOTE ((A)))))

      ((V . (QUOTE 2)) (X . (QUOTE 5)) (Y . (QUOTE (A B C))))

      ((V . (QUOTE 2)) (X . (QUOTE 5)) (Y . (QUOTE (2 2 2))))

      ((V . (QUOTE 2)) (X . (QUOTE 5)) (Y . (QUOTE (7 5 3 2))))

      ((V . (QUOTE 2)) (X . (QUOTE 5)) (Y . (QUOTE (8 6 4 2 0))))

      ((V . (QUOTE 1)) (X . (QUOTE 0)) (Y . (QUOTE NIL)))

      ((V . (QUOTE 1)) (X . (QUOTE 0)) (Y . (QUOTE (C))))

      ((V . (QUOTE 1)) (X . (QUOTE 0)) (Y . (QUOTE (A B C))))

      ((V . (QUOTE 1)) (X . (QUOTE 0)) (Y . (QUOTE (3)))))

      0.2 COUNT-LIMIT

Case 21:  (EQUAL (REMAINDER Y A)
                 (REMAINDER Z A))

        ((A . (QUOTE 0)) (Y . (QUOTE 0)) (Z . (QUOTE 0)))
```

```
((A . (QUOTE 3)) (Y . (QUOTE 2)) (Z . (QUOTE 2)))

((A . (QUOTE 3)) (Y . (QUOTE 2)) (Z . (QUOTE 5)))

((A . (QUOTE 3)) (Y . (QUOTE 9)) (Z . (QUOTE 0)))

((A . (QUOTE 3)) (Y . (QUOTE 9)) (Z . (QUOTE 9)))

((A . (QUOTE 3)) (Y . (QUOTE 9)) (Z . (QUOTE 3)))

((A . (QUOTE 3)) (Y . (QUOTE 2)) (Z . (QUOTE 17)))

((A . (QUOTE 3)) (Y . (QUOTE 9)) (Z . (QUOTE 6)))

((A . (QUOTE 0)) (Y . (QUOTE 0)) (Z . (QUOTE *1*FALSE)))

((A . (QUOTE 0)) (Y . (QUOTE 0)) (Z . (QUOTE C)))

0.4 COUNT-LIMIT
```

Case 22:

```
(AND (EQUAL (REMAINDER (TIMES M A) P)
            (REMAINDER (TIMES M B) P))
     (NOT (EQUAL (REMAINDER M P) 0))
     (PRIME P))

     ((A . (QUOTE 2)) (B . (QUOTE 5))
      (M . (QUOTE 8)) (P . (QUOTE 3)))

     ((A . (QUOTE 2)) (B . (QUOTE 2))
      (M . (QUOTE 8)) (P . (QUOTE 3)))

     ((A . (QUOTE 5)) (B . (QUOTE 2))
      (M . (QUOTE 8)) (P . (QUOTE 3)))

     ((A . (QUOTE 5)) (B . (QUOTE 5))
      (M . (QUOTE 8)) (P . (QUOTE 3)))

     ((A . (QUOTE 3)) (B . (QUOTE 0))
      (M . (QUOTE 8)) (P . (QUOTE 3)))

     ((A . (QUOTE 3)) (B . (QUOTE 9))
      (M . (QUOTE 8)) (P . (QUOTE 3)))

     ((A . (QUOTE 3)) (B . (QUOTE 3))
      (M . (QUOTE 8)) (P . (QUOTE 3)))

     ((A . (QUOTE 4)) (B . (QUOTE 4))
      (M . (QUOTE 7)) (P . (QUOTE 5)))

     ((A . (QUOTE 1)) (B . (QUOTE 1))
      (M . (QUOTE 7)) (P . (QUOTE 5)))

     ((A . (QUOTE NIL)) (B . (QUOTE A))
      (M . (QUOTE 7)) (P . (QUOTE 5)))

     1.5 COUNT-LIMIT
```

Case 23:

```
(AND (PRIME P)
     (PRIME Q)
     (NOT (EQUAL P Q))
     (EQUAL (REMAINDER A P)
            (REMAINDER B P))
     (EQUAL (REMAINDER A Q)
            (REMAINDER B Q))
     (NUMBERP B)
     (LESSP B (TIMES P Q)))

     ((A . (QUOTE 3)) (B . (QUOTE 3))
      (P . (QUOTE 2)) (Q . (QUOTE 3)))

     ((A . (QUOTE 9)) (B . (QUOTE 3))
      (P . (QUOTE 2)) (Q . (QUOTE 3)))

     ((A . (QUOTE 0)) (B . (QUOTE 0))
      (P . (QUOTE 2)) (Q . (QUOTE 7)))

     ((A . (QUOTE 0)) (B . (QUOTE 0))
      (P . (QUOTE 2)) (Q . (QUOTE 13)))

     ((A . (QUOTE 4)) (B . (QUOTE 4))
      (P . (QUOTE 2)) (Q . (QUOTE 11)))

     ((A . (QUOTE 4)) (B . (QUOTE 4))
      (P . (QUOTE 2)) (Q . (QUOTE 37)))

     ((A . (QUOTE 3)) (B . (QUOTE 3))
      (P . (QUOTE 2)) (Q . (QUOTE 31)))

     ((A . (QUOTE 3)) (B . (QUOTE 3))
      (P . (QUOTE 2)) (Q . (QUOTE 11)))

     ((A . (QUOTE A)) (B . (QUOTE 0))
      (P . (QUOTE 2)) (Q . (QUOTE 5)))

     ((A . (QUOTE 0)) (B . (QUOTE 0))
      (P . (QUOTE 2)) (Q . (QUOTE 3)))

     6.0 COUNT-LIMIT
```

5. EGS: An Application

Examples can used in many different ways. Especially, the theorem proving environment of the Boyer-Moore system serves as a rich source of applications of examples. Examples can be used to guide the search for proofs, check conjectures, develop new concepts and hypotheses, etc. We are specially interested in using examples to control backward chaining of the Boyer-Moore theorem prover. The backward chaining problem is important because a substantial portion of the cost of the theorem prover is due to uncontrolled backward chaining. In this chapter we describe an experiment of attempting to use examples for the problem. We first introduce the problem of backward chaining in the theorem prover. We have interfaced EGS with the theorem prover to generate examples to be used in the theorem prover and then describe the interface. Finally, we present some statistics on the experiment.

5.1 Controlling Backward Chaining

5.1.1 Referencing Problem

It is critical that in a knowledge based system knowledge should be retrieved efficiently and used effectively. A large body of knowledge may not be useful and sometimes may detract from the performance of the system unless it is under a good control that allows only relevant and useful knowledge to be retrieved and used. Increased knowledge should improve a system's performance; however, it typically expands the search space and may cause inefficient search because of the larger search space. Bledsoe has named this the **referencing problem** [3]. The referencing problem arises not only in theorem proving but in many systems whose functions are based on their internal knowledge where such knowledge is extensible. Resolution theorem proving suffers considerably from this problem. To avoid or reduce this problem, knowledge can be divided into different groups. When knowledge is being applied, only knowledge in those groups which best match the current task is retrieved. In this case, however, such grouping is static and largely based on the future use of knowledge, which is often difficult to predict. Another solution to the referencing problem is a machine learning scheme. In this scheme knowledge is dynamically reorganized as it is used. The status in the knowledge structure is upgraded for those pieces of knowledge which have been used successfully. This enables knowledge to be organized in such a way that useful knowledge is more easily accessible than less useful knowledge.

Attempts have been made in using semantic information to solve the referencing problem, especially in the area of theorem proving. Gelernter has experimented with the idea that a human uses diagrams when proving geometry theorems. The diagram is considered as a (approximate) model of the conjecture being proved and conveys semantic information about geometry. When any existing knowledge such as axioms and lemmas is being used, the knowledge is checked against the diagram to see whether it would be useful in the current context. Using diagrams he succeeded in reducing the search space of his geometry machine by several orders of magnitude. Several other researchers also have experimented with using examples in theorem proving. They include Ballantyne and Bledsoe, Reiter, Aubin, and Brotz [2, 40, 1, 9].

5.1.2 Referencing Problem in the Boyer-Moore Theorem Prover

The referencing problem also arises in the Boyer-Moore theorem prover. Theorem proving is carried out primarily by lemma-driven simplification and induction. Simplification is an important component of the theorem prover; about 80% of theorem proving effort (in time) is spent on simplifying formulas. When a formula is simplified the simplifier attempts to rewrite it into a simpler form by using lemmas. However, the lemma base consists of axioms which are created automatically by the system (or given by the user) and of previously proven theorems. Each time a theorem has been proved, it is stored in the knowledge base of the theorem prover with the type specified by the user. The type of lemma specifies the way that the lemma is used in the theorem prover. Potential types include: rewrite lemma, elimination lemma, generalization lemma, and induction lemma. The simplifier only uses rewrite lemmas when it simplifies formulas; therefore, we are most concerned with rewrite lemmas. In the Boyer-Moore theorem prover, rewrite lemmas are stored under the key function symbol occurring in the formula stating the lemma. The order in which the lemmas are stored and retrieved is important. Currently, the theorem prover has adopted the strategy that lemmas are stored in reverse chronological order and they are retrieved in that order. This simple strategy has proved to be effective even when the theorem prover operates within an environment that contains approximately 2000 lemmas. To prove a major theorem it is generally the case that the user gets appropriate lemmas proved before the theorem is proved and those lemmas are carefully stated and ordered by the user for the final proof of the major theorem. Thus, the strategy of the reverse chronological order works. Also the Boyer-Moore theorem prover allows the user to explicitly specify that particular lemmas should be used and/or disabled. However, the referencing problem still remains critical to the theorem prover when its library of lemmas grows to a considerable size.

5.1.3 Backward Chaining in the Boyer-Moore Theorem Prover

To make things worse, when lemmas are used in proving a theorem they are often chained backward in arbitrary depth. For example, to prove a conjecture p applying a lemma (IMPLIES h c) where c unifies with p, it is required to prove the corresponding instance h' of h. To prove h', it may be necessary to prove another hypothesis, and so on. This backward chaining expands the proof search space exponentially in terms of the number of the available lemmas. We can effectively use examples to control this backward chaining problem. Because backward chaining occurs recursively the early pruning of unpromising paths of reasoning would save a lot of proof search effort.

The following describes how examples are used for controlling backward chaining in the Boyer-Moore theorem prover. We first review simplification in the Boyer-Moore theorem prover. The simplifier primarily rewrites the conjectures generated during a proof attempt into simpler form. It may reduce a conjecture to T (true) or to some simpler form so that other routines such as generalization and induction can proceed smoothly with it. Conjectures are represented in the clausal form - a list of literals disjoined. Suppose we are in the process of simplifying the clause

$$\{ l_1 \;\; l_2 \;\; . \; . \; . \;\; l_n \}.$$

The simplifier rewrites each literal l_i sequentially under the assumption that all the literals except l_i are false. Even though the simplifier employs various techniques such as, using type information and the linear arithmetic package to simplify formulas, we are most concerned with simplifying them with lemmas. Simplification in the theorem prover is carried out from inside to outside. Suppose that we are to simplify the literal l_i of the form (p t_1 ... t_l). Each t_i must be simplified before the whole literal is simplified; this applies recursively to the simplification of the t_i's as well. Suppose now we simplify a term t occurring in l_i

and try to apply a lemma of the form

```
(IMPLIES hyp (EQUAL lhs rhs)).
```

The rewrite lemma is interpreted to mean that any instance of lhs can be replaced by the corresponding instance of rhs, provided the corresponding instance of hyp is true. Suppose that the term t is an instance of lhs under some substitution s. Then we instantiate the hypothesis hyp of the lemmas with s and rewrite the hypothesis recursively. If the hypothesis rewrites to true, then the simplifier replaces the term t by the instantiated rhs and then recursively rewrites that. When the simplifier tries to rewrite the hypothesis recursively backward chaining can occur via another lemma; here we can use examples to control it. Whenever we attempt to establish such a hypothesis we can evaluate the hypothesis with some examples before the simplifier is called. If any example causes the hypothesis to evaluate to false, then the hypothesis can never be established. In this case the counter-example also needs to satisfy the assumption that all the literals in the clause except l_i must be false; now, we reject the lemma as not applicable. In this way examples can be effectively used to prune unpromising backward chaining in the theorem prover and to cause a substantial reduction of the proof search space. The effect is even greater if there are a large number of available lemmas. In the following section we will describe the interface that we have made between the Boyer-Moore theorem prover and EGS in order to use examples for controlling backward chaining in the theorem prover.

5.1.4 Using Examples in the Boyer-Moore Theorem Prover

To make the Boyer-Moore theorem prover capable of using examples, we first need to interface our example generator with the theorem prover. In the integrated system, EGS generates examples and the theorem prover uses those generated examples while proving theorems. The proved theorems are also stored as lemmas to be used by the example generator; both components of the system are symbiotic. Also, for examples to be used effectively by the theorem prover, we need to slightly modify the relevant routines of the prover. This experiment of using examples in theorem proving has raised several questions. Each of them requires appropriate decisions to be made. They are the following:

- When should examples be generated? What type of examples - typical or boundary - and how many of them need to be generated?
- How should resources such as time be allocated for example generation?
- How should the generated examples be maintained through the theorem proving process?
- How should the examples actually be used?

Answers to these questions are closely related to the structure of the theorem prover. Incorporating the example facility into the theorem prover requires a fairly deep understanding of the theorem prover's functions and a close examination of the theorem prover's structure. In the following we will describe how such incorporation has been implemented and explain the decisions made for each of the above questions where appropriate.

The first decision we need to make is when to generate examples. This question requires careful analysis as to when the theorem prover - more precisely the simplifier - needs examples. As indicated above it would be *ideal* if examples could be generated each time a literal in a clause is being rewritten with lemmas. However, it is true that during the proof of a theorem the need to simplify literals occurs quite often and if we consider the non- negligible cost for example generation, we can easily conclude that it is infeasible to generate examples for every simplification. Another alternative is to generate many

examples once and for all, by considering the various cases which can be obtained by modifying the conjecture. This is done at the very early stage of the proof attempt and later on we simply retrieve and use the stored examples. In this case we have two further choices:

1. we never generate examples in the middle of proof;

2. we generate new examples when we find that none of the early examples succeeded.

In our experiment we have actually tried with the second strategy first because it seemed more appropriate than the first. Unfortunately, we had to give it up for an important reason[34]. We finally chose the first strategy.

5.1.4.1 Example Generation in the Boyer-Moore Theorem Prover

Given a theorem to be proven, the theorem prover now calls for EGS to generate examples for the theorem before any attempts to prove the theorem are made. For a given theorem, examples are generated, once and for all, at the beginning of the proof. Those examples are retained and used as needed. Suppose that we have a theorem of the form

(IMPLIES (AND h_1 h_2 ... h_n) c)[35].

We initially generate examples for each of the following constraints:

(AND h_1 h_2 ... h_n),
(AND (NOT h_1) h_2 ... h_n (NOT c)),
(AND h_1 (NOT h_2) ... h_n (NOT c)),
.
.
.

(AND h_1 h_2 ... (NOT h_n) (NOT c)).

Each constraint except the first one expresses how significant the corresponding hypotheses is to the conclusion; in other words, examples for each constraint illustrate why the corresponding hypothesis is necessary to make the conclusion true in the theorem. Here it is true that such constraints can be constructed arbitrarily. For instance, one can generate examples for less restricting constraints like (AND h_1 h_2), (NOT h_1), and (AND (NOT h_2) h_3), etc. However, those examples may not be very useful during the proof because such examples rarely convey the semantics of the stated conjecture and possibly they cause the rejection of even promising paths of reasoning.

For each generated constraint EGS proceeds to generate examples until either the specified number of examples have been generated or the given time limit has been exceeded[36].

Once examples have been generated for each of the constraints, we collect them and perform some simple checking, such as checking for multiple occurrences of the same example. We call the resulting collection of examples the **initial example set** with respect to the theorem. The following shows the initial example sets generated for two theorems.

---

[34]The reason is explained in the next chapter.

[35]This form is regarded as general. We assume that the user's statement of the theorem is reasonably well written.

[36]The user can explicitly specify the limits. In our experiment we usually set the number of examples and the time allocation to 6 examples and 10 seconds, respectively. These numbers have been determined by considering the performance of EGS and the efficiency of the theorem prover.

```
Theorem:
  (IMPLIES (PLISTP L)
           (EQUAL (REVERSE (REVERSE L)) L))

Initial Example Set:
        (((L . (QUOTE (A B C))))
         ((L . (QUOTE (C))))
         ((L . (QUOTE NIL)))
         ((L . (QUOTE (A . A))))
         ((L . (QUOTE (3 7 . 3))))
         ((L . (QUOTE (C B . A))))
         ((L . (QUOTE A))))


Theorem:
 (IMPLIES (AND (PRIME C)
               (PRIME-LIST L)
               (NOT (MEMBER C L)))
          (NOT (EQUAL (REMAINDER (TIMES-LIST L) C) 0)))

Initial Example Set:
       (((C . (QUOTE 7)) (L . (QUOTE A)))
        ((C . (QUOTE 7)) (L . (QUOTE B)))
        ((C . (QUOTE 31)) (L . (QUOTE A)))
        ((C . (QUOTE 31)) (L . (QUOTE B)))
        ((C . (QUOTE 29)) (L . (QUOTE A)))
        ((C . (QUOTE 29)) (L . (QUOTE B)))
        ((C . (QUOTE 1)) (L . (QUOTE (7 2))))
        ((C . (QUOTE 14)) (L . (QUOTE (7 2))))
        ((C . (QUOTE 1)) (L . (QUOTE (7))))
        ((C . (QUOTE 1)) (L . (QUOTE 1)))
        ((C . (QUOTE 1)) (L . (QUOTE (2 3))))
        ((C . (QUOTE 1)) (L . (QUOTE (3 2 . 1))))
        ((C . (QUOTE 2)) (L . (QUOTE (A))))
        ((C . (QUOTE 2)) (L . (QUOTE (A B C))))
        ((C . (QUOTE 2)) (L . (QUOTE (0))))
        ((C . (QUOTE 3)) (L . (QUOTE (6 4))))
        ((C . (QUOTE 2)) (L . (QUOTE (6 4))))
        ((C . (QUOTE 2)) (L . (QUOTE (4 3))))
        ((C . (QUOTE 3)) (L . (QUOTE (3 2 5))))
        ((C . (QUOTE 7)) (L . (QUOTE (7))))
        ((C . (QUOTE 2)) (L . (QUOTE (3 2 5))))
        ((C . (QUOTE 5)) (L . (QUOTE (3 2 5))))
        ((C . (QUOTE 2)) (L . (QUOTE (5 3 2))))
        ((C . (QUOTE 3)) (L . (QUOTE (3 5)))))
```

5.1.5 How are examples actually used in the theorem prover?

Our strategy in handling examples in the theorem prover is to associate each clause with an appropriate example set. We initially associate the initial example set with the clause corresponding to the theorem. Example sets are inherited; each of the clauses derived from a parent clause is associated with the parent's example set. When an example set is assigned to a clause we need to make a minor adjustment on it. An example is a list of variable bindings each of which is represented by a variable/constant value pair. It may be the case that some bound variables in an example never occur in the associated clause. We need to eliminate such pairs from examples in the example set. The reason

why this adjustment is necessary is that it is possible that some variables are eliminated from clauses during the proof. When a new variable is introduced, the theorem prover generates a variable symbol arbitrarily, possibly one of the eliminated variable symbols but with a different meaning attached to it[37]. Without such adjustment the associated examples do not correctly correspond to the new conjecture and may become useless.

When one of the literals in the clause is simplified the simplifier assumes that all other literals are false; the literals appearing before the literal being simplified in the clause have already been simplified. Suppose we have a clause

$$\{ l_1 \ \ l_2 \ \ . \ . \ . \ \ l_i \ \ . \ . \ . \ \ l_n \}$$

with its associated example set s and we are about to simplify the literal li. And suppose that we have the following new version of the clause at the time the literal li is being simplified.

$$\{ l_1' \ \ l_2' \ \ . \ . \ . \ \ l_i \ \ . \ . \ . \ \ l_n \}$$

where the l_j''s are the simplified results of the l_j's. We first need to select from the example set s those examples which satisfy each of the following: (NOT l_1'), (NOT l_2'), ..., (NOT l_{i-1}'), (NOT l_{i+1}), ..., (NOT l_n). We call those examples the **selected examples**. Next the selected examples are used in simplifying any terms occurring in the literal l_i. When a term t occurring in l_i is being rewritten with a lemma and we need to establish the instantiated hypothesis of the lemma, we first evaluate the hypothesis in the corresponding environment of each of the selected examples. When an example falsifies the hypothesis, we can safely conclude that the attempt to apply the lemma would fail. The simplifier stops applying the lemma and tries to apply the next available lemma. If such an example falsifying the hypothesis cannot be found, then simplification proceeds as normal.

As mentioned above, during a proof the Boyer-Moore theorem prover can possibly introduce new variables into conjectures. This problem is important for maintaining examples properly throughout the proof. There are two places where new variables are introduced in the theorem prover: destructor elimination and the generalization procedures[38]. Destructor elimination attempts to trade bad terms for good terms. For example, under an appropriate assumption every occurrence of the terms (REMAINDER X Y) and (QUOTIENT X Y) can be replaced in a conjecture by new variables, say I and J, respectively. We need to add restrictions on the new variables such as (NUMBERP I), (NUMBERP J), and (EQUAL (PLUS I (TIMES Y J)) X) to the hypothesis of the resulting conjecture. In this case we favor the terms involving PLUS and TIMES over those involving REMAINDER and QUOTIENT because more knowledge (lemmas) about the new functions is available. The generalization routine replaces certain common subterms in a conjecture with new variables. For example, during the proof of

```
(IMPLIES (PLISTP X)
         (EQUAL (REVERSE (REVERSE X)) X))
```

we have the conjecture

```
(IMPLIES (PLISTP X)
         (EQUAL (REVERSE (APPEND (REVERSE X) (LIST Z)))
```

[37]For new variables introduced we will discuss separately.

[38]To be more precise, new variables can also be introduced by the simplification and induction routines. Applying hints possibly cause new variables to be introduced. However, those cases are very rare and so unusual that they have been ignored in this experiment.

```
(CONS Z (REVERSE (REVERSE X)))))
```

to be processed by the generalization routine. The subterm (REVERSE X) is replaced by a new variable Y to produce:

```
(IMPLIES (PLISTP X)
         (EQUAL (REVERSE (APPEND Y (LIST Z)))
                (CONS Z (REVERSE Y)))).
```

We notice that each time a new variable is introduced by the theorem prover, a certain old term is replaced by the variable; such an introduction of variables is carefully traced by the interface of EGS with the theorem prover. Based on information about the terms being replaced and the associated example set of the current clause, we can easily construct the new example set to be associated with the new clauses. For each example in the example set we compute the value for the new variable by evaluating the term being replaced under the environment of the example and we add to the example the pair of the new variable and its value. Consider the above generalization case. The example (X . (QUOTE (B C))) (Z . (QUOTE A))) associated with the original conjecture would be adjusted to become the example ((X . (QUOTE (B C))) (Z . (QUOTE A)) (Y . (QUOTE (C B)))) associated with the generalized conjecture. We also need to remove from the example those pairs corresponding to the eliminated variables.

The following is an illustration of proving a theorem using examples for controlling backward chaining. We annotate some conjectures with their associated example sets (examples are printed as unQUOTEd).

```
(PROVE-LEMMA DIVIDES-TIMES-LIST
             (REWRITE)
             (IMPLIES (AND (NOT (ZEROP C)) (MEMBER C L))
                      (EQUAL (REMAINDER (TIMES-LIST L) C)
                             0)))
```

EGS generated the following 12 initial examples in 13.9 seconds:

```
             ((C . 4) (L . (6 4)))
             ((C . 4) (L . (2 3 4)))
             ((C . 7) (L . (7)))
             ((C . 3) (L . (11 3)))
             ((C . 3) (L . (3 5)))
             ((C . 2) (L . (5 3 2)))
             ((C . 5) (L . 1))
             ((C . 2) (L . 1))
             ((C . 5) (L . (3 2 2)))
             ((C . 5) (L . (11 3)))
             ((C . 2) (L . (11 3)))
             ((C . 2) (L . (1 . 2)))
```

```
This formula can be simplified, using the abbreviations ZEROP, NOT,
AND, and IMPLIES, to:

      (IMPLIES (AND (NOT (EQUAL C 0))
                    (NUMBERP C)
                    (MEMBER C L))
               (EQUAL (REMAINDER (TIMES-LIST L) C)
                      0)),

which we will name *1.
```

```
     Perhaps we can prove it by induction.  The recursive terms in
the conjecture suggest two inductions.  However, they merge into one
likely candidate induction.  We will induct according to the
following scheme:
      (AND (IMPLIES (NLISTP L) (p L C))
           (IMPLIES (AND (NOT (NLISTP L))
                         (EQUAL C (CAR L)))
                    (p L C))
           (IMPLIES (AND (NOT (NLISTP L))
                         (NOT (EQUAL C (CAR L)))
                         (p (CDR L) C))
                    (p L C))).
Linear arithmetic, the lemmas CDR-LESSEQP and CDR-LESSP, and the
definition of NLISTP establish that the measure (COUNT L) decreases
according to the well-founded relation LESSP in each induction step
of the scheme.  The above induction scheme produces the following
four new goals:

Case 4. (IMPLIES (AND (NLISTP L)
                      (NOT (EQUAL C 0))
                      (NUMBERP C)
                      (MEMBER C L))
                 (EQUAL (REMAINDER (TIMES-LIST L) C)
                        0)).

  This simplifies, expanding the functions NLISTP and MEMBER, to:

        T.

Case 3. (IMPLIES (AND (NOT (NLISTP L))
                      (EQUAL C (CAR L))
                      (NOT (EQUAL C 0))
                      (NUMBERP C)
                      (MEMBER C L))
                 (EQUAL (REMAINDER (TIMES-LIST L) C)
                        0)).

  This simplifies, applying REMAINDER-TIMES, and unfolding the
  definitions of   NLISTP, MEMBER, TIMES-LIST, and EQUAL, to:

        T.

Case 2. (IMPLIES (AND (NOT (NLISTP L))
                      (NOT (EQUAL C (CAR L)))
                      (NOT (MEMBER C (CDR L)))
                      (NOT (EQUAL C 0))
                      (NUMBERP C)
                      (MEMBER C L))
                 (EQUAL (REMAINDER (TIMES-LIST L) C)
                        0)),

  which simplifies, expanding the definitions of NLISTP and MEMBER,
to:

        T.

Case 1. (IMPLIES (AND (NOT (NLISTP L))
```

```
                    (NOT (EQUAL C (CAR L)))
                    (EQUAL (REMAINDER (TIMES-LIST (CDR L)) C)
                           0)
                    (NOT (EQUAL C 0))
                    (NUMBERP C)
                    (MEMBER C L))
               (EQUAL (REMAINDER (TIMES-LIST L) C)
                      0)),
```

which simplifies, expanding the definitions of NLISTP, MEMBER, and TIMES-LIST, to the goal:

```
      (IMPLIES (AND (LISTP L)
                    (NOT (EQUAL C (CAR L)))
                    (EQUAL (REMAINDER (TIMES-LIST (CDR L)) C)
                           0)
                    (NOT (EQUAL C 0))
                    (NUMBERP C)
                    (MEMBER C (CDR L)))
               (EQUAL (REMAINDER (TIMES (CAR L)
                                        (TIMES-LIST (CDR L)))
                                 C)
                      0)).
```

Its associated example set is the same as the initial example set:

```
          (((C . 4) (L . (6 4)))
           ((C . 4) (L . (2 3 4)))
           ((C . 7) (L . (7)))
           ((C . 3) (L . (11 3)))
           ((C . 3) (L . (3 5)))
           ((C . 2) (L . (5 3 2)))
           ((C . 5) (L . 1))
           ((C . 2) (L . 1))
           ((C . 5) (L . (3 2 2)))
           ((C . 5) (L . (11 3)))
           ((C . 2) (L . (11 3)))
           ((C . 2) (L . (1 . 2)))).
```

Appealing to the lemma CAR-CDR-ELIM, we now replace L by (CONS X Z) to eliminate (CAR L) and (CDR L). The result is the goal:

```
      (IMPLIES (AND (NOT (EQUAL C X))
                    (EQUAL (REMAINDER (TIMES-LIST Z) C) 0)
                    (NOT (EQUAL C 0))
                    (NUMBERP C)
                    (MEMBER C Z))
               (EQUAL (REMAINDER (TIMES X (TIMES-LIST Z)) C)
                      0)).
```

The following example set is associated with the above conjecture:

```
          (((C . 4) (X . 6) (Z . (4)))
           ((C . 4) (X . 2) (Z . (3 4)))
           ((C . 7) (X . 7) (Z . NIL))
           ((C . 3) (X . 11) (Z . (3)))
           ((C . 3) (X . 3) (Z . (5)))
           ((C . 2) (X . 5) (Z . (3 2)))
           ((C . 5) (X . 0) (Z . 0))
```

```
          ((C . 2) (X . 0) (Z . 0))
          ((C . 5) (X . 3) (Z . (2 2)))
          ((C . 5) (X . 11) (Z . (3)))
          ((C . 2) (X . 11) (Z . (3)))
          ((C . 2) (X . 1) (Z . 2))).
```

We will try to prove the above formula by generalizing it, replacing (TIMES-LIST Z) by Y. We restrict the new variable by recalling the type restriction lemma noted when TIMES-LIST was introduced. We thus obtain the new goal:

```
      (IMPLIES (AND (NUMBERP Y)
                    (NOT (EQUAL C X))
                    (EQUAL (REMAINDER Y C) 0)
                    (NOT (EQUAL C 0))
                    (NUMBERP C)
                    (MEMBER C Z))
               (EQUAL (REMAINDER (TIMES X Y) C) 0)).
```

The example set associated with the conjecture is:

```
          (((C . 4) (X . 6) (Z . (4)) (Y . 4))
           ((C . 4) (X . 2) (Z . (3 4)) (Y . 12))
           ((C . 7) (X . 7) (Z . NIL) (Y . 1))
           ((C . 3) (X . 11) (Z . (3)) (Y . 3))
           ((C . 3) (X . 3) (Z . (5)) (Y . 5))
           ((C . 2) (X . 5) (Z . (3 2)) (Y . 6))
           ((C . 5) (X . 0) (Z . 0) (Y . 1))
           ((C . 2) (X . 0) (Z . 0) (Y . 1))
           ((C . 5) (X . 3) (Z . (2 2)) (Y . 4))
           ((C . 5) (X . 11) (Z . (3)) (Y . 3))
           ((C . 2) (X . 11) (Z . (3)) (Y . 3))
           ((C . 2) (X . 1) (Z . 2)) (Y . 1)).
```

Applying the lemma REMAINDER-QUOTIENT-ELIM, replace Y by (PLUS V (TIMES C W)) to eliminate (REMAINDER Y C) and (QUOTIENT Y C). We use LESSP-REMAINDER2, the type restriction lemma noted when REMAINDER was introduced, and the type restriction lemma noted when QUOTIENT was introduced to restrict the new variables. We would thus like to prove the new formula:

```
      (IMPLIES (AND (NUMBERP V)
                    (EQUAL (LESSP V C) (NOT (ZEROP C)))
                    (NUMBERP W)
                    (NOT (EQUAL C X))
                    (EQUAL V 0)
                    (NOT (EQUAL C 0))
                    (NUMBERP C)
                    (MEMBER C Z))
               (EQUAL (REMAINDER (TIMES X (PLUS V (TIMES C W)))
                                 C)
                      0)),
```

which further simplifies, rewriting with COMMUTATIVITY-OF-TIMES, COMMUTATIVITY2-OF-TIMES, and REMAINDER-TIMES, and expanding the definitions of NUMBERP, EQUAL, LESSP, ZEROP, NOT, and PLUS, to:

```
      T.
```

```
That finishes the proof of *1.  Q.E.D.

[ 44.5 2.1 (12 13.9) 814 ]
DIVIDES-TIME-LIST
```

The table 5-1 compares two cases where examples are used and where examples are not used for the above theorem. Twelve initial examples are generated in 13.9 seconds by EGS. The first column

Table 5-1: Comparison between Using Examples and Not Using Examples

	Elapsed	I/O	Ex. Gen.	No. of BC
Not Using Examples	92.7(sec)	2.7(sec)	0.0(sec)	2750
Using Examples	44.5	2.1	13.9	814

compares the total elapsed times (this includes example generation time if examples are generated) for proving the theorem. The last column indicates that about 70% of backward chainings are pruned.

5.2 Some Experimental Statistics

In this section we will briefly describe some statistics resulting from our experiment using examples in the Boyer-Moore theorem prover. We have experimented with 488 theorems involving 137 functions[39] and all of them have been successfully proved. Those function definitions and theorems are divided into five "libraries": PROVEALL, RSA, WILSON, GAUSS, and ZTAK. Roughly, each library can be considered to contain a list of events which specify shell definitions, function definitions, and proving lemmas. The following briefly describes each of the above libraries.

1. PROVEALL contains basic definitions and theorems about numbers and lists;
2. RSA includes definitions and theorems about the invertibility of the Rivest-Shamir-Adleman public key encryption algorithm [7];
3. WILSON contains definitions and theorems for Wilson's theorem [45];
4. GAUSS contains definitions and theorems for Gauss's law of quadratic reciprocity;
5. ZTAK includes definitions and theorems for the termination of Takeuchi's function [35].

Table 5-2 shows the statistics of contents in each library; Table 5-3 shows the number of theorems in each library by group; Table 5-4 and Table 5-5 presents, by group, statistics for the ratio of pruned backward chainings and for the percentage of (elapsed) time saved by using examples, respectively. The column labels of A, B, and C in the last three tables indicate those theorems whose proofs take more than 0, 50, and 100 seconds,respectively when proved without using examples. In Tables 5-4 and 5-5 the ZTAK library is not considered because the corresponding lemma base is so small that backward chaining in the theorem prover can be ignored.

Appendix C compares the performance of the theorem prover for the cases that 1) examples are not used and 2) examples are used with the theorems in the RSA library.

[39] They include 7 non-interpretable (not explicit value preserving) functions and 26 theorems involving non-interpretable functions.

Table 5-2: Contents of Libraries

Lib.	No. of Shells	No. of Defs	No. of lemmas
PROVEALL	2	96	267
RSA	0	8	42
WILSON	0	3	42
GAUSS	0	19	120
ZTAK	1	11	17
total	3	137	488

Table 5-3: Theorems in Libraries

Lib.	A	B	C
PROVEALL	267	16	9
RSA	42	14	4
WILSON	42	16	9
GAUSS	120	51	36
ZTAK	17	0	0

Table 5-4: Backward Chaining Cut-Off Ratio

Lib.	A	B	C
PROVEALL	0.54	0.26	0.20
RSA	0.29	0.45	0.71
WILSON	0.29	0.25	0.28
GAUSS	0.32	0.21	0.21
ZTAK	-	-	-

Table 5-5: Time Saved by Using Examples

Lib.	A	B	C
PROVEALL	-	0.22	0.19
RSA	-	0.29	0.55
WILSON	-	-0.04	0.03
GAUSS	-	-0.02	0.02
ZTAK	-	-	-

5.3 Some Observations

From this experiment several interesting observations can be made. One important observation is that in the Boyer-Moore theorem prover the effective use of examples is often prohibited. The primary reason is because of the structure of the theorem prover, especially its proof by induction. When the theorem prover simplifies a clause corresponding to a conjecture it attempts to simplify each literal in the clause, assuming that all the other literals are false. When examples are used we select from the associated example set those examples which falsify all the literals except the one being simplified. However, it is often the case that there are some redundant or irrelevant literals in the clause and the clause appears to be true because of literals other than those redundant or irrelevant ones. Suppose we have a conjecture of the form

(IMPLIES (AND h_1 h_2 ... h_i h_j ... h_n) c).

Such a conjecture is often (trivially) proved to be true because the conjunction of a few h_i's are false. The conjecture is represented in clausal form as

((NOT h_1) (NOT h_2) ... (NOT h_i) (NOT h_j) ... (NOT h_n) c).

Suppose that the whole conjecture is true because (AND h_i h_j) is false. In trying to prove the conjecture, we first simplify the literal (NOT h_1) - assume neither i nor j is equal to 1 - under the assumption that (AND h_2 ... h_i h_j ... h_n (NOT c)) is true. This assumption is actually false because (AND h_i h_j) is false. Therefore, there is no example satisfying the assumption to be used in simplifying the literal. This situation blocks examples from being used effectively. This is the very reason that our strategy of generating examples as often as necessary, such as when no successful earlier examples are found, is not acceptable. Attempts to generate examples for this case fail at significant cost. In the Boyer-Moore theorem prover, clauses containing redundant and irrelevant literals are often introduced by the induction routine, especially as base cases of induction proofs.

The following are some other observations which can be made about our experiment.

1. The effectiveness of examples for theorems are largely unpredictable and the effectiveness of pruning backward chaining is uneven over the theorems tested.
2. EGS sometimes generates inappropriate examples which cause computation of exceptionally big numbers; it costs significant resources to evaluate some terms with those inappropriate examples[40].
3. Useful examples are generally expensive, and EGS tends to generate cheaper examples first.
4. Initial examples become less useful as theorem proving proceeds because formulas are transformed and the examples do not correspond to the new formulas. Also, when examples are computed for newly introduced variables, they often get degenerated.
5. The proofs of some difficult theorems are well guided by the user with hints. Such guidance deprives examples of chances being used.
6. Evenly distributed examples - not only typical examples but boundary examples as well - are important.

A limited experiment has indicated that the user's interaction would improve the effectiveness of using

[40]For instance, the evaluation of the term (PRIME (EXP X (TIMES U V))) with respect to (QUOTE 3), (QUOTE 4), and (QUOTE 5) for X, U, and V, respectively, easily overflows the stack of the LISP machine.

examples. In a proof session when examples are being generated the user can suggest his own examples, give hints by adding certain conditions to the constraints, and allocate resources, such as time, appropriately.

Our method of using examples is sound in the sense that it does not cause any proof of a non-theorem. However, it may not be complete in the sense that it possibly causes the failure of proof of a theorem that is proved without using examples. Each time examples are used to simplify a literal, those examples must satisfy the assumption under which the literal is simplified. Examples not satisfying the assumption possibly prevent useful lemmas from being applied in simplifying the literal, thus possibly causing the failure of the proof. We have experimented with a couple of different strategies for computing "selected examples", such as testing each example in the associated example set against a proper subset of the clause as opposed to the entire clause. The results have shown that for some cases the effectiveness has improved but often the proof attempts have failed. We have experimented with the idea of rearranging the literals of a clause. In simplification more plausible literals (they can be identified by testing with examples) are processed first. A slight increase in efficiency has been observed.

To summarize, the result of our experiment is negative. We have found that the effectiveness of examples in controlling backward chaining is reduced by irrelevant literals in the conjectures. Such irrelevant literals are often introduced by the induction strategy of the theorem prover. It is questionable whether using examples is suitable for an induction-based theorem prover to control backward chaining.

6. EGS: Weaknesses and Limitations

EGS is still in an experimental stage and the current implementation has exhibited several weaknesses. In this chapter we first describe some characteristics of EGS briefly, then we discuss its weaknesses and limitations. Improvements on some weak points will be discussed in chapter 8.

6.1 Some Characteristics of EGS

EGS is nondeterministic in that it can possibly generate different sets of examples for the same constraint. The reason we designed EGS with nondeterminism is twofold:

1. to model the nondeterminism of human example generation;
2. to avoid biased generation of examples.

The nondeterministic behavior of EGS is mainly due to the strategy of testing stored examples in random order.

EGS supports the extensibility of the underlying Boyer-Moore logic. When a new shell type or function is defined, EGS still can generate examples of constraints involving the shell type or function without user-given knowledge such as known examples. In such cases examples are generated by appealing to more basic knowledge such as definition and the free examples[41].

EGS provides some kinds of controllability.

1. The user can control the behavior of the system by modifying certain parameter values of the plausibility score algorithm. For instance, by appropriately assigning weights to task operators, the user can prevent certain types of tasks, such as EXPAND tasks,from being generated.
2. The ratio of the typical examples and boundary examples generated can be controlled[42]. It is approximately the ratio at which we select the successful stored examples of each type when a TEST task is performed. For instance, in default we select two typical examples and one boundary example. The number of examples we select for each type when stored examples are tested also determine the branching factor of the search space for example generation and affect the population of generated examples.

6.2 Weaknesses of EGS

6.2.1 No analytic reasoning

In EGS, no attempts are made to evaluate how closely candidate examples satisfy the current constraint to identify the cause of failure when some candidate examples fail to satisfy the constraint. Such analytic reasoning would allow EGS to generate examples by modifying unsuccessful candidate examples to satisfy the constraint. The strategy of example modification based on information of such analytic reasoning is desirable for efficient example generation. EGS currently generates examples by expanding definitions when no stored examples succeed; it is generally inefficient and EGS easily gets lost when the definitions are complex.

---

[41] Refer to "Performing CONSTRUCT task".

[42] Criteria for classifying the generated examples into typical examples and boundary examples are not well defined.

6.2.2 Shallowness

EGS tends to generate simple examples first, largely because of its plausibility score computation method and its definition expansion strategy. Simplification after a function definition is expanded in a constraint splits the constraint into subproblem constraints corresponding to the base cases and recursion cases. Tasks of the subproblem constraints corresponding to the base cases are often given higher plausibility scores than tasks corresponding to the recursion cases because the constraints of the base cases are simpler than the constraints of the recursion cases. Examples generated for the base cases are generally boundary examples. Rich typical stored examples would solve this problem to some degree.

6.2.3 Lack of user interfaces

The current implementation of EGS lacks various user-interfaces:

- explanation and trace capabilities;
- interactive facility with which the user can examine the intermediate steps of the example generation process and control the process;
- tools for building and maintaining the knowledge base;
- multi-window graphic display.

6.2.4 Discontinuity in performance

A desirable characteristic of any system is a continuity in the system's performance; it allows one to predict the performance of the system for a certain type of problems. EGS should perform with similar efficiency for similar constraints. However, EGS may perform considerably differently even for similar constraints. It is largely because of EGS's strategy of testing stored examples and its lack of example modification. If EGS has stored examples satisfying a certain constraint, then EGS quickly generates examples. However, if another constraint is slightly different but no stored examples satisfy the constraint, EGS may take much longer to generate examples by applying other strategies such as definition expansion. Such discontinuity of performance can also arise when several examples are generated for a single constraint. The first few examples are generated quickly, but the others take much longer.

6.2.5 The Constructiveness of the Boyer-Moore Logic

The constructive nature of the underlying Boyer-Moore logic allows us to have an efficient interpreter. However, its constructiveness sometimes works against efficient example generation in EGS. In the Boyer-Moore logic functions are generally defined recursively; such definitions are just like programs and are possibly much more complicated[43]. In EGS, function definitions are one of the basic forms of knowledge and they are applied when applications of other heuristic and ad hoc knowledge fail. When a function definition is expanded, the procedural nature of the definition often causes difficulties to reason with it. Consider the constraint (EQUAL (REMAIDER X Y) 5). Suppose we expand the definition of REMAINDER and we have the following non-trivial case

```
(AND (NOT (ZEROP Y))
```

[43]Consider that a program is much more complicated than its corresponding specification.

```
(NOT (LESSP X Y))
(EQUAL (REMAINDER (DIFFERENCE X Y) Y) 5)).
```

We may need to repeat the expansion of the definition of REMAINDER several times to get non-trivial examples of X and Y. Consider the conventional definition (in the first-order number theory) of remainder.

```
remainder(x, y) = r if there exist numbers q and r such that
                       r < y and x = q*y + r.
```

With this definition, given r = 5, we can easily compute the values of q and y, thus that of x.

6.3 Limitations of EGS

6.3.1 Incompleteness

Theoretically, the example generation problem in the Boyer-Moore logic is undecidable[44]. No claim can be made about the completeness of EGS either. The incompleteness of the EGS example generation causes some difficulties in the applications of examples, especially, where efficiency is critical. If a given constraint is not satisfiable, the cost of EGS to attempt to generate examples for the constraint would be wasted. Therefore, applications of examples must be carefully chosen so that constraints for example generation are generally satisfiable and not very restrictive. It may be true that for a specialized application one can possibly develop a "complete" example generation system. In such cases, by employing an appropriate internal problem representation technique and a powerful reasoning capability specific to the application, the example generation system also can be implemented to be efficient. However, for our example generation the problem domain and types of problems to be handled are so general that one can hardly expect such a complete system.

6.3.2 Knowledge Representation

Our knowledge based approach to example generation has some drawbacks. One of them is that it is difficult to represent some useful knowledge. Some knowledge is better represented in procedural form than in declarative form. Consider the linear equation/inequality solver. Various kinds of knowledge about natural numbers and linear arithmetic operations are integrated to be under good control to achieve efficiency. If such kinds of knowledge are represented separately, it would be difficult to make them to work in a well organized way. For our linear solver separate applications of the commutativity law, associativity law, and cancellation law of PLUS with linear equations or inequalities would not work. It is generally the case that pieces of knowledge that are working collectively are coded into a procedure. Also, we often find that the recursive definitions of functions are difficult to reason with. Human understanding of concepts seems to be much more flexible. For instance, a human can reverse a list consisting of an indefinite number of elements such as (A B ...) and still can reason with the reverse of it to see that A is the last element of the reverse. Such flexible representations of concepts are desirable because they cause the reduction of search space in example generation. The current knowledge representation scheme does not support representation of such flexible representations of concepts and objects.

[44]This issue is discussed in the appendix.

6.3.3 Inability with restrictive constraints

Even though EGS employs some strategies such as definition expansion and lemma based simplification to generate examples for restrictive constraints, it has exhibited frequent failures and significant reduction in efficiency in generating examples for such constraints. One of the assumptions under which EGS is designed is that examples are generated for not very restricting constraints; it is assumed that there are many examples satisfying any given constraint. Strategies and heuristics are employed by EGS to generate several examples efficiently with many possibilities, as opposed to generating a few difficult examples.

7. Related Work

Various attempts have been made to generate and apply examples. Examples have long been used for many AI tasks, especially for automated reasoning [19, 20, 2, 40] and machine learning [33, 10, 52, 32, 34]. They are also used in the areas of intelligent computer-assisted instruction and tutoring and intelligent human interfaces [54, 41]. However, in many cases examples are given by the user interactively or stored initially in the system. In the introductory chapter we described Gelernter's geometry machine[45]. In this chapter we describe some other related work on generating and using examples.

7.1 Lenat's AM

AM [33] automatically discovers concepts and conjectures in elementary set theory and number theory by being guided by various heuristic rules. Its initial knowledge base includes primitive set-theoretic concepts such as "sets" and "bags", operations such as "set-union", "set-intersection", and Cartesian product (and analogous operations for bags) and various heuristic rules for discovering new concepts and conjectures. Based on the knowledge AM discovers interesting mathematical concepts and conjectures. In AM, examples play an important role. When a new concept is created - in the sense that its definition in terms of LISP function is determined[46] - a task for filling the "EXAMPLES" slot of the concept is generated and put on the task agenda. When the task is being performed, heuristic rules such as

```
If the current task is to fill in examples of the concept C
and C is a kind of D (for some more general concept D), check
the examples of D; some of them may be examples of C as well.
```

tell how to generate examples for the concept.

The following are some important heuristics employed by AM for example generation.

1. *Instantiation of definitions:* the base case of the recursive LISP definition of a concept can be instantiated to produce boundary examples and the recursion steps can be further instantiated with those boundary examples to create normal positive examples. For instance, the LISP-like definition of the concept "set" is:

   ```
   (set s) =
     (or (= s {})
         (set (remove (any.member s) s))).
   ```

   A boundary example {} is produced from the base case of the definition. By further reasoning with the definition that a set is created by "inserting" (the inverse of "remove") any example into a known example of a set, more examples like {{}}, {a}, and {1 2 3} can be generated.

2. *Testing known examples:* Examples of a concept C are generated by testing the examples of its relative concepts such as more general concepts than C, concepts that share a common generalization or specialization with C, and operations having C in their range.

3. *Inheritance of examples:* Positive examples of more specialized concepts can be inherited up to a more generalized concept. Similarly, negative examples can be inherited down the generalization hierarchy of concepts.

---

[45] Gilmore [21] and Bundy [11] have discussed a rational reconstruction of the geometry machine.

[46] In AM a concept is represented as a frame, collection of slot and value pairs.

Those generated examples are used further to develop new concepts and conjectures. Known examples are used to generate examples for new concepts. The use of examples is also guided by heuristic rules. Some such heuristic rules are:

```
If some (but not most) examples of C are also examples of
D (for some Concept D), create a new concept defined
as the intersection of those two concepts C and D.

If very few examples of C are found,
then add the following task to the agenda:
"generalize the concept C".
```

If a concept has many exceptions (negative boundary examples), a new concept can be created whose instances are those negative examples. AM also can create a concept whose instances are those positive examples but not boundary examples. Also, conjectures are discovered by performing heuristically guided examination of examples. Conjectures take one of the following forms:

1. A concept C_1 is an example of another concept C_2;
2. A concept C_1 is a specialization or generalization of another concept C_2;
3. A concept C_1 is equivalent to another concept C_2;
4. A concept C_1 is related to another concept C_2 by a certain predicate P;
5. An operation C has a certain domain D or range R.

For instance, if all examples of a concept C_1 are also examples of a concept C_2, then AM conjectures that C_1 is a specialization of C_2. If a concept has no negative examples, it conjectures that the concept is trivially true. If all examples in the range of a concept C_1 are also examples of a concept C_2, AM conjectures that the range of C_1 is C_2.

AM provides valuable insights into the discovery of concepts and conjectures using examples. It also clearly demonstrates that heuristic search is a model of a scientific theory formation.

Although the domain of AM is analogous to that of EGS and some similar methods are used, their approaches to example generation are significantly different. One of the important differences is that example generation in AM is concept-oriented while example generation in EGS is constraint-oriented. In AM examples are generated largely by simple manipulation of known examples and such manipulation is guided by heuristic rules. However, in EGS example generation often requires that multiple conjunctive conditions in a constraint be resolved. In EGS such resolution is achieved by a formal reasoning - definition expansion and lemma-driven simplification. Lacking formality in its problem domain and a formal reasoning, AM may have significant difficulties in generating examples even for moderately restricting constraints.

7.2 CEG

CEG (Constrained Example Generation) [43] generates examples in the domain of LISP data and programs that meet specified constraints. In CEG an example generation problem is specified by a list of pairs of desired properties and desired values. For instance, the problem to find an example of a list of length 3 where the depth of the first element is 1 can be represented as

```
Constraint-1  DESIRED-PROP: (TYPE X)
              DESIRED-VALUE: LIST,
```

Constraint-2 **DESIRED-PROP: (LENGTH X)**
DESIRED-VALUE: 3, and

Constraint-3 **DESIRED-PROP: (DEPTH (FIRST-ELT X))**
DESIRED-VALUE: 1.

The major knowledge base of CEG consists of an examples-space and difference-operator table similar to that of GPS [15]. The examples-space consists of known examples - LISP data structures - organized according to the relation of "constructional derivation" representing which examples are constructed from which others. The CEG example generation process consists of three major phases: retrieval of known examples, modification of known examples, and construction of examples from general principles and model examples. Given a constraint in the form of a list of desired property-value pairs, CEG first searches through the examples-space attempting to find any known examples satisfying the constraint. If this retrieval phase fails, CEG invokes the modification phase trying to modify "close" examples; examples are rated according to how well they satisfy the constraint. For modifying an example, the desired properties and the difference between the desired value and the actual values of the example are identified. Operators appropriate for reducing the differences are then obtained from the difference-operator table and executed with the example. The construction phase is called for to construct examples by instantiating model examples or templates when the first two phases fail.

CEG has adopted a general paradigm for example generation, namely, GPS-like difference reduction. However, its constraint specification power is limited. Many interesting constraints may not be expressed in the form of property-value pairs. For instance, the value parts may not be in the form of explicit constants. Also, its means-ends analysis method for example modification may not be suitable for cases where moderate expressive power is required for constraint specification. It is generally difficult to compute appropriate differences between the current state and the goal state and to identify operators suitable for reducing the differences if constraints are complicated. CEG also differs from EGS in that its problem domain is not formally specified and its knowledge is still at the suffice-level.

Rissland also discusses mathematical knowledge as a structure of "concept space", "example space", and "result space". She also groups mathematical examples into *epistemological classes* - start-up examples, reference examples, model examples, and counter-examples each of which plays a different role in understanding mathematics. The interested reader should refer to Rissland [42].

7.3 GRAPHER

Ballantyne and Bledsoe's GRAPHER [2] generates counter-examples for non-trivial topological conjectures within the domain of finite point sets. Given a conjecture, GRAPHER constructs a set-theoretic relationship graph which serves as a global constraint specification. GRAPHER employs its fairly extensive knowledge in set theory and topology in constructing the graph. The constraint graph represents the subset and membership relationships among the set objects mentioned in the conjecture. Also, the "openess" or "closedness" of each set object is incorporated into the graph. After the constraint graph is constructed, an attempt is made to assign values to the set objects. GRAPHER starts assigning values at the bottom - the smallest set objects - of the graph and proceeds upward. When a value is assigned, the smallest possible value is assigned. Each time a value assignment has been made, the value is substituted for the corresponding set object occurring in the graph and all possible reductions are performed, bringing to bear considerable knowledge about elementary set theory and topology. If an assignment causes a contradiction, GRAPHER backs up to the last assignment and try to add another

point into the old value. If the program cannot add a new point, it backs up to the next choice point and tries again. The alternation of value assignment and reduction continues until all set objects in the graph are instantiated. Some domain-specific heuristics are employed in modifying values to assign and traversing the graph for value assignment. In GRAPHER the construction of constraint graph and heuristics are are problem-specific and the problem domain is substantially restricted. However, GRAPHER models some aspects of human example generation in a non-trivial domain by exploiting extensive domain-specific knowledge. Example generation in GRAPHER is of higher-order in the sense that the domain of examples involves sets.

Ballantyne and Bledsoe have also experimented with constructing counter-examples and using them as a subgoal filter in proof discovery in analysis [2]. In the experiment linear examples (linear or piecewise linear functions) for function variables are automatically constructed and used as counter-examples in actual proofs [4]. They have discussed that examples can also be used positively - examples are used in conjecturing lemmas which can be proved and which, in turn, are used in establishing the proof of the main theorem.

7.4 REF-ARF

REF-ARF [16] was designed as a problem solving system such as GPS; its problem solving method is constraint satisfaction. REF is a language for stating problems, which is an extension of an ALGOL-like syntax. ARF, the REF interpreter, translates the REF problem statement into the form of "contexts", internal problem representations in REF-ARF. The interpreter as a constraint satisfier processes a context by alternating assigning a value to a variable and manipulating constraints. When a variable is assigned a value, the constraints are manipulated to reflect the value assignment. If all the constraints have been processed without finding a solution, another variable is selected and assigned a value, then the constraints are manipulated again. This alternation between value assignment and constraint manipulation continues until a solution is found or all possibilities of value assignments to variables have been considered. Each time a variable is assigned a value, the most restricting variable is selected. Similarly, the most restricting constraint is manipulated first. Various kinds of heuristic information for determining the most restrictive constraints and variables are used. They include the number of variables occurring in the constraints, the primary operators of the constraints, and the size of the range of variables. When a context is processed, the "closest" one is selected.

REF-ARF has solved many classical problems such as a magic square problem, the water jug problem, a crypt-arithmetic problem, etc. It also has provided some useful insights for designing EGS. However, the problem solving in REF-ARF takes advantage of the fact that many of problems fit well within the domain of integers and scalar data types for which the control structure and heuristics of REF-ARF have been designed.

7.5 Test Generation

Hardware (logic circuits) and software are often validated by testing. In the testing method test patterns or test data are (automatically or manually) generated to be used in locating faults in hardware or software.

D-algorithm [44] has been developed and used for test generation for logic circuits. In this algorithm computation of a test for a fault proceeds in two stages: error propagation and line justification. Error

propagation is done by selecting a path from the site of the fault leading to an output of the circuit and determining the inputs to the gates along the path so as to propagate the fault signal to the primary output of the circuit. Line justification is done by tracing backward from the inputs to the gates determined during the error propagation stage to the primary inputs of the circuit. Test generation in D-algorithm can be viewed as search. Recently, more efficient algorithms such as PODEM [22] and FAN [17] have been developed. These test generation methods are error-based in the sense that test patterns are generated which are designed to uncover a certain specific type of faults such as "stuck-at" faults.

Different methods have been used for test data generation for software. In structural testing test data are constructed that result in the execution of components of a program. Branch testing, for example, requires that test data be constructed which result in the execution of every branch of the program at least once. In general test data generation in structural testing consists of multi-phases [27]. First, a given program is analyzed to construct description of standard classes of paths through the program. Then, (implicit) descriptions of the sets of input data are constructed which cause the different standard classes of paths to be followed. Implicit descriptions consist of the assignments, loops, function calls, and branch predicates which characterize the data causing the paths to be followed. Implicit descriptions are further transformed into explicit descriptions in predicates and relations. Finally, test data are generated which satisfy explicit descriptions.

Error-based testing involves the construction of test data which are designed to locate specific errors or classes of errors. Mutation testing is an error-based testing method [28]. In the method a given program is modified by applying a certain mutation transformation to a simple component of the program. Mutation transformations may include changing variable names, altering coefficients in arithmetic expressions, and altering arithmetic relations and Boolean expressions. Test data are constructed so that for each of test data the original program (or a component) and its mutant give different outputs. Mutation testing is based on the "coupling-effect" principle[47].

Attempts have been made to systematically identify error-prone constructs and associated "revealing" test data [23]. Test data generation combining the path-based method and "error-revealing subdomains" has also been proposed [51].

7.6 Other Related Work

The *regression* method for plan modification [49], can be useful, especially for constraint-based example generation. For any goal G and action (or plan step) A, there is no guarantee that G will be achieved after A, but a new goal G' can be found such that if G' holds before A, G will hold after A. Finding such a new goal G' is called goal regression. In order to achieve several conjunctive goals, the plan modification method constructs a plan by solving a single conjunctive goal, then modifying the plan by regressing subsequent goals from the end of the plan to a point in the plan where the achievement of the new goal will not violate those goals that have previously been achieved. Goal regression can be used to guarantee that goals that have been achieved are not violated by subsequent actions. During the plan modification when a proposed action that achieves a new goal causes a violation of a previously achieved goal, an attempt is made to insert the action at earlier point in the plan, checking to see whether the interaction is avoided and to see that no new protection violations occur. The regression method

[47]Tests designed to detect simple kinds of errors are also effective in detecting much more complicated errors.

attempts to achieve multiple interacting goals by resolving them at the lower-level descriptions of problems. It is analogous to definition expansion and simplification as constraint refinement schemes in EGS.

Brotz and Aubin have used examples in a limited way to check the non-provability of generalized conjectures in their theorem provers. In his arithmetic prover Brotz used the assignments of 0's, 1's, or randomly chosen numbers to variables. A generalized formula is evaluated by an interpreter with respect to such assignments; if any assignment causes the formula to evaluate to false, the corresponding generalization must be abandoned. Aubin's theorem prover generates *structures* systematically and uses them as examples. A structure of a type is a term consisting of the constructor function symbol of the type and variables. Certain levels of structures of a type are generated once and for all when the type is defined, and they are used for checking the non-provability of conjectures by simplifying the conjectures with variables replaced by structures.

In LTP [12] examples are in general terms which satisfy a collection of required properties. For example, the term 2f(x) is an example of even numbers. LTP uses examples for pruning the proof search space; its example evaluation is based on simplification. Reiter [40] uses examples as a semantic guide for efficient theorem proving in a formal natural deductive logic. In his system, examples are given by the user and are used to cut off implausible reasoning paths in theorem proving. Wang [50] has experimented with using examples in a resolution-based theorem prover and has proved some difficult theorems. In Wang's system examples are designed by the user and they are used to semantically guide the proof search. The set of support strategy [53] can be considered as an attempt to to use examples (or models) in resolution theorem proving for efficient search.

QA3 [25], deductive database systems [18], and the PROLOG interpreter [48, 37] can answer questions specified in logical formulas. The underlying reasoning mechanism is resolution-based theorem proving. The derivation of answers to questions can be regarded as example generation. Such deductive systems do not attempt to computationally implement the semantics of the underlying theories. They employ pattern matching (unification) for example evaluation as opposed to interpretation. The reasoning mechanism in the deductive systems is analogous to the combination of the definition expansion and testing methods in EGS. However, the deductive systems lack the capability of applying domain-specific procedural knowledge - such knowledge is in the form of procedures and is often heuristically justified - to enhance efficiency as EGS does.

Hardy [26] and Summers [47] have investigated automatic construction of LISP programs from examples.

8. Future Research

In chapter 6 we pointed out several weaknesses of EGS. In this chapter we discuss possible improvements on some of the weaknesses of our future work. Example generation in EGS can be viewed as search. Many weaknesses of EGS are largely due to the vast search space. Thus, the proposed improvements are methods for reducing the search space of example generation. We also propose potential applications of EGS in the Boyer-Moore theorem prover.

8.1 Further Improvements in Example Generation

8.1.1 Analytic Reasoning and Example Modification

A significant drawback to the current implementation of EGS is that it lacks example modification based on analytic reasoning. When we test known examples against a constraint we can possibly analyze the failure cases to find how closely the failed examples satisfy the constraint and why they fail. Based on the analytic information, we can modify the failed known examples to satisfy the constraint. It would not be desirable to throw away those approximate known examples simply because they do not satisfy the constraint completely. Human example generation seems to employ the example modification scheme effectively. Consider a simple constraint

```
((MEMBER X L) (MEMBER (QUOTE M) L).
```

Suppose we test a known example ((QUOTE A) (QUOTE (C A B))) of MEMBER where X and L are assigned values of (QUOTE A) and (QUOTE (C A B)), respectively. The known example satisfies the first condition but fails with the second condition because the value assigned to L does not have (QUOTE M) as its member. The current implementation of EGS would withdraw the known example as a failure. However, we can modify the value assigned to L to satisfy the second condition by inserting (QUOTE M) into (QUOTE (C A B)). Such modification can be done by applying a (maybe hand-coded) procedure implementing the insertion.

Example modification may require a large body of relevant knowledge to bring to bear and it is, in many cases, difficult to determine the actual cause of the failure and how to correct it. Also, such modification may cause the modified example to violate conditions which were originally satisfied. However, a simple form of example modification can be incorporated into EGS with relative ease. For instance, we could allow the modification scheme to be called only when very few conditions are left in the constraint. Such cases largely occur when the system is close to successfully finding examples and the remaining constraints are relatively restrictive - many of the variables are already instantiated with constant values and it is less likely that known examples completely satisfy the constraints. When certain known examples are tested against a constraint but none of them succeed, the modification scheme is invoked (or a new task MODIFY is generated) to apply the corresponding procedure to modify some "near miss" examples to become successful ones. The procedural form of knowledge for modification can be represented in the form that is similar to that of equation solvers.

8.1.2 Implicit Information

During example generation information can be derived from the constraint and such information can be used effectively as additional constraints. Suppose we have the following condition in a constraint:

```
(EQUAL (REMAINDER X Z) Y).
```

From this condition we can derive an implicit condition that

(LESSP Y Z) if **(NOT (ZEROP Z)).**

This kind of implicit information is important because it allows us to prevent unsatisfiable or undesirable subproblem constraints from being generated. For the above example, any attempts to instantiate the variables Y and Z in a way violating the implicit condition should be rejected. However, it may be the case that such implicit conditions should not explicitly be added into the constraint because if added, they may be removed as subsumed when the constraint is simplified. In GRAPHER [2] implicit information about set objects is contained in the graph representation of the global constraint. It is used as additional constraint when variables corresponding to set objects are instantiated during example generation for topological conjectures. In EGS we may have a certain form of knowledge which can produce such implicit information associated with functions or certain patterns of formulas . Implicit information is associated with each task and can be inherited down to the derived tasks of a task.

8.1.3 Prioritizing Conditions

One way of reducing search space for example generation is that we prioritize conditions in a constraint in terms of their restrictiveness and within a constraint we resolve the most restricting condition first. Consider the constraint:

```
((EQUAL (LENGTH L) 3) (NUMBERP (CAR L)) (LISTP (CADR L))).
```

The first condition is most restricting and it should be solved first. After the first condition is solved to rewrite into

```
(AND (EQUAL L (CONS X (CONS Y (CONS U V))))
     (NOT (LISTP V))),
```

the rest of the conditions can be solved relatively easily without expanding the search space much. The restrictiveness of a condition can be heuristically determined in a way similar to computing the syntax score of a constraint when computing the plausibility score. Important factors to be considered are the restrictiveness of functions involved, occurrences of variables and constants, and syntactical patterns of the condition formula. The restrictiveness of functions can be determined analytically from their definitions or it can be provided by the user. Multiple occurrences of the same variables in a condition or conditions indicate the high restrictiveness of the condition. Also, the syntactical pattern of a condition affects the restrictiveness. For example, a condition of the form (EQUAL lhs rhs) is generally more restrictive than a condition of the form (NOT (EQUAL lhs rhs)). It is important that when we process the most restricting condition first, we need to find the most general (partial) solution to the condition. The general solution allows us to reduce the search space substantially[48]. We will discuss more of the general solution issue in the next section.

8.1.4 Flexible Concept Representation

In example generation, it is important to keep a constraint in as general a form as possible because the general form subsumes multiple specific instances, thus allowing us to reduce the search space. To keep a constraint in general form it may be necessary to represent the involving concepts (functions) in more flexible form. Consider a human's understanding of the concept REVERSE; his understanding would be:

---

[48]This is just like the strategy of finding the most general unifier in resolution theorem proving.

```
given a list, rearrange each element of the list such that
the first element becomes the last element, the second element
becomes the next last element and so on, and finally the last
element element becomes the first element in the resulting list.
```

In other words, he understands the function in terms of more flexible structures and operations. Given a (pseudo) list of the form (A B ... D) - a list consisting of A, B, and D as the first, second, and last elements, respectively, and a variable number of elements inbetween - a human can still compute the REVERSE of the list to be (D ... B A). Furthermore, he can even handle arbitrary variable elements in such representation of a list. As an interpreter, he evaluates REVERSE of (A x ... y B) to (B y ... x A) where x and y denote variable elements. Such flexible representation of objects and concepts allows us to keep a constraint in its general form and delay the value assignment to variables until needed. Suppose we have the constraint

```
((EQUAL (LENGTH L) 3)
 (MEMBER X L)
 (MEMBER Y L)
 (NOT (EQUAL X Y))
 (EQUAL (REVERSE L) L)).
```

The most general solution to the first condition would be that L is assigned (x y u . v) with v a non-list, where x, y, u, and v are variables. When we evaluate the last condition with this assignment, we have that (x y u . v) is equal to (u y x). We also find that x and y are equal to u and z, respectively, and v must be NIL. So far we have still one single equivalent constraint

```
((EQUAL L (CONS X (CONS Y (CONS X NIL))))
 (MEMBER X L)
 (MEMBER Y L)
 (NOT (EQUAL X Y))).
```

Currently EGS, to some extent, has the capability of such computation. (REVERSE (CONS X (CONS Y (CONS U NIL)))) will be rewritten to (CONS U (CONS Y (CONS X))). However, such transformation is based on simplification as opposed to interpretation, and it is inefficient.

Reinterpretation of complicated functions in terms of more flexible "quasi" objects and "quasi" operations would be useful for example generation in EGS. For example, the quasi REVERSE would operate on an object such as (A x y B) to produce (B y x A) where x and y are variables. The flexible reinterpretation of functions, together with prioritizing conditions in example generation, can be well incorporated into the constraint graph, a semantic net-like representation of a constraint. EGS can combine such different representation schemes - logical formula and constraint graph - to represent a constraint appropriately. Ballantyne and Bledsoe's GRAPHER employs a graph structure which represents the global constraint in terms of set-theoretic relationships among set objects. The graph structure is used as a global constraint specification when the set objects are instantiated.

EGS as an experimental system for automatic example generation has raised many problems to investigate. The above are some short-term problems to remedy. Several other long-term improvements could be made. They include

- incorporating machine learning capability;
- building a user-friendly interface to EGS;
- devising a tool for building and maintaining the knowledge base;

- implementing interactive example generation.

8.2 Example Applications

The result of our experiment of using examples to control the backward chaining problem in the Boyer-Moore theorem prover was evaluated to be negative. More study should be done for a more effective method of using examples for such a problem in the theorem prover. However, there are several other potential applications of examples in the theorem prover. In this section we will propose such applications.

8.2.1 Conjecture Checking

Conjectures can be shown to be invalid by counter-examples. This makes counter-examples an efficient way to demonstrate the invalidity of a conjecture or hypotheses. To prove the validity of a conjecture, on the other hand, requires a rigorous chain of arguments from the axioms or previously proved facts. Humans seem especially good at using counter examples.

EGS generates counter-examples for conjectures in the Boyer-Moore theorem prover. When the Boyer-Moore theorem prover attempts to prove a non-theorem, it may either terminate with failure or run forever. Even when the theorem prover terminates with failure, it is not determined whether or not the given conjecture is valid. In such cases the theorem prover calls the example generator to find counter-examples for the conjecture. If counter-examples are found we can conclude that the conjecture is not valid. Such counter-examples can be analyzed to correct the invalid conjecture; counter-examples are effectively used to determine why the given conjecture is not valid. For example, suppose we have a conjecture that

```
(EQUAL (SUBST X Y (SUBST X Y Z))
       (SUBST X Y Z)),
```

where SUBST is defined as

```
(SUBST X Y Z) =
  (IF (EQUAL Y Z)
      X
      (IF (LISTP Z)
          (CONS (SUBST X Y (CAR Z))
                (SUBST X Y (CDR Z)))
          Z)).
```

At first sight, the above conjecture seems very reasonable. However, the theorem prover fails to prove the conjecture. The question is: "Is the failure because of the inability of the theorem prover or because of the invalidity of the conjecture?" Now EGS can be called to generate some counter-examples. They would be:

```
((X . (QUOTE (A)))
 (Y . (QUOTE A))
 (Z . (QUOTE (A B))))
```

and several others. With this counter example one can clearly see why the conjecture is not valid. One can now add a new condition, namely, (NOT (OCCUR Y X)); to eliminate the case that caused the conjecture to fail. We obtain the new conjecture

```
(IMPLIES (NOT (OCCUR Y X))
```

```
(EQUAL (SUBST X Y (SUBST X Y Z))
       (SUBST X Y Z))),
```

which can be proved correct.

The sequence of hypothesizing conjectures, modifying and verifying them is an essential part of theory development. [31, 38, 39] Constructing correct conjectures is difficult in general. It is largely because a human's knowledge that is related to the problem may be incorrect or incomplete and he might often miss the special but critical cases in building a conjecture. Using examples for checking and correcting conjectures is a very challenging problem.

In theorem proving, some conjectures are generalized to be proved. Examples can be used to check the non-provability of the generalized conjectures. If any counter examples are found for a given generalized conjecture, the conjecture is not valid, and thus cannot be proved; if no counter-examples are found, it is more likely that the conjecture is valid. In the Boyer-Moore theorem prover, EGS can be called to generate counter-examples for a generalized conjecture to check its invalidity. Brotz and Aubin have used examples in checking the non-provability of generalized conjectures in their theorem prover.

8.2.2 Hypothesizing

Automatic generation of interesting conjectures or concepts from examples would be a challenging problem. Many machine learning systems use examples for such purposes. Common patterns or properties shared among examples are often developed into new concepts and conjectures. Consider some counter-examples of the conjecture

```
(EQUAL (REVERSE (REVERSE L)) L).
```

They might be:

```
(L . (QUOTE A))
(L . (QUOTE 0))
(L . (QUOTE (A . A)))
(L . (QUOTE (C B . A)))
(L . (QUOTE (3 7 . 3)))
(L . (QUOTE (B 1 2 . 3)))
(L . (QUOTE ((E) B D . B)))
(L . (QUOTE (D 5 (3 8) . E))).
```

Now from these counter-examples a common property - for instance, that the last CDR is not equal to NIL - can be drawn by an automated system. The property corresponds to a concept PLISTP.

The author has experimented with guessing from examples a non-trivial property of recursive functions [30]. A measure of a function is a pair of a well-founded relation and a well-founded structure such that for each recursive call of the function the well-founded structure gets smaller according to the well-founded relation. Extending the measure guessing system to be more intelligent would be quite challenging.

8.2.3 Other Applications

Automatic construction of LISP functions from examples has been investigated [26, 47]. Given a list of pairs of input value and output value, a LISP function is generated by an automatic programming system which returns the corresponding output values for input values. Murray has employed examples in

checking students' programs in his LISP tutoring system [36]. The system called "Talus" has adopted a hierarchical method for debugging students programs - EGS has been incorporated into Talus as a counter-example generator. In Talus, the programming task is presented in English and the student's definition is compared with the corresponding (stored) reference definition. It first checks both definitions with the stored examples to see if they return the same output values for the same input values. If no bugs are found with those examples, then EGS is called to generate counter-examples for the statement that both definitions are equivalent. If EGS fails to find counter examples within a certain time limit, the system attempts to prove that both definitions are equivalent by calling the Boyer-Moore theorem prover.

9. Conclusions

In this thesis we have described research on implementation of an automatic example generation system and an experiment with the system in theorem proving. We summarize our accomplishments here and conclude with some remarks on examples and Artificial Intelligence.

First, we have implemented the EGS system which automatically generates examples in a formal domain, the Boyer-Moore theory. With the system we have attempted to model some aspects of human example generation. Human example generation involves various kinds of domain and heuristic knowledge. It also employs powerful reasoning capability. EGS generates examples by successive refinements of a given constraint formula; such refinements are carried out based on knowledge. EGS employs not only basic knowledge such as function definitions and logically derived facts (theorems) but also procedural knowledge such as equation solvers and case analyzers. Function definitions allow EGS to transform a constraint involving high level abstract concepts into a constraint involving lower level concepts. By doing so, restrictive constraints can be resolved in lower level terms. Lemma-based simplification globally simplifies constraints and propagates throughout the constraint any local transformations which are generally caused by the application of procedural knowledge. Procedural knowledge allows local transformation of a constraint; such knowledge efficiently provides solutions for local constraints. Basic knowledge provides EGS with generality whereas the procedural knowledge provides efficiency. By appropriately employing basic knowledge and procedural knowledge EGS has achieved both generality and efficiency.

Second, we have experimented with EGS for the problem of controlling backward chaining in the Boyer-Moore theorem prover. We have interfaced EGS with the theorem prover, proved more than 400 theorems using examples in controlling backward chaining and compared the performance of the theorem prover with the case in which examples are not used. In some cases examples pruned substantial portion of backward chainings during proof attempts; however, the overall result of our experiment was negative. We have found that (semantically) irrelevant literals in conjectures prohibit effective use of examples in the theorem prover; conjectures containing irrelevant literals are created by the induction strategy of the theorem prover.

Finally, we have proposed several improvements to be made to our example generation system and some potential applications of the system. We have found that man and machine can cooperate with each other to generate useful examples and use them effectively. In our experiment we have observed that the examples generated by EGS are sometimes not very effective in pruning unpromising subgoals. Because useful examples are generally expensive, however, EGS tries to generate cheaper examples first. Also, some EGS-generated examples are not appropriate for certain situations. For instance, the generated examples cause some functions to evaluate to too big numbers. In these cases, the user can provide the system with hints in terms of new conditions restricting certain variables' values or by specifically instantiating some variables to interesting values.

Examples in general are a very valuable tool in human intelligence. Various activities of human intelligence, such as reasoning, hypothesizing, understanding, and explaining, are intimately related to examples. Such activities need a rich stock of good examples and often require to generate new examples when known examples are no longer appropriate. A human seems to employ powerful reasoning when he generates examples.

Examples are also important in Artificial Intelligence, as the study of computer modelling of human intelligence. They have been used in various areas of Artificial Intelligence, such as automated reasoning, machine learning, and intelligent computer- assisted instruction and tutoring. Examples are computation-oriented representation. They can be computationally (efficiently) processed to produce an explicit result and the intermediate steps of such processes can be examined. Also, the epistemological adequacy of examples guarantees that they can be easily understood. These characteristics of examples enable us to clearly understand abstract concepts and complicated procedures with examples. Common patterns and properties shared among examples can be quickly computed and they often develop into new concepts and conjectures. This characteristic of examples allows us to hypothesize conjectures and create new concepts from examples.

Investigation into the philosophical aspects of examples could provide valuable insights to automatic example generation and example application. Also, better understanding of human example generation would enable us to design a powerful and efficient example generation system. We strongly believe that better understanding of the true nature of examples would greatly enrich Artificial Intelligence.

I. Some Function Definitions in the Boyer-Moore Theory

```
(APPEND X Y) =
  (IF (LISTP X)
      (CONS (CAR X)
            (APPEND (CDR X) Y))
      Y)

(DELETE X Y) =
  (IF (LISTP Y)
      (IF (EQUAL X (CAR Y))
          (CDR Y)
          (CONS (CAR Y) (DELETE X (CDR Y))))
      Y)

(SUBBAGP X Y) =
  (IF (LISTP X)
      (IF (MEMBER (CAR X) Y)
          (SUBBAGP (CDR X) (DELETE (CAR X) Y))
          F)
      T)

(BAGDIFF X Y) =
  (IF (LISTP Y)
      (IF (MEMBER (CAR Y) X)
          (BAGDIFF (DELETE (CAR Y) X) (CDR Y))
          (BAGDIFF X (CDR Y)))
      X)

(REVERSE X) =
  (IF (LISTP X)
      (APPEND (REVERSE (CDR X))
              (CONS (CAR X) NIL))
      NIL)

(PLISTP X) =
  (IF (LISTP X)
      (PLISTP (CDR X))
      (EQUAL X NIL))

(EQP X Y) =
  (EQUAL (FIX X) (FIX Y))

(FLATTEN X) =
  (IF (LISTP X)
      (APPEND (FLATTEN (CAR X))
              (FLATTEN (CDR X)))
      (CONS X NIL))

(MC-FLATTEN X Y) =
  (IF (LISTP X)
      (MC-FLATTEN (CAR X) (MC-FLATTEN (CDR X) Y))
      (CONS X Y))

(INTERSECT X Y) =
  (IF (LISTP X)
```

```
      (IF (MEMBER (CAR X) Y)
          (CONS (CAR X) (INTERSECT (CDR X) Y))
          (INTERSECT (CDR X) Y))
      NIL)

(UNION X Y) =
  (IF (LISTP X)
      (IF (MEMBER (CAR X) Y)
          (UNION (CDR X) Y)
          (CONS (CAR X) (UNION (CDR X) Y)))
      Y)

(NTH X N) =
  (IF (ZEROP N)
      X
      (NTH (CDR X) (SUB1 N)))

(GREATEREQP X Y) =
  (NOT (LESSP X Y))

(ORDERED L) =
  (IF (LISTP L)
      (IF (LISTP (CDR L))
          (IF (LESSP (CADR L) (CAR L))
              F
              (ORDERED (CDR L)))
          T)
      T)

(ADDTOLIST X L) =
  (IF (LISTP L)
      (IF (LESSP X (CAR L))
          (CONS X L)
          (CONS (CAR L) (ADDTOLIST X (CDR L))))
      (CONS X NIL))

(SORT L) =
  (IF (LISTP L)
      (ADDTOLIST (CAR L) (SORT (CDR L)))
      NIL)

(ASSOC X Y) =
  (IF (LISTP Y)
      (IF (EQUAL X (CAAR Y))
          (CAR Y)
          (ASSOC X (CDR Y)))
      NIL)

(BOOLEAN X) =
  (OR (EQUAL X T)
      (EQUAL X F))

(IFF X Y) =
  (AND (IMPLIES X Y) (IMPLIES Y X))

(ODD X) =
  (IF (ZEROP X)
```

```
      F
      (IF (ZEROP (SUB1 X))
          T
          (ODD (SUB1 (SUB1 X)))))

(EVEN1 X) =
  (IF (ZEROP X)
      T
      (ODD (SUB1 X)))

(EVEN2 X) =
  (IF (ZEROP X)
      T
      (IF (ZEROP (SUB1 X))
          F
          (EVEN2 (SUB1 (SUB1 X)))))

(DOUBLE I) =
  (IF (ZEROP I)
      0
      (ADD1 (ADD1 (DOUBLE (SUB1 I)))))

(HALF I) =
  (IF (ZEROP I)
      0
      (IF (ZEROP (SUB1 I))
          0
          (ADD1 (HALF (SUB1 (SUB1 I)))))))

(EXP I J) =
  (IF (ZEROP J)
      1
      (TIMES I (EXP I (SUB1 J)))))

(COUNT-LIST A L) =
  (IF (LISTP L)
      (IF (EQUAL A (CAR L))
          (ADD1 (COUNT-LIST A (CDR L)))
          (COUNT-LIST A (CDR L)))
      0)

(NUMBER-LISTP L) =
  (IF (LISTP L)
      (IF (NUMBERP (CAR L))
          (NUMBER-LISTP (CDR L))
          F)
      (EQUAL L NIL))

(XOR P Q) =
  (IF Q (IF P F T) (EQUAL P T))

(SUBST X Y Z) =
  (IF (EQUAL Y Z)
      X
      (IF (LISTP Z)
          (CONS (SUBST X Y (CAR Z))
                (SUBST X Y (CDR Z)))
```

```
          Z))

(OCCUR X Y) =
  (IF (EQUAL X Y)
      T
      (IF (LISTP Y)
          (IF (OCCUR X (CAR Y))
              T
              (OCCUR X (CDR Y)))
          F))

(FACT I) =
  (IF (ZEROP I)
      1
      (TIMES I (FACT (SUB1 I)))))

(GCD X Y) =
  (IF (ZEROP X)
      (FIX Y)
      (IF (ZEROP Y)
          X
          (IF (LESSP X Y)
              (GCD X (DIFFERENCE Y X))
              (GCD (DIFFERENCE X Y) Y)))))

(DIVIDES X Y) =
  (ZEROP (REMAINDER Y X))

(PRIME1 X Y) =
  (IF (ZEROP Y)
      F
      (IF (EQUAL Y 1)
          T
          (AND (NOT (DIVIDES Y X))
               (PRIME1 X (SUB1 Y))))))

(PRIME X) =
  (AND (NOT (ZEROP X))
       (NOT (EQUAL X 1))
       (PRIME1 X (SUB1 X)))

(GREATEST-FACTOR X Y) =
  (IF (OR (ZEROP Y)
          (EQUAL Y 1))
      X
      (IF (DIVIDES Y X)
          Y
          (GREATEST-FACTOR X (SUB1 Y)))))

(PRIME-FACTORS X) =
  (IF (OR (ZEROP X)
          (EQUAL (SUB1 X) 0))
      NIL
      (IF (PRIME1 X (SUB1 X))
          (CONS X NIL)
          (APPEND (PRIME-FACTORS (GREATEST-FACTOR X (SUB1 X)))
                  (PRIME-FACTORS
```

```
                    (QUOTIENT X (GREATEST-FACTOR X (SUB1 X)))))))

(PRIME-LIST L) =
  (IF (NLISTP L)
      T
      (AND (PRIME (CAR L))
           (PRIME-LIST (CDR L))))

(TIMES-LIST L) =
  (IF (NLISTP L)
      1
      (TIMES (CAR L)
             (TIMES-LIST (CDR L))))

(GREATEREQPR W Z) =
  (IF (ZEROP W)
      (ZEROP Z)
      (IF (EQUAL W Z)
          T
          (GREATEREQPR (SUB1 W) Z)))

(PERM A B) =
  (IF (NLISTP A)
      (NLISTP B)
      (IF (MEMBER (CAR A) B)
          (PERM (CDR A)
                (DELETE (CAR A) B))
          F))

(MAXIMUM L) =
  (IF (NLISTP L)
      0
      (IF (LESSP (CAR L) (MAXIMUM (CDR L)))
          (MAXIMUM (CDR L))
          (CAR L)))

(ORDERED2 L) =
  (IF (LISTP L)
      (IF (LISTP (CDR L))
          (IF (LESSP (CAR L) (CADR L))
              F
              (ORDERED2 (CDR L)))
          T)
      T)

(DSORT L) =
  (IF (NLISTP L)
      NIL
      (CONS (MAXIMUM L)
            (DSORT (DELETE (MAXIMUM L) L))))

(ADDTOLIST2 X L) =
  (IF (LISTP L)
      (IF (LESSP X (CAR L))
          (CONS (CAR L) (ADDTOLIST2 X (CDR L)))
          (CONS X L))
      (CONS X NIL))
```

```
(SORT2 L) =
  (IF (NLISTP L)
      NIL
      (ADDTOLIST2 (CAR L) (SORT2 (CDR L))))

(EVEN X) =
  (EQUAL 0 (REMAINDER X 2))
```

II. Some Theoretical Aspects of the EGS Example Generation

This section describes some theoretical aspects of example generation in the Boyer-Moore theory. They include the undecidability of the example generation problem and the soundness of the EGS example generation.

II.1 Example Generation Problem is Undecidable

We will prove that the example generation problem in the Boyer-Moore theory is unsolvable. The sketch of the proof is following. We first define some new functions over integers which correspond to PLUS and TIMES over natural numbers. Within the theory we show that diophantine equations can be expressed. Next, we show that if the problem of example generation in the Boyer-Moore theory is decidable, then the problem whether any diophantine equation has integer solutions (called Hilbert's tenth problem) is solvable. However, Hilbert's tenth problem is unsolvable[49], therefore we can conclude that the example generation problem is unsolvable.

Let's assume that we have a theory in which the shell class of numbers is axiomatized and some number theoretic functions such as PLUS, TIMES, and DIFFERENCE are defined. We introduce a new shell class negative numbers as follows:

```
Shell Definition:
Add the shell MINUS of one argument
with recognizer MINUSP,
accessor ABS,
type restriction NUMBERP,
default value (ZERO), and
well-founded relation ABSP.
```

Also, we make a notational convention that we abbreviate (MINUSP n) to -n where n abbreviate a number. Now we define new functions ZPLUS, ZTIMES, and ZEXP as follows.

```
(ZPLUS X Y) =
  (IF (AND (NUMBERP X) (NUMBERP Y))
      (PLUS X Y)
      (IF (AND (NUMBERP X) (MINUSP Y))
          (IF (LESSP X (ABS Y))
              (MINUS (DIFFERENCE Y X))
              (DIFFERENCE X (ABS Y)))
          (IF (AND (MINUSP X) (NUMBERP Y))
              (IF (LESSP (ABS X) Y)
                  (DIFFERENCE Y X)
                  (MINUS (DIFFERENCE (ABS X) Y)))
              (MINUS (PLUS (ABS X) (ABS Y)))))))

(ZTIMES X Y) =
  (IF (AND (NUMBERP X) (NUMBERP Y))
      (TIMES X Y)
      (IF (AND (NUMBERP X) (MINUSP Y))
          (MINUS (TIMES X (ABS Y)))
          (IF (AND (MINUSP X) (NUMBERP Y))
              (MINUS (TIMES (ABS X) Y))
```

[49]It was proved by Matiyacevic, a Russian mathematician [13].

```
                (TIMES (ABS X) (ABS Y)))))

(ZEXP X Y) =
  (IF (ZEROP Y)
      1
      (ZTIMES X (ZEXP X (SUB1 Y)))).
```

The functions ZPLUS, ZTIMES, and ZEXP actually extend PLUS, TIMES, and EXP over the domain of integers instead of the domain of natural numbers. It can easily be shown that an arbitrary diophantine equation can be expressed in terms of ZPLUS, ZTIMES, and ZEXP. For example, any polynomial term of the form $c_i x_i^{n_i}$ can be represented as (ZTIMES c_i (ZEXP x_i n_i)) where c_i, x_i, and n_i denote an integer, a variable, and a natural number, respectively. A diophantine equation of the form

$$t_1 + t_2 + \ldots + t_k = 0 \qquad (1)$$

where each t_i denotes a polynomial term of the form $c_i x_i^{n_i}$ can be represented in the Boyer-Moore theory as

$$(\text{EQUAL } (\text{ZPLUS } s_1\ s_2\ \ldots\ s_k)\ 0) \qquad (2)$$

where each s_i denotes a term (ZTIMES c_i (ZEXP x_i n_i)). Hilbert's tenth problem can be expressed as

> Is there an algorithm which determines whether any given diophantine equation has integer solutions?

Consider the example generation problem with the constraint (2) in the above i.e.

> Find examples satisfying the constraint (2).

In this case the examples are equivalent to the integer solutions of the equation (1). If we had an algorithm which always terminates and returns correct examples if it is the case that the equation of (2) is satisfiable and it returns "no" if the equation (2) is not satisfiable, then Hilbert's tenth problem is solvable. However, Hilbert's tenth problem is not solvable, therefore the example generation problem in the Boyer-Moore theory is not solvable.

II.2 Soundness of Example Generation

EGS example generation is sound in that the generated examples are genuine examples of the constraint. In other words the examples generated by EGS actually satisfy the given constraint. The soundness of examples is guaranteed by that of the interpreter; the soundness of examples is subject to that of the interpreter. In EGS when a task is performed, the current constraint is refined to create other constraints which are hopefully easier to handle. However, such refinement is not necessarily logically sound and is largely based on heuristics. Such heuristic refinement may cause the generation of children constraints which are logically unsound with respect to their parent constraints. In such cases it is possible that the examples derived from those unsound constraints fail to satisfy the original constraint. In EGS the evaluator finally checks that each of the proposed examples actually satisfies the original constraint. The evaluator invokes the interpreter and the interpreter evaluates the constraint with respect to the environment corresponding to the example being evaluated.

It can be argued that the Boyer-Moore interpreter is sound. The soundness of the interpreter is primarily based on the correctness of the LISP code of functions. Details of the LISP code and the interpreter have been described in chapter 3. The constructive nature allows the interpreter to be simple and easy to understand. This simplicity also increases the possibility that the interpreter is sound. For

some primitive functions such as EQUAL, IF, CONS, CAR, and CDR etc, the corresponding LISP code is hand-coded; we simply trust that they are coded correctly.

A formal proof of the correctness of the interpreter is beyond the scope of this thesis. However, we believe that the interpreter is correct and the example generation of EGS is sound with respect to the interpreter.

III. Some of Our Experiment Results

The following compares the performance of the Boyer-Moore theorem prover between the case in which examples are used and the case in which examples are not used. Definitions and theorems are those for proving the correctness of the Rivest-Shamir-Adleman encryption algorithm. Each theorem is followed by two lists which contain performance information: the first list corresponds to the case in which examples are not used and the second one to the case in which examples are used. Each list contains three components: the first component shows the elapsed time (including the time for example generation if examples are generated) for proving the theorem; the second component shows the I/O time; the third component is a pair (NIL in the first list) of the number of examples generated for the proof of the theorem and the time spent by EGS in generating the examples.

```
(DEFN CRYPT (M E N)
  (IF (ZEROP E)
      1
      (IF (EVEN E)
          (REMAINDER (SQUARE (CRYPT M (QUOTIENT E 2) N)) N)
          (REMAINDER
            (TIMES M
                   (REMAINDER (SQUARE (CRYPT M (QUOTIENT E 2) N))
                              N))
            N))))

(PROVE-LEMMA TIMES-MOD-1 (REWRITE)
  (EQUAL (REMAINDER (TIMES X (REMAINDER Y N)) N)
         (REMAINDER (TIMES X Y) N)))

 (20.6 1.3 NIL)
 (9.7 0.4 (6 0.7))

(PROVE-LEMMA TIMES-MOD-2 (REWRITE)
  (EQUAL (REMAINDER (TIMES A (TIMES B (REMAINDER Y N))) N)
         (REMAINDER (TIMES A B Y) N))
             ((USE (TIMES-MOD-1 (X (TIMES A B))))
              (DISABLE TIMES-MOD-1)))

 (0.8 0.1 NIL)
 (4.7 0.0 (6 4.3))

(PROVE-LEMMA CRYPT-CORRECT (REWRITE)
  (IMPLIES (NOT (EQUAL N 1))
           (EQUAL (CRYPT M E N)
                  (REMAINDER (EXP M E) N))))

 (252.8 6.5 NIL)
 (206.1 5.3 (7 1.7))

(PROVE-LEMMA TIMES-MOD-3 (REWRITE)
  (EQUAL (REMAINDER (TIMES (REMAINDER A N) B) N)
         (REMAINDER (TIMES A B) N)))

 (2.0 0.0 NIL)
 (2.0 0.0 (1 0.2))

(PROVE-LEMMA REMAINDER-EXP-LEMMA (REWRITE)
  (IMPLIES (EQUAL (REMAINDER Y A)
```

```
               (REMAINDER Z A))
          (EQUAL (EQUAL (REMAINDER (TIMES X Y) A)
                        (REMAINDER (TIMES X Z) A))
                 T)))

 (62.7 1.2 NIL)
 (51.1 1.2 (12 3.3))

(PROVE-LEMMA REMAINDER-EXP (REWRITE)
  (EQUAL (REMAINDER (EXP (REMAINDER A N) I) N)
         (REMAINDER (EXP A I) N)))

 (10.0 0.8 NIL)
 (10.6 0.8 (6 1.0))

(PROVE-LEMMA EXP-MOD-IS-1 (REWRITE)
  (IMPLIES (EQUAL (REMAINDER (EXP M J) P) 1)
           (EQUAL (REMAINDER (EXP M (TIMES I J)) P)
                  1))
    ((USE (EXP-EXP (I M) (J J) (K I))
          (REMAINDER-EXP (A (EXP M J)) (N P)))
     (DISABLE EXP-EXP REMAINDER-EXP)))

 (2.7 0.0 NIL)
 (6.3 0.0 (12 4.0))

(DEFN PDIFFERENCE (A B)
      (IF (LESSP A B)
          (DIFFERENCE B A)
          (DIFFERENCE A B))
      NIL)

(PROVE-LEMMA TIMES-DISTRIBUTES-OVER-PDIFFERENCE (REWRITE)
  (EQUAL (TIMES M (PDIFFERENCE A B))
         (PDIFFERENCE (TIMES M A)
                      (TIMES M B))))

 (7.8 0.2 NIL)
 (8.6 0.2 (6 3.5))

(PROVE-LEMMA EQUAL-MODS-TRICK-1 (REWRITE)
  (IMPLIES (EQUAL (REMAINDER (PDIFFERENCE A B) P)
                  0)
           (EQUAL (EQUAL (REMAINDER A P)
                         (REMAINDER B P))
                  T)))

 (27.1 0.9 NIL)
 (29.0 1.0 (12 4.4))

(PROVE-LEMMA EQUAL-MODS-TRICK-2 (REWRITE)
  (IMPLIES (EQUAL (REMAINDER A P)
                  (REMAINDER B P))
           (EQUAL (REMAINDER (PDIFFERENCE A B) P)
                  0))
    ((DISABLE DIFFERENCE-ELIM)))
```

```
 (67.7 2.4 NIL)
 (51.4 1.6 (12 5.5))

(PROVE-LEMMA PRIME-KEY-TRICK (REWRITE)
  (IMPLIES (AND (EQUAL (REMAINDER (TIMES M A) P)
                       (REMAINDER (TIMES M B) P))
                (NOT (EQUAL (REMAINDER M P) 0))
                (PRIME P))
           (EQUAL (EQUAL (REMAINDER A P)
                         (REMAINDER B P))
                  T))
    ((USE (PRIME-KEY-REWRITE (A M)
                             (B (PDIFFERENCE A B)))
          (EQUAL-MODS-TRICK-2 (A (TIMES M A))
                              (B (TIMES M B))))
     (DISABLE PRIME-KEY-REWRITE EQUAL-MODS-TRICK-2)))

 (32.8 0.1 NIL)
 (49.1 0.1 (24 17.1))

(PROVE-LEMMA PRODUCT-DIVIDES-LEMMA (REWRITE)
  (IMPLIES (EQUAL (REMAINDER X Z) 0)
           (EQUAL (REMAINDER (TIMES Y X) (TIMES Y Z))
                  0)))

 (13.3 0.4 NIL)
 (9.2 0.4 (12 3.6))

(PROVE-LEMMA PRODUCT-DIVIDES (REWRITE)
  (IMPLIES (AND (EQUAL (REMAINDER X P) 0)
                (EQUAL (REMAINDER X Q) 0)
                (PRIME P)
                (PRIME Q)
                (NOT (EQUAL P Q)))
           (EQUAL (REMAINDER X (TIMES P Q)) 0)))

 (65.8 0.6 NIL)
 (86.8 0.6 (29 27.2))

(PROVE-LEMMA THM-53-SPECIALIZED-TO-PRIMES NIL
  (IMPLIES (AND (PRIME P)
                (PRIME Q)
                (NOT (EQUAL P Q))
                (EQUAL (REMAINDER A P)
                       (REMAINDER B P))
                (EQUAL (REMAINDER A Q)
                       (REMAINDER B Q)))
           (EQUAL (REMAINDER A (TIMES P Q))
                  (REMAINDER B (TIMES P Q))))
    NIL)

 (39.6 0.1 NIL)
 (68.2 0.1 (18 41.7))

(PROVE-LEMMA COROLLARY-53 (REWRITE)
  (IMPLIES (AND (PRIME P)
                (PRIME Q)
```

```
               (NOT (EQUAL P Q))
               (EQUAL (REMAINDER A P)
                      (REMAINDER B P))
               (EQUAL (REMAINDER A Q)
                      (REMAINDER B Q))
               (NUMBERP B)
               (LESSP B (TIMES P Q)))
          (EQUAL (EQUAL (REMAINDER A (TIMES P Q)) B)
                 T))
        ((USE (THM-53-SPECIALIZED-TO-PRIMES))
         (DISABLE THM-53-SPECIALIZED-TO-PRIMES)))

 (23.1 0.2 NIL)
 (75.6 0.2 (31 53.7))

(PROVE-LEMMA THM-55-SPECIALIZED-TO-PRIMES (REWRITE)
  (IMPLIES (AND (PRIME P)
               (NOT (EQUAL (REMAINDER M P) 0)))
          (EQUAL (EQUAL (REMAINDER (TIMES M X) P)
                        (REMAINDER (TIMES M Y) P))
                 (EQUAL (REMAINDER X P)
                        (REMAINDER Y P)))))

 (111.2 0.3 NIL)
 (69.0 0.2 (12 20.0))

(PROVE-LEMMA COROLLARY-55 (REWRITE)
  (IMPLIES (PRIME P)
          (EQUAL (EQUAL (REMAINDER (TIMES M X) P)
                        (REMAINDER M P))
                 (OR (EQUAL (REMAINDER M P) 0)
                     (EQUAL (REMAINDER X P) 1))))
            ((USE (THM-55-SPECIALIZED-TO-PRIMES (Y 1)))
             (DISABLE THM-55-SPECIALIZED-TO-PRIMES)))

 (83.8 0.6 NIL)
 (48.2 0.2 (11 10.4))

(DEFN ALL-DISTINCT (L)
  (IF (NLISTP L)
      T
      (AND (NOT (MEMBER (CAR L) (CDR L)))
           (ALL-DISTINCT (CDR L)))))

(DEFN ALL-LESSEQP (L N)
  (IF (NLISTP L)
      T
      (AND (NOT (LESSP N (CAR L)))
           (ALL-LESSEQP (CDR L) N))))

(DEFN ALL-NON-ZEROP (L)
  (IF (NLISTP L)
      T
      (AND (NOT (ZEROP (CAR L)))
           (ALL-NON-ZEROP (CDR L)))))

(DEFN POSITIVES (N)
```

```
  (IF (ZEROP N)
      NIL
      (CONS N (POSITIVES (SUB1 N)))))

(PROVE-LEMMA LISTP-POSITIVES (REWRITE)
  (EQUAL (LISTP (POSITIVES N))
         (NOT (ZEROP N))))

 (4.8 3.6 NIL)
 (1.2 0.4 (7 0.3))

(PROVE-LEMMA CAR-POSITIVES (REWRITE)
  (EQUAL (CAR (POSITIVES N))
         (IF (ZEROP N) 0 N)))

 (5.3 0.8 NIL)
 (1.3 0.4 (6 0.3))

(PROVE-LEMMA MEMBER-POSITIVES (REWRITE)
  (EQUAL (MEMBER X (POSITIVES N))
         (IF (ZEROP X) F (LESSP X (ADD1 N)))))

 (6.5 1.9 NIL)
 (6.8 1.0 (12 2.3))

(PROVE-LEMMA ALL-NON-ZEROP-DELETE (REWRITE)
  (IMPLIES (ALL-NON-ZEROP L)
           (ALL-NON-ZEROP (DELETE X L))))

 (2.5 0.8 NIL)
 (1.9 0.4 (12 0.6))

(PROVE-LEMMA ALL-DISTINCT-DELETE (REWRITE)
  (IMPLIES (ALL-DISTINCT L)
           (ALL-DISTINCT (DELETE X L))))

 (3.8 0.4 NIL)
 (3.2 0.4 (12 0.5))

(PROVE-LEMMA PIGEON-HOLE-PRINCIPLE-LEMMA-1 (REWRITE)
  (IMPLIES (AND (ALL-DISTINCT L)
                (ALL-LESSEQP L (ADD1 N)))
           (ALL-LESSEQP (DELETE (ADD1 N) L) N)))

 (33.9 8.0 NIL)
 (21.9 1.7 (13 6.7))

(PROVE-LEMMA PIGEON-HOLE-PRINCIPLE-LEMMA-2 (REWRITE)
  (IMPLIES (AND (NOT (MEMBER (ADD1 N) X))
                (ALL-LESSEQP X (ADD1 N)))
           (ALL-LESSEQP X N)))

 (10.4 3.7 NIL)
 (5.4 0.9 (18 1.8))

(PROVE-LEMMA PERM-MEMBER (REWRITE)
  (IMPLIES (AND (PERM A B) (MEMBER X A))
```

```
          (MEMBER X B)))

 (8.0 2.9 NIL)
 (5.2 0.7 (17 2.3))

(DEFN PIGEON-HOLE-INDUCTION (L)
  (IF (LISTP L)
      (IF (MEMBER (LENGTH L) L)
          (PIGEON-HOLE-INDUCTION (DELETE (LENGTH L) L))
          (PIGEON-HOLE-INDUCTION (CDR L)))
      T))

(PROVE-LEMMA PIGEON-HOLE-PRINCIPLE NIL
  (IMPLIES (AND (ALL-NON-ZEROP L)
                (ALL-DISTINCT L)
                (ALL-LESSEQP L (LENGTH L)))
           (PERM (POSITIVES (LENGTH L)) L))
             ((INDUCT (PIGEON-HOLE-INDUCTION L))))

 (76.0 2.0 NIL)
 (36.7 1.2 (18 1.8))

(PROVE-LEMMA PERM-TIMES-LIST NIL
  (IMPLIES (PERM L1 L2)
           (EQUAL (TIMES-LIST L1)
                  (TIMES-LIST L2))))

 (11.2 2.1 NIL)
 (6.7 0.9 (12 0.8))

(PROVE-LEMMA TIMES-LIST-POSITIVES (REWRITE)
  (EQUAL (TIMES-LIST (POSITIVES N))
         (FACT N)))

 (1.2 0.4 NIL)
 (1.0 0.2 (4 0.3))

(PROVE-LEMMA TIMES-LIST-EQUAL-FACT (REWRITE)
  (IMPLIES (PERM (POSITIVES N) L)
           (EQUAL (TIMES-LIST L) (FACT N)))
             ((USE (PERM-TIMES-LIST (L1 (POSITIVES N))
                                    (L2 L)))
              (DISABLE PERM-TIMES-LIST)))

 (2.0 0.6 NIL)
 (4.2 0.2 (12 2.8))

(PROVE-LEMMA PRIME-FACT (REWRITE)
  (IMPLIES (AND (PRIME P) (LESSP N P))
           (NOT (EQUAL (REMAINDER (FACT N) P) 0)))
             ((INDUCT (FACT N))))

 (38.1 0.7 NIL)
 (22.0 0.2 (18 4.5))

(DEFN S (N M P)
  (IF (ZEROP N)
```

```
      NIL
      (CONS (REMAINDER (TIMES M N) P)
            (S (SUB1 N) M P))))

(PROVE-LEMMA REMAINDER-TIMES-LIST-S NIL
  (EQUAL (REMAINDER (TIMES-LIST (S N M P)) P)
         (REMAINDER (TIMES (FACT N) (EXP M N))
                    P)))

 (54.4 1.3 NIL)
 (15.6 0.3 (6 1.7))

(PROVE-LEMMA ALL-DISTINCT-S-LEMMA (REWRITE)
  (IMPLIES (AND (PRIME P)
                (NOT (EQUAL (REMAINDER M P) 0))
                (NUMBERP N1)
                (LESSP N2 N1)
                (LESSP N1 P))
           (NOT (MEMBER (REMAINDER (TIMES M N1) P)
                        (S N2 M P))))
             ((INDUCT (S N2 M P)))))

 (87.1 1.3 NIL)
 (50.4 0.7 (30 15.6))

(PROVE-LEMMA ALL-DISTINCT-S (REWRITE)
  (IMPLIES (AND (PRIME P)
                (NOT (EQUAL (REMAINDER M P) 0))
                (LESSP N P))
           (ALL-DISTINCT (S N M P))))

 (194.0 3.9 NIL)
 (56.9 2.5 (24 4.4))

(PROVE-LEMMA ALL-NON-ZEROP-S (REWRITE)
  (IMPLIES (AND (PRIME P)
                (NOT (EQUAL (REMAINDER M P) 0))
                (LESSP N P))
           (ALL-NON-ZEROP (S N M P))))

 (961.7 3.4 NIL)
 (61.4 1.6 (24 4.3))

(PROVE-LEMMA ALL-LESSEQP-S (REWRITE)
  (IMPLIES (NOT (ZEROP P))
           (ALL-LESSEQP (S N M P) (SUB1 P))))

 (17.4 2.3 NIL)
 (6.8 0.7 (12 0.8))

(PROVE-LEMMA LENGTH-S (REWRITE)
  (EQUAL (LENGTH (S N M P)) (FIX N)))

 (5.7 0.7 NIL)
 (2.3 0.3 (6 0.3))

(PROVE-LEMMA FERMAT-THM (REWRITE)
```

```
  (IMPLIES (AND (PRIME P)
                (NOT (EQUAL (REMAINDER M P) 0)))
           (EQUAL (REMAINDER (EXP M (SUB1 P)) P) 1))
         ((USE (PIGEON-HOLE-PRINCIPLE (L (S (SUB1 P) M P)))
               (REMAINDER-TIMES-LIST-S (N (SUB1 P))))
          (DISABLE PIGEON-HOLE-PRINCIPLE REMAINDER-TIMES-LIST-S)))

 (44.9 0.2 NIL)
 (31.4 0.2 (18 6.7))

(PROVE-LEMMA CRYPT-INVERTS-STEP-1 NIL
  (IMPLIES (PRIME P)
           (EQUAL
             (REMAINDER (TIMES M (EXP M (TIMES K (SUB1 P)))) P)
             (REMAINDER M P))))

 (16.1 0.2 NIL)
 (23.9 0.1 (8 6.3))

(PROVE-LEMMA CRYPT-INVERTS-STEP-1A (REWRITE)
  (IMPLIES (PRIME P)
           (EQUAL
             (REMAINDER
               (TIMES M (EXP M (TIMES K (SUB1 P) (SUB1 Q))))
               P)
             (REMAINDER M P)))
         ((USE (CRYPT-INVERTS-STEP-1 (K (TIMES K (SUB1 Q)))))
          (DISABLE CRYPT-INVERTS-STEP-1)))

 (60.7 0.1 NIL)
 (41.0 0.1 (6 9.5))

(PROVE-LEMMA CRYPT-INVERTS-STEP-1B (REWRITE)
  (IMPLIES (PRIME Q)
           (EQUAL
             (REMAINDER
               (TIMES M (EXP M (TIMES K (SUB1 P) (SUB1 Q))))
               Q)
             (REMAINDER M Q)))
         ((USE (CRYPT-INVERTS-STEP-1 (P Q)
                                     (K (TIMES K (SUB1 P)))))
          (DISABLE CRYPT-INVERTS-STEP-1)))

 (28.0 0.2 NIL)
 (63.1 0.2 (2 9.2))

(PROVE-LEMMA CRYPT-INVERTS-STEP-2 (REWRITE)
  (IMPLIES (AND (PRIME P)
                (PRIME Q)
                (NOT (EQUAL P Q))
                (NUMBERP M)
                (LESSP M (TIMES P Q))
                (EQUAL (REMAINDER ED
                                  (TIMES (SUB1 P) (SUB1 Q)))
                       1))
           (EQUAL (REMAINDER (EXP M ED) (TIMES P Q))
                  M)))
```

```
 (72.8 0.8 NIL)
 (91.0 1.1 (32 35.4))

(PROVE-LEMMA CRYPT-INVERTS NIL
  (IMPLIES (AND (PRIME P)
                (PRIME Q)
                (NOT (EQUAL P Q))
                (EQUAL N (TIMES P Q))
                (NUMBERP M)
                (LESSP M N)
                (EQUAL (REMAINDER (TIMES E D)
                                  (TIMES (SUB1 P) (SUB1 Q)))
                       1))
           (EQUAL (CRYPT (CRYPT M E N) D N) M)))

 (67.5 0.1 NIL)
 (93.9 0.1 (25 55.0))
```

Bibliography

1. Aubin, R. *Mechanizing Structural Induction*. Ph.D. Th., University of Edinburgh, 1976.

2. Ballantyne, A. M. and Bledsoe, W. W. "On Generating and Using Examples in Proof Discovery". *Machine Intelligence 10* (1982).

3. Bledsoe, W. W. The SUP-INF Method in Presburger Arithmetic. Tech. Rept. ATP-18, Department of Mathematics, Unversity of Texas at Austin, 1974.

4. Bledsoe, W. W. "Using Examples to Generate Instantiations for Set Variables". *Proc. of IJCAI-83* (1983), 892-901.

5. Boyer, R. S. and Moore, J S. *A Computational Logic.* Academic Press, New York, 1979.

6. Boyer, R. S. and Moore, J S. Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures. In *The Correctness Problem in Computer Science*, Boyer, R. S. and Moore, J S., Eds., Academic Press, London, 1981.

7. Boyer, R. S. and Moore, J S. Proof Checking the RSA Public Key Encryption Algorithm. Tech. Rept. ICSCA-CMP-33, Institute for Computing Science and Computer Applications, University of Texas at Austin, 1982.

8. Boyer, R. S. and Moore, J S. Integrating Decision Procedures into Heuristic Theorem Provers: A Case Study of Linear Arithmetic. Tech. Rept. ICSCA-CMP-44, Institute for Computing Science and Computer Applications, University of Texas at Austin, 1985.

9. Brotz, D. *Proving Theorems by Mathematical Induction*. Ph.D. Th., Stanford University, 1974.

10. Buchanan, B. G. and Feigenbaum, E. A. "DENDRAL and META-DENDRAL: Their Applications Dimension". *Artificial Intelligence 11* (1978), 5-24.

11. Bundy, Alan. *Artificial Mathematicians: The Computer Modelling of Mathematical Reasoning.* , 1982. A Draft.

12. Cohen, D. *Knowledge Based Theorem Proving and Learning*. Ph.D. Th., Carnegie-Mellon University, 1980.

13. Davis, M. "Hilbert's Tenth Problem is Unsolvable". *American Mathematical Monthly 80* (1973), 233-269.

14. Dietterich, T. G. and Michalski, R. S. "Inductive Learning of Structural Descriptions: Evaluation Criteria and Comparative Review of Selected Methods". *Artificial Intelligence 16* (1981), 257-294.

15. Ernst, George W. and Newell, Allen. *GPS: A Case Study in Generality and Problem Solving.* Academic Press, New York, 1969.

16. Fikes, R. E. "REF-ARF: A System for Solving Problems Stated as Procedures". *Artificial Intelligence 1* (1970), 27-120.

17. Fujiwara, H. and Shimono, T. "On the Acceleration of Test Generation Algorithms". *IEEE Trans. Computers C-32* (1983), 1137-1144.

18. Gallaire, H. and Minker, J. (editors). *Logic and Databases.* Plenum Press, New York, 1978.

19. Gelernter, H. Realization of a Geometry-Theorem Proving Machine. In *Computers and Thoughts*, Feigenbaum, E. A. and Feldman, J., Eds., McGraw-Hill Book Company, 1963.

20. Gelernter, H. and Hansen, J. R. and Loveland, D. W. Empirical Explorations of the Geometry-Theorem Proving Machine. In *Computers and Thoughts*, Feigenbaum, E. A. and Feldman, J., Eds., McGraw-Hill Book Company, 1963.

21. Gilmore, P. C. "An Examination of the Geometry Theorem Machine". *Artificial Intelligence 1* (1970), 171-187.

22. Goel, P. "An Implicit Enumeration Algorithm to Generate Tests for Combinatorail Logic Circuits". *IEEE Trans. Computers C-30* (1981), 215-222.

23. Goodenough, J. and Gerhart, S. L. "Toward a Theory of Test Data Selection". *IEEE Trans. Software Eng. SE-1* (1975), 156-173.

24. Goodstein, R. L. *Recursive Number Theory. A Development of Recursive Arithmetic in a Logic Free Equation Calculus.* North-Holland Publishing Co., Amsterdam, 1957.

25. Green, Cordell. "Application of Theorem Proving to Problem Solving". *Proc. of IJCAI* (1969), 219-239.

26. Hardy, Steven. Automatic Induction of LISP Functions. Essex University, 1973.

27. Howden, W. E. "Methodology for the Generation of Program Test Data". *IEEE Trans. Computers C-24* (1975), 554-559.

28. Howden, W. E. "Weak Mutation Testing and Completeness of Test Sets". *IEEE Trans. Software Eng. SE-8* (1982), 371-379.

29. Jouannaud, Jean-Pierre et al. "SISP/1 An Interactive System Able to Synthesize Functions from Examples". *Proc. of IJCAI-77* (1977), 412-418.

30. Kim, Myung W. Measure Guessing: An Experiment with Hypothesis Generation From Examples. Institute for Computing Science and Computer Applications, University if Texas at Austin, 1986.

31. Lakatos, I. *Proofs and Refutations.* Cambridge University Press, 1976.

32. Langley, P. W. *Descriptive Discovery Processes: Experiments in Baconian Science*. Ph.D. Th., Carnegie-Mellon University, 1980.

33. Lenat, D. B. *AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search*. Ph.D. Th., Stanford University, 1976.

34. Mitchell, T. M. *Version Spaces: An Approach to Concept Learning*. Ph.D. Th., Stanford University, 1978.

35. Moore, J S. "A Mechanical Proof of the Termination of Takeuchi's Function". *Information Processing Letters 9* (1979), 176-181.

36. Murray, W. R. *Automatic Program Debugging for Intelligent Tutoring Systems*. Ph.D. Th., University of Texas at Austin, 1986.

37. Pereira, L. M. and Pereira, F. C. and Warren, D. H. User's Guide to DECsystem-10 PROLOG. . Sept., 1978.

38. Polya, G. *Mathematics and Plausible Reasoning.* Princeton University Press, 1954. Vol. 1 and 2.

39. Polya, G. *Mathematical Discovery.* Wiley, New York, 1962. Vol. 1 and 2.

40. Reiter, R. "A Semantically Guided Deductive System for Automatic Theorem Proving". *IEEE Trans. on Computers* , C-25(4) (1976), 328-334.

41. Rissland, Edwina L. and Valcarce, Eduardo M. and Ashley, Kevin D. "Explaining and Arguing with Examples". *Proc. of AAAI-84* (1978), 288-294.

42. Rissland, Edwina. "Understanding Understanding Mathematics". *Cognitive Science 2* (1978).

43. Rissland, Edwina and Soloway, Elliot. Generating Examples in LISP: Data and Programs. Tech. Rept. COINS TR-80-07, Department of Computer and Information Science, University of Massachusetts at Amherst, 1980.

44. Roth, J. P. "Diagnosis of Automata Failure: A Calculus and a Method". *IBM J. Res. Develop. 10* (1966), 278-291.

45. Russinoff, D. M. An Experiment with the Boyer-Moore Program Verification System: A Proof of Wilson's Theorem. Tech. Rept. ICSCA-CMP-38, Institute for Computing Science and Computer Applications, University of Texas at Austin, 1983.

46. Skolem, T. The Foundations of Elementary Arithmetic Established by Means of the Recursive Mode of Thought, without the use of Apparent Variables Ranging over Infinite Domains. In *From Frege to Goedel*, J. van Heijenoort, Ed., Harvard University Press, Cambridge, Massachusetts, 1967, pp. 302-333.

47. Summers, P. D. A Methodology for LISP Program Construction from Examples. IBM T.J. Watson Research Center, 1976.

48. Van Emden, M. H. and Kowalski, R. A. "The Semantics of Predicate Logic as a Programming Language". *JACM 23* (1976), 733-742.

49. Waldinger R. J. "Achieving Several Goals Simutaneously". *Machine Intelligence 8* (1977), 94-136.

50. Wang, T. C. "Designing Examples for Semantically Guided Hierarchical Deduction". *Proc. of IJCAI-85 2* (1985), 1201-1207.

51. Weyuker, E. J. and Ostrand, T. J. "Theories of Program Testing and the Application of Revealing Subdomains". *IEEE Trans. Software Eng. SE-6* (1980).

52. Winston, P. H. Learning Structural Descriptions from Examples. In *The Psychology of Computer Vision*, Winston, P. H., Ed., McGraw-Hill Book Company, 1975.

53. Wos, L., Robinson, G. A., and Carson, D. F. "Efficiency and Completeness of the Set of Support Strategy in Theorem Proving". *J. ACM 12* (1965), 536-541.

54. Zloof, M. M. "Query-by-Example: A Data Base Language". *IBM Systems Journal 16(4)* (1977), 324-342.

Table of Contents

List of Figures

List of Tables