# AN EXPERIMENT WITH THE BOYER-MOORE PROGRAM VERIFICATION SYSTEM: A PROOF OF WILSON'S THEOREM

David M. Russinoff

Technical Report #38 July 1983

Institute for Computing Science
The University of Texas at Austin
Austin, Texas 78712
(512)471-1901

# TABLE OF CONTENTS

# ACKNOWLEDGEMENTS

I would like to thank my supervising professors, Robert S. Boyer and J Strother Moore, who introduced me to automatic theorem proving and program verification. I am grateful to them for suggesting the problem for this thesis.

I am also indebted to M. Cochinwala, N. Shankar, W. Uhrig, and W. Young for their help in acquainting me with the TOPS-20 operating system, the Scribe document formatter, and the Boyer-Moore theorem prover.

David M. Russinoff

The University of Texas at Austin
August, 1983

# 1 Introduction

In 1961, McCarthy [8] initiated an effort to create a mathematical theory of computation based on recursively defined functions, conditional expressions, and mathematical induction. Among his goals [7] were the automation of the processes of mathematical proof and verification of computer program specifications.

McCarthy introduced a formalism for describing computable functions and related entities and developed the method of "recursion induction" for proving the equivalence of functions defined within this formalism. The method was discovered in the consideration of mathematical conjectures but has been found to be applicable to a wide class of problems in computer science. McCarthy used it to derive various results in number theory and the theory of symbolic expressions.

The relevance of mathematics to any mechanization of the intellectual process is affirmed by Boole's thesis that "The laws of thought, in all its processes of conception and of reasoning, in all those operations of which language is the expression or the instrument, are of the same kind as are the laws of the acknowledged processes of mathematics."[1] McCarthy recognized this correspondence in connection with his theory of computation, observing a similarity between the problems involved in deriving results in the theory of numbers and those in proving the equivalence of algorithms. He also discussed the limitations of his methods in number theoretic terms:

> In number theory one gets as far as the theorem that if a prime $p$ divides $ab$ then it divides either $a$ or $b$. However, to formulate the unique factorization theorem requires a notation for dealing with sets of integers. Wilson's theorem, a moderately deep result, can be expressed in this formalism but apparently cannot be proved by recursion induction. [8]

The obstacles to the proof of the unique factorization theorem have been overcome by Boyer and Moore, the designers of an automatic program verification system which implements McCarthy's formalism and proof methods. This theorem is among many familiar results in elementary number theory which they have mechanically verified.

The last theorem to which McCarthy referred is a well-known result, originally published by Waring in 1770 and attributed to his student, John Wilson. [6] It states that for any prime $p$, $(p\text{-}1)!$ and $p\text{-}1$ are congruent modulo $p$. In this paper, we describe the use of the Boyer-Moore system in generating a mechanical proof of Wilson's theorem.

## 2 The Boyer-Moore Theorem Prover

In the Boyer-Moore system, as described in [2], proofs are constructed in a quantifier-free logical theory which is built on the propositional calculus with equality, variables, and function symbols. Terms are written in the prefix notation of LISP. The basic theory includes four functions:

- TRUE and FALSE are both functions of zero arguments. The constants (TRUE) and (FALSE) are abbreviated T and F, respectively. The axiom $T \neq F$ allows them to be considered as distinct truth values.
- EQUAL is a function of two arguments, axiomatized to require that (EQUAL l r) have the value T or F, depending on whether $l = r$.
- IF is a function of three arguments. Axioms insure that the value of (IF t u v) is that of v if $t = F$ and is otherwise the value of u.

The function IF allows the use of conditionals in axioms which define new functions, as in the definition

```
(AND P Q) = (IF P (IF Q T F) F).
```

A function capturing the semantics of each of the other logical connectives is similarly defined, allowing a term p to be constructed, corresponding to any formula $\phi$, such that the formulas $p \neq F$ and $\phi$ have the same truth values. In this situation, the term p is said to be a "theorem" if $\phi$ is a theorem. All formulas may thus be represented as terms.

The theory also includes

- a principle which permits the introduction of axioms specifying new types of inductively constructed objects
- a principle for admitting axioms which define new recursive functions
- a principle of induction (a rule of inference) based on the notion of a well-founded relation, which is well-suited for inferring theorems about these objects and functions.

Two types of inductively constructed objects are relevant to the proofs presented below:

- The natural numbers are formalized by the type "number". Peano arithmetic is realized through axioms relating to the functions NUMBERP, ADD1, SUB1, and the constant 0. NUMBERP is the "recognizer" for this type, i.e., (NUMBERP t) returns T or F depending on whether the value of the term t is a number. ADD1 is the usual successor function, and SUB1 is its inverse function.
- Ordered pairs are formalized by the type "list". The functions CONS, CAR, and CDR are axiomatized to have the properties of the familiar LISP functions, and LISTP is the recognizer for this type. Finite sequences are represented by means of CONS and the special constant NIL; the sequence of terms $t_1, \ldots, t_n$ is represented by the term

(CONS $t_1$ (CONS $t_2$ (...(CONS $t_n$ NIL)...))),

which is abbreviated (LIST $t_1$ ... $t_n$).

The theorem proving program naturally relies heavily on induction. Several heuristics are employed to formulate induction schemes based on analysis of the structure of the recursive function definitions and the inductively constructed types which are involved in a conjecture. A proof by induction is only attempted, however, after a simplification procedure has been followed and has failed to establish the theorem.

Every conjecture which is presented to or generated by the theorem prover is first written as a single term and then represented internally as a conjunction of simpler "clauses", each of which is a disjunction of atomic formulas called "literals". In order to establish a clause, the prover first attempts to simplify each of its literals in turn, assuming the others to have the value F. A variety of heuristics are employed, including the use of previously proved lemmas as rewrite rules. A term l is replaced by the term r if a rewrite rule of the form

(IMPLIES h (EQUAL l r))

is encountered and the hypothesis h can be established. The manner of use of a lemma therefore depends on its precise syntactic form.

The user's manual [3] describes the input commands to the system. Of these, only three were used in our proof of Wilson's theorem:

1. The command

   DEFN(<name> <args> <expression>)

   results in the definition of a function with the designated name and formal argument list, to be evaluated by computing the value of the expression. Before admitting a function definition, however, the system must verify that certain conditions are met which guarantee the existence of a unique function satisfying the proposed definition.

2. The command

   PROVE.LEMMA(<name> (REWRITE) <term>)

   initiates an attempt to prove the conjecture "term". If this succeeds, the theorem is stored as a rewrite rule under the given name. Once the proof attempt has begun, there is no interaction with the user until it has completed. However, there is a "hint" facility associated with this command (not mentioned in [3]) which allows the user some control over the method of proof:

   - A list of (previously proved) lemmas may be specified to be used in the proof. These lemmas are instantiated as indicated by the user and inserted into the hypothesis of the conjecture.

- An induction scheme may be specified. Its application becomes the first step in the attempted proof.
- A "disable" feature allows a specified list of theorems to be temporarily deleted from the system's library.

3. The command

INIT(<file>)

serves to initialize the system's data base to that contained in the designated library file. The file most useful for our purpose was RSA.LIB, prepared by Boyer and Moore for their proof of the RSA encryption algorithm (to appear in the American Mathematical Monthly). The file contains the several hundred definitions and lemmas which are listed in [4] and Appendix A of [2]. Included are many basic results of number theory, some of which are cited in the proofs below. The relevant function definitions from RSA.LIB are listed in an appendix.

# 3 The proof of Wilson's theorem

With no guidance by the user permitted during the theorem proving process, the system must be led to the discovery of the proof of a complicated theorem by means of a carefully ordered sequence of relatively simple lemmas. In the case of Wilson's theorem, it was found that three function definitions and forty-two lemmas were sufficient. These are presented in the sequel, numbered according to the order in which they were submitted to the theorem prover. Along with a description of the theorem prover's activity, we include informal proofs where appropriate, using conventional mathematical notation.

## 3.1 The function INVERSE

In this section, we define and establish the properties of a function which assigns to each reduced residue modulo the prime P its multiplicative inverse modulo P:

```
Definition 1.  (INVERSE J P)
                    =
               (IF (EQUAL P 2)
                   (REMAINDER J 2)
                   (REMAINDER (EXP J (DIFFERENCE P 2)) P))
```

where DIFFERENCE and EXP are the ordinary operations of subtraction and exponentiation and (REMAINDER X Y) is the least non-negative residue of X modulo Y. In our original formulation of this function, the case P=2 was overlooked. The (EQUAL P 2) clause was omitted, resulting in (INVERSE 0 2) = 1 (since (EXP 0 0) returns 1). This was discovered only when the attempted proof of Lemma 18 failed, forcing a careful analysis which led to the new definition.

The proofs in this section depend on two important results contained in RSA.LIB. The first is a version of Euclid's First Theorem, which states that for prime $p$, if $p|ab$, then $p|a$ or $p|b$:

```
Lemma.  PRIME.KEY.REWRITE
        (IMPLIES (PRIME P)
                 (EQUAL (EQUAL (REMAINDER (TIMES A B) P) 0)
                        (OR (EQUAL (REMAINDER A P) 0)
                            (EQUAL (REMAINDER B P) 0)))).
```

The second is Fermat's theorem:

```
Lemma.  FERMAT.THM
        (IMPLIES (AND (PRIME P)
                      (NOT (EQUAL (REMAINDER M P) 0)))
                 (EQUAL (REMAINDER (EXP M (SUB1 P)) P) 1)).
```

**3.1.1.** Our first goal is to establish that if J is not divisible by the prime P, then the product of J and (INVERSE J P) is congruent to 1 modulo P :

```
Lemma 2.  (IMPLIES (NOT (ZEROP P))
                   (EQUAL (REMAINDER (TIMES (INVERSE J P) J) P)
                          (REMAINDER (EXP J (SUB1 P)) P))).


Lemma 3.  INVERSE.INVERTS
          (IMPLIES (AND (PRIME P)
                        (NOT (EQUAL (REMAINDER J P) 0)))
                   (EQUAL (REMAINDER (TIMES (INVERSE J P) J) P) 1))
          Hints:  use Lemma 2
                  disable INVERSE.
```

Lemma 3, named INVERSE.INVERTS, is the desired result. It follows immediately from FERMAT.THM. The original proof attempt, however, which did not use Lemma 2, failed, apparently because of confusion over the many possible uses of the hypothesis (PRIME P). Thus, the more general lemma was established first and the system was instructed to use it to prove INVERSE.INVERTS. Since no other properties of the function INVERSE were needed, the second hint was provided in order to prevent the use of its definition, which would only have complicated matters.

**3.1.2.** The next lemma establishes that for a given value of J, (INVERSE J P) is the unique residue modulo P with the property described in INVERSE.INVERTS:

```
Lemma 4.  INVERSE.IS.UNIQUE
          (IMPLIES (AND (PRIME P)
                        (EQUAL 1 (REMAINDER (TIMES M X) P)))
                   (EQUAL (INVERSE M P) (REMAINDER X P)))
          Hints:  use INVERSE.INVERTS with {J/M}
                  use THM.55.SPECIALIZED.TO.PRIMES
                      with {Y/(INVERSE M P)}.
```

The theorem cited in the second hint is found in the library file. When instantiated as indicated in the hint, it becomes

```
(IMPLIES (AND (PRIME P)
              (NOT (EQUAL (REMAINDER M P) 0)))
         (EQUAL (EQUAL (REMAINDER (TIMES M X) P)
                       (REMAINDER (TIMES M (INVERSE M P)) P))
                (EQUAL (REMAINDER X P)
                       (REMAINDER (INVERSE M P) P)))).
```

Thus, the congruence between (TIMES M X) and (TIMES M (INVERSE M P)), both of which are known to be congruent to 1, implies that X is congruent to (INVERSE M P). The conclusion of INVERSE.IS.UNIQUE follows easily (although the mechanical proof involved an analysis of sixteen cases).

**3.1.3.** Lemma 4 is useful in proving that P-1 is its own inverse modulo P, which now follows from the observation that

$$(P-1)(P-1) = 1 + P(P-2).$$

Thus,

```
Lemma 5.  (IMPLIES (AND (NOT (ZEROP N)) (NOT (EQUAL N 1)))
                   (EQUAL (TIMES (SUB1 N) (SUB1 N))
                          (PLUS 1 (TIMES N (SUB1 (SUB1 N)))))).

Lemma 6.  (IMPLIES (AND (NOT (ZEROP N)) (NOT (EQUAL N 1)))
                   (EQUAL (REMAINDER (TIMES (SUB1 N) (SUB1 N)) N) 1))
          Hints:  use Lemma 5
                  use REMAINDER.PLUS.TIMES 2
                      with {J/N, X/1, I/(SUB1 (SUB1 N))}
                  disable Lemma 5 and REMAINDER.PLUS.TIMES.2.

Lemma 7.  SUB1.P.IS.INVOLUTION
          (IMPLIES (PRIME P)
                   (EQUAL (INVERSE (SUB1 P) P) (SUB1 P)))
          Hints:  use INVERSE.IS.UNIQUE with {M/(SUB1 P), X/(SUB1 P)}
                  disable INVERSE.
```

The lemma REMAINDER.PLUS.TIMES.2 cited in the hint for Lemma 6 simply states that when (TIMES J I) is added to X, the remainder upon division by J is unchanged. The system

responded to the hint by inserting the following instance of this lemma as an additional hypothesis of Lemma 6:

```
(EQUAL (REMAINDER (PLUS 1 (TIMES P (SUB1 (SUB1 P)))) P)
       (REMAINDER 1 P)).
```

If the prover had then rediscovered REMAINDER.PLUS.TIMES.2 on its own and applied it as a rewrite rule, then

```
(REMAINDER (PLUS 1 (TIMES P (SUB1 (SUB1 P)))) P)
```

would have been replaced by (REMAINDER 1 P), rendering the above hypothesis useless. Therefore, the instruction was given to disable REMAINDER.PLUS.TIMES.2 (and, for the same reason, Lemma 5).

**3.1.4.** Our next goal is to prove that no integer greater than 1 and less than P-1 can be its own inverse modulo P. The principal ingredient of this proof is the lemma PRIME.KEY.REWRITE, cited earlier, which implies that if $P \mid J^2-1$, then $P \mid J+1$ or $P \mid J-1$. The theorem prover, however, was unable to prove this directly, failing to discover the factorization of $J^2-1$. This was inserted, therefore, as an extra lemma:

```
Lemma 8.  (EQUAL (DIFFERENCE (TIMES X X) 1)
                 (TIMES (ADD1 X) (SUB1 X))).
```

The prover was then able to discover and apply Lemma 8 and PRIME.KEY.REWRITE in establishing

```
Lemma 9.  (IMPLIES (AND (PRIME P)
                        (EQUAL (REMAINDER (DIFFERENCE (TIMES J J) 1) P)
                               0))
                   (OR (EQUAL (REMAINDER (ADD1 J) P) 0)
                       (EQUAL (REMAINDER (SUB1 J) P) 0))).
```

If J is its own inverse and is not divisible by P, then it follows from INVERSE.INVERTS that $J^2 \equiv 1 \pmod{P}$. Hence, $P \mid J^2-1$ and we reach the conclusion of Lemma 9, that $P \mid J+1$ or $P \mid J-1$. The system, however, was unable to construct this argument in one step, requiring assistance in discovering that $J^2 \equiv 1 \pmod{P}$ implies $P \mid J^2-1$:

```
Lemma 10.  (IMPLIES (AND (NOT (LESSP A 1))
                         (EQUAL (REMAINDER A P) 1))
                    (EQUAL (REMAINDER (SUB1 A) P) 0))
           Hints: use EQUAL.MODS.TRICK.2 with (B/1)
                  disable Lemma 8.
```

Lemma 10 was derived as a special case of EQUAL.MODS.TRICK.2, which states that if $A \equiv B \pmod{P}$, then $P \mid A-B$. It was then used in the proof of

```
Lemma 11.  (IMPLIES (AND (PRIME P)
                         (NOT (EQUAL (REMAINDER J P) 0))
                         (EQUAL (INVERSE J P) J))
                    (OR (EQUAL (REMAINDER (ADD1 J) P) 0)
                        (EQUAL (REMAINDER (SUB1 J) P) 0)))
           Hints:  use INVERSE.INVERTS and Lemma 9
                   disable Lemma 8 and INVERSE.
```

We note that Lemma 8, which had served its purpose in the proof of Lemma 9, interfered with the next two events and was, therefore, temporarily disabled.

Finally, using Lemma 11 and observing that no J in the interval $1<J<P-1$ satisfies either $P \mid J+1$ or $P \mid J-1$, the prover was able to establish our goal:

```
Lemma 12.  NO.OTHER.INVOLUTIONS
           (IMPLIES (AND (PRIME P)
                         (LESSP J (SUB1 P))
                         (LESSP 1 J))
                    (NOT (EQUAL (INVERSE J P) J)))
           Hint: use Lemma 11
                 disable INVERSE.
```

**3.1.5.** An essential property of the function INVERSE is that for prime P and $0<J<P$,

(INVERSE (INVERSE J P) P) = J.

The proof of this fact requires two preliminary lemmas. The first of these is simply the algebraic identity

$$(P-2)^2-1 = (P-1)(P-3),$$

which was easily derived as

```
Lemma 13.  (EQUAL (SUB1 (TIMES (DIFFERENCE P 2) (DIFFERENCE P 2)))
                  (TIMES (DIFFERENCE P 3) (SUB1 P))).
```

The other is found in the library file:

```
Lemma.  EXP.MOD.IS.1
        (IMPLIES (EQUAL (REMAINDER (EXP M J) P) 1)
                 (EQUAL (REMAINDER (EXP M (TIMES I J)) P) 1)).
```

With these two lemmas brought to its attention, the theorem prover was able to establish

```
Lemma 14.  INVERSE.OF.INVERSE
           (IMPLIES (AND (PRIME P)
                         (NOT (EQUAL (REMAINDER J P) 0)))
                    (EQUAL (INVERSE (INVERSE J P) P)
                           (REMAINDER J P)))
           Hints:  use Lemma 13
                   use EXP.MOD.IS.1
                       with {M/J, J/(SUB1 P), I/(DIFFERENCE P 3)}.
```

The case P=2 is trivial. For P>2, the proof begins by expanding the definition of INVERSE twice, which leads to

$$\texttt{(INVERSE (INVERSE J P) P)} \equiv J^{(P-2)(P-2)} \equiv J^{(P-1)(P-3)} J \pmod{P}.$$

By FERMAT.THM (which the system applied without prompting) and EXP.MOD.IS.1,

$$J^{(P-1)(P-3)} \equiv 1 \pmod{P}$$

and the result follows.

**3.1.8** We conclude our analysis of INVERSE by showing that if J lies in the interval 0<J<P-1, then so does (INVERSE J P). The upper and lower bounds on (INVERSE J P) were verified separately.

First, the system easily proved that if I is 0 (or not a number), then (INVERSE I P) = 0:

```
Lemma 15. (IMPLIES (AND (ZEROP I) (LESSP 1 P))
                   (EQUAL (INVERSE I P) 0)).
```

If I is replaced by (INVERSE J P), where P∤J, then (INVERSE I P) becomes (INVERSE (INVERSE J P) P), which is (REMAINDER J P) by INVERSE.OF.INVERSE. Thus,

```
Lemma 16.  NON.ZEROP.INVERSE
           (IMPLIES (AND (PRIME P)
                         (NOT (EQUAL (REMAINDER J P) 0)))
                    (NOT (ZEROP (INVERSE J P))))
           Hints:  use Lemma 15 with {I/(INVERSE J P)}
                   use INVERSE.OF.INVERSE
                   disable INVERSE.
```

There is, of course, a more direct approach to Lemma 16, which is to show that if J is positive, then so are all of its powers, including (INVERSE J P). However, this method requires induction and was found to be more difficult for the prover.

The upper bound on (INVERSE J P) was established in three steps. First, the possibility of (INVERSE J P) = (SUB1 P) was disallowed by

```
Lemma 17.  (IMPLIES (AND (PRIME P)
                         (NOT (EQUAL (REMAINDER J P) 0))
                         (EQUAL (INVERSE J P) (SUB1 P)))
                    (EQUAL (REMAINDER J P) (SUB1 P)))
           Hints:  use INVERSE.OF.INVERSE
                   disable INVERSE.
```

The proof of Lemma 17 depends on INVERSE.OF.INVERSE, by which

(REMAINDER J P) = (INVERSE (INVERSE J P) P) = (INVERSE (SUB1 P) P).

At this point, the system discovered SUB1.P.IS.INVOLUTION to complete the proof.

Next, it was observed that (INVERSE J P) can never exceed P-1, i.e.,

```
Lemma 18.  (IMPLIES (LESSP 1 P)
                    (LEQ (INVERSE J P) (SUB1 P))).
```

Finally, combining the last two results, we have

```
Lemma 19.  BOUNDED.INVERSE
           (IMPLIES (AND (PRIME P)
                         (LESSP J (SUB1 P)))
                    (LESSP (INVERSE J P) (SUB1 P))).
           Hints: use Lemmas 17 and 18
                  disable INVERSE
```

## 3.2 The function INVERSE.LIST

For a fixed prime P, (INVERSE J P) may now be viewed as a permutation of the interval $1 \leq J \leq P-1$, fixing only 1 and P-1. By INVERSE.OF.INVERSE and INVERSE.INVERTS, the remaining elements of the interval may be paired off in such a way that the product of each pair is congruent to 1 modulo P. It follows from the elementary properties of congruences that the product (P-1)! of all integers in the interval is congruent to P-1 modulo P. This completes an informal proof of Wilson's theorem.

It is at this point, however, that the formal proof only becomes interesting. The main problem is to formalize the notion of "pairing off" each element with its inverse. This may be achieved by means of

```
Definition 20.  (INVERSE.LIST I P)
                    =
                (IF (ZEROP I)
                    NIL
                    (IF (EQUAL I 1)
                        (CONS 1 NIL)
                        (IF (MEMBER I (INVERSE.LIST (SUB1 I) P))
                            (INVERSE.LIST (SUB1 I) P)
                            (CONS I
                                  (CONS (INVERSE I P)
                                        (INVERSE.LIST (SUB1 I) P))))))).
```

Thus, (INVERSE.LIST 1 P) is the list whose only entry is 1, and for I>1, (INVERSE.LIST I P) is constructed by first recursively computing (INVERSE.LIST (SUB1 I) P) and determining whether this list already contains I; if not, then I and (INVERSE I P) are inserted at the head of the list.

We are interested, in particular, in the value of

```
(INVERSE.LIST (DIFFERENCE P 2) P)
```

Our aim is to show that this list is a permutation of the sequence of integers from 1 to P-2. In this section, we examine those of its properties which are relevant to this goal.

**3.2.1.** First, it is necessary to show that

```
(INVERSE.LIST (DIFFERENCE P 2) P)
```

contains all and only the positive integers less than P-1. The following two lemmas follow easily from the corresponding properties of INVERSE:

```
Lemma 21.  ALL.NON.ZEROP.INVERSE.LIST
           (IMPLIES (AND (PRIME P) (LESSP I (SUB1 P)))
                    (ALL.NON.ZEROP (INVERSE.LIST I P)))
           Hints:  use NON.ZEROP.INVERSE with {J/I}
                   induct according to (INVERSE.LIST I P)
                   disable INVERSE.

Lemma 22.  BOUNDED.INVERSE.LIST
           (IMPLIES (AND (PRIME P)
                         (LESSP I (SUB1 P))
                         (EQUAL J (DIFFERENCE P 2)))
                    (ALL.LESSEQP (INVERSE.LIST I P) J))
            Hints:  use BOUNDED.INVERSE with {J/I}
                    induct according to (INVERSE.LIST I P)
                    disable INVERSE.
```

Thus, each entry I in the list lies in the range $0<I\leq P-2$. In proving both lemmas, the system required the hint to induct according to the scheme suggested by the structure of the definition of (INVERSE.LIST I P), i.e., with base cases (ZEROP I) and I=1 and inductive hypothesis that

the statement holds when I-1 replaces I. The awkward use of the variable J in Lemma 22 could have been avoided, but its presence makes the lemma more useful as a rewrite rule. If the conclusion were

```
(ALL.LESSEQP (INVERSE.LIST I P) (DIFFERENCE P 2))
```

then it would only be automatically applied to terms involving the function DIFFERENCE (see the proof of Lemma 42).

The statement that the list includes all the positive integers through P-2 is formalized by means of the recursive function POSITIVES:

```
Lemma 23.  SUBSETP.POSITIVES
           (SUBSETP (POSITIVES N)
                    (INVERSE.LIST N P)).
```

The prover verified this easily, discovering the appropriate induction without assistance.

We also make the simple observation that 1 is its own inverse:

```
Lemma 24.  INVERSE.1
           (IMPLIES (LESSP 1 P)
                    (EQUAL (INVERSE 1 P) 1)).
```

**3.2.2.** Next, we show that the entries of the list

```
(INVERSE.LIST (DIFFERENCE P 2) P)
```

are pairwise distinct. The definition of INVERSE.LIST suggests a proof by induction on I that the same is true of (INVERSE.LIST I P) for all I<P-1. This amounts to showing that for 1<I<P-1, if (INVERSE.LIST (SUB1 I) P) is a list of distinct entries and I is not among them, then neither is (INVERSE I P). (It should also be noted that I and (INVERSE I P) are distinct by NO.OTHER.INVOLUTION.) This conjecture is further generalized by Lemma 26 below.

```
Lemma 25.  (IMPLIES (AND (PRIME P)
                         (NOT (EQUAL (REMAINDER I P) 0))
                         (LESSP I P)
                         (MEMBER J (INVERSE.LIST I P)))
                    (MEMBER (INVERSE J P) (INVERSE.LIST I P)))
           Hints:  use INVERSE.OF.INVERSE with {J/I}
                   induct according to (INVERSE.LIST I P)
                   disable INVERSE.
```

```
Lemma 26.  (IMPLIES (AND (PRIME P)
                         (NOT (EQUAL (REMAINDER I P) 0))
                         (NOT (EQUAL (REMAINDER J P) 0))
                         (LESSP I P)
                         (LESSP J P)
                         (MEMBER (INVERSE J P) (INVERSE.LIST I P)))
                    (MEMBER J (INVERSE.LIST I P)))
           Hints:  use INVERSE.OF.INVERSE
                   use Lemma 25 with {J/(INVERSE J P)}
                   disable INVERSE, INVERSE.LIST,
                           INVERSE.OF.INVERSE, and Lemma 25.
```

Thus, Lemma 26 asserts that under suitable restrictions on the variables involved, if (INVERSE J P) belongs to (INVERSE.LIST I P), then so does J. Lemma 25, which states the converse, was found to be an easier task for the theorem prover. The proof proceeds by induction on I as follows: suppose that J is a member of (INVERSE.LIST I P). If J already belongs to (INVERSE.LIST (SUB1 I) P), then by the inductive hypothesis, so does (INVERSE J P). Otherwise, J is either I or (INVERSE I P), both of which are members of (INVERSE.LIST I P). In the case J=I, (INVERSE J P) = (INVERSE I P). If J=(INVERSE I P), then by INVERSE.OF.INVERSE, (INVERSE J P)=I. In either case, (INVERSE J P) belongs to (INVERSE.LIST I P))

Lemma 26 is easily derived from Lemma 25, using INVERSE.OF.INVERSE. As noted above, the important result that the entries of (INVERSE.LIST I P) are distinct now follows easily. The theorem prover, however, was unable to verify this directly. It was necessary to introduce the inductive step of this theorem as a preliminary lemma:

```
Lemma 27.  (IMPLIES (AND (PRIME P)
                         (LESSP I (SUB1 P))
                         (ALL.DISTINCT (INVERSE.LIST (SUB1 I) P)))
                    (ALL.DISTINCT (INVERSE.LIST I P)))
           Hints:  use Lemma 26 with {J/I, I/(SUB1 I)}
                   use NO.OTHER.INVOLUTIONS with {J/I}
                   disable INVERSE.
```

Having proved Lemma 27, the system was finally able to verify

```
Lemma 28.  ALL.DISTINCT.INVERSE.LIST
           (IMPLIES (AND (PRIME P)
                         (LESSP I (SUB1 P)))
                    (ALL.DISTINCT (INVERSE.LIST I P)))
           Hints:  use Lemma 27
                   induct according to (POSITIVES I)
                   disable INVERSE.
```

**3.2.3.** The motivation for the definition of INVERSE.LIST was that it results in a list of numbers ordered in such a way that their product is easily computed. The final theorem of this section states that for I<P, the product of the numbers in (INVERSE.LIST I P) is congruent to 1 modulo P.

Here we must make careful use of the properties of REMAINDER and TIMES. We begin with the following established result:

```
Lemma.  TIMES.MOD.3
        (EQUAL (REMAINDER (TIMES (REMAINDER A N) B) N)
               (REMAINDER (TIMES A B) N))
```

Applying TIMES.MOD.3 (with the substitution indicated) and the commutativity of TIMES, the system immediately proved

```
Lemma 29.  (IMPLIES (EQUAL (REMAINDER (TIMES A B) P) 1)
                    (EQUAL (REMAINDER (TIMES A (TIMES B C)) P)
                           (REMAINDER C P)))
           Hints:  use TIMES.MOD.3 with {A/(TIMES A B), B/C, N/P}
                   disable TIMES.MOD.3.
```

If I>1 and I is not already a member of (INVERSE.LIST (SUB1 I) P), then the (previously defined) function TIMES.LIST recursively computes the product of the entries of (INVERSE.LIST I P) as

```
     (TIMES.LIST (INVERSE.LIST I P))
                 =
     (TIMES I (TIMES (INVERSE I P)
                     (TIMES.LIST (INVERSE.LIST (SUB1 I) P)))).
```

Thus, we have, as a special case of Lemma 29,

```
Lemma 30.  (IMPLIES (EQUAL (REMAINDER (TIMES I (INVERSE I P)) P) 1)
                    (EQUAL (REMAINDER (TIMES.LIST (INVERSE.LIST I P)) P)
                           (REMAINDER
                             (TIMES.LIST (INVERSE.LIST (SUB1 I) P)) P)))
           Hints:  use Lemma 29
                       with {A/I, B/(INVERSE I P),
                             C/(TIMES.LIST (INVERSE.LIST (SUB1 I) P))}
                   disable Lemma 29, INVERSE, INVERSE.INVERTS.
```

By INVERSE.INVERTS, it follows that

```
Lemma 31.  (IMPLIES (AND (PRIME P)
                         (NOT (EQUAL (REMAINDER I P) 0)))
                    (EQUAL (REMAINDER (TIMES.LIST (INVERSE.LIST I P))
                                      P)
                           (REMAINDER(TIMES.LIST(INVERSE.LIST(SUB1 I)
                                                              P))
                                     P)))
            Hints:  use INVERSE.INVERTS with {J/I}
                    disable INVERSE, INVERSE.LIST, TIMES.LIST,
                            REMAINDER, and PRIME.
```

Lemma 31 is essentially the inductive step in proving that

```
(REMAINDER (TIMES.LIST (INVERSE.LIST I P)) P) = 1
```

for I<P. The base case is simply

```
Lemma 32.  (IMPLIES (LEQ I 1)
                    (EQUAL (TIMES.LIST (INVERSE.LIST I P)) 1)).
```

Combining the last two results, we have

```
Lemma 33.  TIMES.INVERSE.LIST
           (IMPLIES (AND (PRIME P) (LESSP I P))
                    (EQUAL (REMAINDER (TIMES.LIST (INVERSE.LIST I P)) P)
                           1))
           Hints:  use Lemmas 31 and 32
                   induct according to (POSITIVES I)
                   disable INVERSE, INVERSE.LIST, TIMES.LIST,
                           Lemma 31, and Lemma 32.
```

### 3.3 The pigeon hole principle

For prime P, (INVERSE.LIST (DIFFERENCE P 2) P) is now known to be a list of pairwise distinct positive integers including all and only those which are less than P-1. The main result of this section is that these properties imply that the list is a permutation of (POSITIVES (DIFFERENCE P 2)).

There is a similar result, named PIGEON.HOLE.PRINCIPLE, which is found in the file RSA.LIB. This theorem is not applicable to the problem at hand because it contains a hypothesis concerning the length of the list. Apparently, there is no simple way to compute the length of any list constructed using INVERSE.LIST.

The next six lemmas all deal with list manipulations. The functions concerned all have similarly structured definitions, each recursing on the CDR of one of its arguments. In this situation, the theorem prover's heuristics lead unambiguously to an appropriate induction scheme. The proofs of these lemmas were discovered without any hints.

```
Lemma 34.  (IMPLIES (AND (MEMBER A S) (NOT (EQUAL A X)))
                    (MEMBER A (DELETE X S))).

Lemma 35.  (IMPLIES (AND (SUBSETP R S) (NOT (MEMBER X R)))
                    (SUBSETP R (DELETE X S))).

Lemma 36.  (IMPLIES (AND (ALL.DISTINCT L) (ALL.LESSEQP L N))
                    (ALL.LESSEQP (DELETE N L) (SUB1 N))).
```

```
Lemma 37.  (IMPLIES (LESSP N M)
                    (NOT (MEMBER M (POSITIVES N)))).

Lemma 38.  (IMPLIES (SUBSETP (POSITIVES N) L)
                    (SUBSETP (POSITIVES (SUB1 N)) (DELETE N L))).

Lemma 39.  (IMPLIES (AND (ZEROP N)
                         (ALL.LESSEQP L N)
                         (ALL.NON.ZEROP L))
                    (NOT (LISTP L))).
```

The following function definition is introduced with no intention of ever evaluating the function, but rather as a means of teaching the system a new induction scheme to be used in the next proof:

```
Definition 40.  (PIGEONHOLE2.INDUCTION L N)
                              =
                (IF (ZEROP N)
                    T
                    (PIGEONHOLE2.INDUCTION (DELETE N L) (SUB1 N))).

Lemma 41.  PIGEONHOLE2
           (IMPLIES (AND (ALL.DISTINCT L)
                         (ALL.NON.ZEROP L)
                         (ALL.LESSEQP L N)
                         (SUBSETP (POSITIVES N) L))
                    (PERM (POSITIVES N) L))
            Hint:  induct according to (PIGEONHOLE2.INDUCTION L N).
```

The induction indicated involves one base case, (ZEROP N), and one inductive hypothesis, obtained by the substitutions L/(DELETE N L) and N/(SUB1 N).

In the base case, we observe that (POSITIVES N)=NIL and hence (PERM (POSITIVES N) L) reduces to (NLISTP L). This is true by Lemma 39.

For the inductive step, we assume that all the hypotheses of the lemma hold. Two theorems from RSA.LIB, ALL.DISTINCT.DELETE and ALL.NON.ZEROP.DELETE, then imply (ALL.DISTINCT (DELETE N L)) and (ALL.NON.ZEROP (DELETE N L)), respectively. Lemma 36 yields (ALL.LESSEQP (DELETE N L) (SUB1 N)), and from Lemma 38 we have (SUBSETP (POSITIVES (SUB1 N)) (DELETE N L). We now infer the conclusion of the inductive hypothesis, (PERM (POSITIVES (SUB1 N)) (DELETE N L)), which may be written (PERM (CDR (POSITIVES N)) (DELETE N L)). The hypothesis (SUBSETP (POSITIVES N) L) implies (MEMBER (CAR (POSITIVES N)) L), and the lemma follows.

### 3.4 The final result

For L=(INVERSE.LIST (DIFFERENCE P 2) P) and N=(DIFFERENCE P 2), the hypotheses of PIGEONHOLE2 are among the results of section 3.2. We conclude, therefore,

```
Lemma 42.  PERM.POSITIVES.INVERSE.LIST
           (IMPLIES (AND (PRIME P) (EQUAL I (DIFFERENCE P 2)))
                    (PERM (POSITIVES I) (INVERSE.LIST I P))).
```

The library theorem TIMES.LIST.EQUAL.FACT states that if L is a permutation of (POSITIVES N), then (TIMES.LIST L)=(FACT N). Combining this with PERM.POSITIVES.INVERSE.LIST, we have

```
Lemma 43.  INVERSE.LIST.FACT
           (IMPLIES (AND (PRIME P) (EQUAL I (DIFFERENCE P 2)))
                    (EQUAL (TIMES.LIST (INVERSE.LIST I P)) (FACT I)))
           Hints:  use TIMES.LIST.EQUAL.FACT
                       with {N/I, L/(INVERSE.LIST I P)}
                   disable INVERSE.LIST.
```

Next, we recall TIMES.INVERSE.LIST, which now yields

```
Lemma 44.  (IMPLIES (AND (PRIME P)
                         (EQUAL I (DIFFERENCE P 2)))
                    (EQUAL (REMAINDER (FACT I) P) 1))
           Hint:  use TIMES.INVERSE.LIST.
```

Finally, multiplying the congruence of Lemma 44 by P-1, we have

```
Lemma 45.  WILSON.THM
           (IMPLIES (PRIME P)
                    (EQUAL (REMAINDER (FACT (SUB1 P)) P) (SUB1 P)))
           Hints:  use Lemma 44 with {I/(SUB1 (SUB1 P))}
                   use THM.55.SPECIALIZED.TO.PRIMES
                       with {M/(SUB1 P), X/(FACT (SUB1 (SUB1 P))),
                             Y/1}
                   disable Lemma 44 and THM.55.SPECIALIZED.TO.PRIMES.
```

## 4 Conclusions

In a lecture delivered to the 1982 Joint International Seminar on the Teaching of Computer Science, P. M. Melliar-Smith [5] remarked that

> A drawback to heuristic theorem proving attempts is that successful proof depends upon intimate knowledge of the heuristics employed. One must understand how very subtle changes in specification structure, even those that preserve semantic equivalence, can affect the direction and final outcome of the proof attempt. Lemma form becomes as important as content.

To some extent, the Boyer-Moore system is susceptible to this criticism. A familiarity with the structure of the theorem proving program is probably necessary for success in verifying any nontrivial theorem. However, the expertise required of the user is less than might be expected. The author, a mathematician acquainted with the logical theory involved but with no prior experience in automatic theorem proving, executed this project in three phases, each occurring within a period of one week. First, a modest understanding of the system's heuristics and current library was gained by reading [4] and Chapters V–XIII of [2]. Next, a sequence of twenty-five lemmas and function definitions was prepared for submission to the theorem prover. Informal proofs were constructed in an attempt to predict the prover's behavior. Finally, this list of events was presented to the system. Some lemmas were verified immediately, others required modification (e.g., additional hints), and new lemmas were inserted where necessary.

The responses of the theorem prover were generally found to be difficult to predict. The analysis of proof attempts was often so complicated that the effects of minor changes in conjectures could only be determined empirically. In many cases, hints were inserted liberally and somewhat arbitrarily until a lemma was finally proved. Upon subsequent examination of the system's output, it was generally possible to explain why each hint or extra step was warranted.

There is no doubt that a more "intimate knowledge of the heuristics employed" by a theorem prover would allow its user greater efficiency. However, it is possible for a novice in the field to enjoy some success. The verification of meaningful results, which might be impractical with a system based solely on decision procedures, may be achieved relatively easily with a heuristic theorem prover with only a limited understanding of its design.

# Appendix A.

# A list of system functions

Here are listed some function definitions from RSA.LIB which were used in the proof of Wilson's theorem.

## A.1 Arithmetic functions

```
1.  (ZEROP X)
       =
    (OR (EQUAL X 0) (NOT (NUMBERP X)))

2.  (LESSP X Y)
       =
    (IF (ZEROP Y)
        F
        (IF (ZEROP X)
            T
            (LESSP (SUB1 X) (SUB1 Y))))

3.  (LEQ X Y)
       =
    (NOT (LESSP Y X))

4.  (FIX X)
       =
    (IF (NUMBERP X) X 0)

5.  (PLUS X Y)
       =
    (IF (ZEROP X)
        (FIX Y)
        (ADD1 (PLUS (SUB1 X) Y)))

6.  (TIMES I J)
       =
    (IF (ZEROP I)
        0
        (PLUS J (TIMES (SUB1 I) J)))
```

```
7.  (DIFFERENCE I J)
        =
    (IF (ZEROP I)
        0
        (IF (ZEROP J)
            I
            (DIFFERENCE (SUB1 I) (SUB1 J))))

8.  (REMAINDER I J)
        =
    (IF (ZEROP J)
        (FIX I)
        (IF (LESSP I J)
            (FIX I)
            (REMAINDER (DIFFERENCE I J) J)))

9.  (EXP I J)
        =
    (IF (ZEROP J)
        1
        (TIMES I (EXP I (SUB1 J))))

10.  (FACT I)
        =
     (IF (ZEROP I)
         1
         (TIMES I (FACT (SUB1 I))))

11.  (DIVIDES X Y)
        =
     (ZEROP (REMAINDER Y X))

12.  (PRIME1 X Y)
        =
     (IF (ZEROP Y)
         F
         (IF (EQUAL Y 1)
             T
             (AND (NOT (DIVIDES Y X))
                  (PRIME1 X (SUB1 Y)))))

13.  (PRIME X)
        =
     (AND (NOT (ZEROP X))
          (NOT (EQUAL X 1))
          (PRIME1 X (SUB1 X)))
```

## A.2 List-related functions

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

15.  (SUBSETP X Y)
       =
     (IF (LISTP X)
         (IF (MEMBER (CAR X) Y)
             (SUBSETP (CDR X) Y)
             F)
         T)

16.  (NLISTP L)
       =
     (NOT (LISTP L))

17.  (DELETE X L)
       =
     (IF (NLISTP L)
         L
         (IF (EQUAL X (CAR L))
             (CDR L)
             (CONS (CAR L) (DELETE X (CDR L)))))

18.  (PERM A B)
       =
     (IF (NLISTP A)
         (NLISTP B)
         (AND (MEMBER (CAR A) B)
              (PERM (CDR A) (DELETE (CAR A) B))))

19.  (TIMES.LIST L)
       =
     (IF (NLISTP L)
         1
         (TIMES (CAR L) (TIMES.LIST (CDR L))))

20.  (POSITIVES N)
       =
     (IF (ZEROP N)
         NIL
         (CONS N (POSITIVES (SUB1 N))))

21.  (ALL.NON.ZEROP L)
       =
     (IF (NLISTP L)
         T
         (AND (NOT (ZEROP (CAR L)))
              (ALL.NON.ZEROP (CDR L))))
```

```
22.  (ALL.LESSEQP L N)
            =
     (IF (NLISTP L)
         T
         (AND (LEQ (CAR L) N)
              (ALL.LESSEQP (CDR L) N)))

23.  (ALL.DISTINCT L)
            =
     (IF (NLISTP L)
         T
         (AND (NOT (MEMBER (CAR L) (CDR L)))
              (ALL.DISTINCT (CDR L))))
```

# Bibliography


[1] George Boole.
*An Investigation of the Laws of Thought on which are Founded the Mathematical Theories of Logic and Probabilities.*
Dover Publications, New York, 1951.

[2] Robert S. Boyer and J Strother Moore.
*A Computational Logic.*
Academic Press, New York, 1979.

[3] Robert S. Boyer and J Strother Moore.
A Theorem Prover for Recursive Functions, a User's Manual.
Computer Science Laboratory, SRI International, 1979.

[4] Robert S. Boyer and J Strother Moore.
*Proof Checking the RSA Public Key Encryption Algorithm.*
Technical Report ICSCA-CMP-33, Institute for Computing Science and Computer Applications, University of Texas at Austin, 1982.

[5] Edsger W. Dijkstra.
Trip Report, Newcastle-upon-Tyne.
Sept. 6-10, 1982.

[6] G. H. Hardy and E. M. Wright.
*An Introduction to the Theory of Numbers.*
Oxford University Press, 1968.

[7] John McCarthy.
Computer Programs for Checking Mathematical Proofs.
*Recursive Function Theory, Proc. Symp. Pure Math., Amer. Math. Soc.* V:219-227, 1962.

[8] John McCarthy.
A Basis for a Mathematical Theory of Computation.
In P. Braffort and D. Hershberg (editor), *Computer Programming and Formal Systems*, pages 33-70. North-Holland Publ. Co., Amsterdam, 1963.