A Mechanical Proof of the Turing Completeness of Pure Lisp

Robert S. Boyer and J Strother Moore

Technical Report #37 May 1983

Institute for Computing Science
2100 Main Building
The University of Texas at Austin
Austin, Texas 78712
(512) 471-1901

ABSTRACT

We describe a proof by a computer program of the Turing completeness of a computational paradigm akin to Pure LISP. That is, we define formally the notions of a Turing machine and of a version of Pure LISP and prove that anything that can be computed by a Turing machine can be computed by LISP. While this result is straightforward, we believe this is the first instance of a machine proving the Turing completeness of another computational paradigm.

The work here was supported in part by NSF Grant MCS-8202943 and ONR Contract N00014-81-K-0634.

1. Introduction. In our paper [Boyer & Moore 84] we present a definition of a function EVAL that serves as an interpreter for a language akin to Pure LISP, and we describe a mechanical proof of the unsolvability of the halting problem for this version of LISP. We claim in that paper that we have proved the recursive unsolvability of the halting problem. It has been pointed out by a reviewer that we cannot claim to have mechanically proved the recursive unsolvability of the halting problem unless we also mechanically prove that the computational paradigm used is as powerful as partial recursive functions. To this end we here describe a mechanically checked proof that EVAL is at least as powerful as the class of Turing machines. The proof was constructed by a descendant of the systems described in [Boyer & Moore 79] and [Boyer & Moore 81]. For an informal introduction to our system see [Boyer&Moore 84], in this volume.

One of our motivations in writing this paper is to add further support to our conjecture that, with the aid of automatic theorem-provers, it is now often practical to do mathematics formally. Our proof of the completeness theorem is presented as a sequence of lemmas, stitched together by some English sentences motivating or justifying the lemmas. Our description of the proof is at about the same level it would be in a careful informal presentation of Turing completeness. However, every formula exhibited as a theorem in this document has actually been proved mechanically. In essence, our machine has read the informal proof and assented to each step (without bothering with the motivations). Thus, while the user of the machine was describing an informal proof at a fairly high level, the machine was filling in the often large gaps.

In the next section we briefly describe the Pure LISP programming language, our formal logic, and our use of the logic to formalize Pure LISP. These topics are covered more thoroughly in [Boyer & Moore 84] where we prove the unsolvability of the halting problem. We then sketch our formalization of Turing machines and we give our formal statement of the Turing completeness of Pure LISP. The remainder of the document is an extensively annotated list of definitions and lemmas leading our theorem-prover through the formalization of Turing machines and the proof of the completeness result.

2. A Sketch of Our Formalization of Pure LISP.

2.1. *A Quick Introduction to LISP*

LISP is a programming language originally defined by McCarthy in [McCarthy 65]. Pure LISP is a subset of LISP akin to the lambda calculus [Church 41]. Pure LISP has no operations that have ‘‘side-effects.’’ For example, there is no way in Pure LISP to ‘‘change’’ the ordered pair <1,2> into the ordered pair <2,1>. The version of Pure LISP we describe here differs from McCarthy’s version primarily by its omission of ‘‘functional arguments’’ and ‘‘global variables.’’ We are here concerned entirely with our version of Pure LISP, which henceforth we will simply refer to as LISP.

The syntax of LISP is very similar to that of the lambda calculus. Program names and variable symbols are simply words, such as `plus`, `x`, and `y`. LISP expressions (sometimes called s-expressions) are formed from symbols grouped together with parentheses. A variable symbol is an s-expression. If *f* is the name of a LISP program and *x* and *y* are s-expressions then `(`*f x y*`)` is an s-expression denoting the application of the program named *f* to the values of *x* and *y*. Thus, `(plus x y)` is an s-expression. Because `plus` is the name of the addition operation, `(plus x y)` is the LISP notation for `x+y`. The syntax of LISP also provides a notation for writing down certain constants.

LISP provides a variety of data structures. In addition to the natural numbers, the two most commonly used data types are ‘‘atoms’’ and ordered pairs. Atoms are words. The properties of atoms are largely unimportant here, except that two atoms are equal if and only if they have the same spelling. Atomic constants are usually written in ‘‘`quote` notation.’’ For example, `(quote abc)` is a LISP expression whose value is the word `abc`. Observe that `quote` is not like other LISP programs. If *f* is the name of a LISP program of one argument, and *f* is not `quote`, the value of the LISP expression `(`*f* `abc)` is obtained by running the program named *f* on the value of the variable named `abc`; if *f* is

quote, the value of (*f*abc) is the atom abc.

Given two LISP objects, *x* and *y*, it is possible to construct the ordered pair <*x*,*y*>. quote notation may be used to denote ordered pairs. For example, (quote (1 . 2)) is a LISP expression whose value is the ordered pair <1,2>. Similarly, (quote (abc . 0)) evaluates to <abc,0>.

In LISP, lists of objects are represented with ordered pairs and the atom nil. The empty list is represented by nil. The list obtained by adding the element *x* to the list *y* is represented by <*x*,*y*>. Thus <3,nil> represents the singleton list containing 3. quote notation treats specially pairs whose second component is nil; for example, (quote (3 . nil)) can also be written (quote (3)). The list containing successively 1, 2, 3, and no other element is represented by the nest of three ordered pairs <1,<2,<3,nil>>>, and is typically written (quote (1 2 3)).

LISP provides primitive programs for manipulating such objects. For example, the program cons, when applied to two objects *x* and *y*, returns the ordered pair <*x*,*y*>. The programs car and cdr extract the two components of a pair.

Complex manipulations on such data structures are described by defining recursive programs. For example, the program app, which takes two lists and constructs their concatenation, can be defined as follows:

```
(app (lambda (x y)
       (if (equal x (quote nil))
           y
           (cons (car x) (app (cdr x) y))))).
```

This definition is interpreted as follows. Suppose it is desired to apply the program app to two objects *x* and *y*. We first ask whether *x* is the atom nil. If so, the final answer is *y*. Otherwise, we construct the final answer by applying cons to two objects. The first object is obtained by applying car to *x* and is thus the first element of the list *x*. The second object is obtained by recursively applying app to the remaining elements of *x* and *y*. Thus, (app (quote (a b c)) (quote (1 2 3))) computes out to the same thing as (quote (a b c 1 2 3)).

It is conceivable that the computation thus described never terminates. For example, in our LISP applying cdr to the non-pair 0 produces 0. Thus the execution of (app 0 (quote nil)) examines the successive cdr's of 0, app never encounters nil, and the computation ''runs forever.''

We wish to prove a theorem about what can be computed by LISP. We must therefore be precise in our definition of LISP. One way to specify a programming language is to define mathematically an interpreter for the language. That is, we might define a function that determines the value of every expression in the language. However, the value of an arbitrary s-expression, e.g., (app a b), is a function of the values of the variables in it, e.g., a and b, and the definitions of the program names used, e.g., app. Thus the LISP interpreter can be thought of as a function with three inputs: an s-expression, a mapping from variable names to values, and a mapping from program names to definitions. The interpreter either returns the value of the expression or indicates that the computation does not terminate.

2.2. *A Quick Introduction to Our Logic*

Since we wish to prove our theorem formally (indeed, mechanically), we must define the interpreter in a formal logic. The logic we use is that described in [Boyer & Moore 79].

In what is perhaps a confusing turn of events, the logic happens to be inspired by Pure LISP. For example, to denote the application of the function symbol PLUS to the variable symbols X and Y we write (PLUS X Y) where others might write PLUS(X,Y) or even X+Y.

Axioms provide for a variety of ''types.'' For example, the constants T and F are axiomatized to be distinct. The function EQUAL is axiomatized so that (EQUAL X Y) is T if X=Y and is F if X#≠#Y. A set of axioms similar to the

Peano axioms describes the natural numbers as being inductively constructed from the constant 0 by application of the successor function, named ADD1. Another set of axioms describes the literal atoms as being constructed by the application of the function PACK to objects describing the ‘‘print names’’ of the atoms. Similarly, a set of axioms describes the ordered pairs as being inductively constructed from all other objects by application of the CONS function. One of those axioms is (CAR (CONS X Y)) = X, which specifies the behavior of the function CAR on ordered pairs. Another axiom specifies the function LISTP to return T when given an ordered pair and F otherwise.

We use a notation similar to LISP's quote notation to write down constants. For example, 97 is an abbreviation for a nest of ninety-seven ADD1's around the constant 0. '(97 112 112 . 0) is our abbreviation for the list constant

(CONS 97 (CONS 112 (CONS 112 0))).

We have arbitrarily adopted the ASCII [--- 77] standard as a convention for associating numbers with characters and thus agree that 'app is an abbreviation for the constant term:

(PACK '(97 112 112 . 0)).

'NIL thus abbreviates:

(PACK (CONS 78 (CONS 73 (CONS 76 0)))).

(LIST t_1 t_2 ... t_n) abbreviates

(CONS t_1
(CONS t_2
...
(CONS t_n **'NIL)...)).**

Thus, (LIST A B C) is (CONS A (CONS B (CONS C 'NIL))).

The logic provides for the recursive definition of functions, provided certain conditions are met insuring that there exist one and only one function satisfying the definitional equation. For example, the user of the logic can define the function APP with the following equation:

```
(APP X Y)
   =
(IF (LISTP X)
    (CONS (CAR X) (APP (CDR X) Y))
    Y).
```

The definitional principle admits this equation because it can be proved that a certain measure of the arguments decreases in a well-founded sense in the recursion. From these and other syntactic observations it can be shown that one and only one function satisfies this equation. Once admitted, the equation is regarded as a new axiom in which X and Y are implicitly universally quantified.

Various theorems can then be proved about APP. For example, it is a theorem that (APP '(A B C) '(1 2 3)) = '(A B C 1 2 3). It is also a theorem that (APP (APP X Y) Z) = (APP X (APP Y Z)). The latter theorem requires induction to prove.

The reader interested in the details of the logic should see [Boyer & Moore 79, Boyer & Moore 81].

We wish to use the logic to formalize LISP. We will thus be using formulas in the logic to discuss programs in LISP. We will use logical constants to represent particular LISP expressions. For example, we will use literal atom constants in the logic, such as 'app, 'a, and 'b, to represent LISP program names and variables. We will use list constants, such as '(app a b) and '(quote (a b c)), to represent non-variable LISP s-expressions.

Since the logic and LISP have very similar syntax, use of the one to discuss the other may be confusing. We therefore have followed four conventions. First, instead of displaying LISP s-expressions, e.g., `(app a b)`, we will henceforth display only the logical constants we use to represent them, e.g., `'(app a b)`. Second, despite the practice common to LISP programmers of referring to their programs as ''functions,'' we are careful here to use the word *program* when referring to a LISP program and reserve the word *function* for references to the usually understood mathematical concept. Third, we display all constants denoting LISP s-expressions in lower case. Fourth, variable symbols in the logic that intuitively take s-expressions for values are usually written in lower case. For perfect syntactic consistency with [Boyer & Moore 84], all lower case letters in formulas should be read in upper case.

Thus, `'(app a b)` is a LISP s-expression and `'app` is the name of a LISP program; `(APP A B)` is a term in the logic and `APP` is the name of a function.

2.3. *The Function* `EVAL`

In [Boyer & Moore 84] we formalize LISP in the logic sketched above. We will sketch here that formalization.

The ideal LISP interpreter is a function that takes three inputs: an s-expression `X`, a mapping `VA` that associates variable symbols with values, and a mapping `PA` that associates program symbols with their definitions. The ideal interpreter either returns the value of `X` under `VA` and `PA` or it returns an indication that the computation never terminates. We would like to define such a function under the principle of definition in our logic. However, the principle requires that when `(`*f* `X VA PA)` = *body* is submitted as a definition, there exist a measure of <X,VA,PA> that decreases in a well-founded sense in every call of *f* in *body*. This prevents us from directly defining the ideal interpreter as a function.

Instead we define the function `EVAL` which takes four arguments, `X`, `VA`, `PA`, and `N`. `N` can be thought of as the amount of ''stack space'' available to the interpreter; that is, `N` is the maximum depth of recursion available to interpreted LISP programs defined on `PA`. `(EVAL X VA PA N)` returns either the value of `X` under `VA` and `PA`, or else it returns the logical constant `(BTM)` to denote that it ran out of stack space.

`(BTM)` is axiomatized as an object of a ''new'' type; it is not equal to `T`, `F`, any number, literal atom, or list. `(BTM)` is recognized by the function `BTMP`, which returns `T` or `F` according to whether its argument is equal to `(BTM)`.

The mappings associating variable and program names with values and definitions are represented by lists of pairs, called *association lists* or simply *alists*. For example, the logical constant:

```
'((a . 1) (b . 2) (c . 3))
```

is an alist in which the literal atom `'a` has the value `1`, `'b` has the value `2`, `'c` has the value `3`, and all other objects have the value `(BTM)`, by default. The logical function `GET` is defined to take an object and an alist and return the associated value of the object in the alist.

Here is a brief sketch of the definition of `(EVAL x VA PA N)`. If `x` is a built-in LISP constant, such as `'t`, `'f`, `'nil`, or `97`, its value is `T`, `F`, `NIL`, or `97`, as appropriate. If `x` is a variable symbol, its value is looked up in `VA`. If `x` is of the form `'(quote` *v*`)`, its value is *v*. If `x` is of the form `'(if` *test true-value false-value*`)`, then *test* is evaluated; if the value of *test* is `(BTM)`, then `(BTM)` is returned, if the value is `F`, then the value of *false-value* is returned, and otherwise the value of *true-value* is returned. If the function symbol of `x` is a primitive, e.g., `cons` or `car`, the value of `x` is determined by applying the appropriate function, e.g., `CONS` or `CAR`, to the recursively obtained values of the arguments, unless `(BTM)` is among those values, in which case the answer is `(BTM)`. Otherwise, `x` is of the form `'(`$f\ a_1 \ldots a_n$`)`, where *f* is supposedly defined on `PA`. If `N` is 0 or *f* is not found on `PA`, the value of `x` is `(BTM)`. Otherwise, the definition of *f* specifies a list of parameters names `'(`$x_1 \ldots x_n$`)` and an s-expression *body*. `EVAL` recursively evaluates the arguments, $a_1, \ldots, a_n$ to obtain *n* values, $v_1, \ldots, v_n$. If `(BTM)` is among the values, the value of `x` is `BTM`. Otherwise, `EVAL` constructs a new variable alist associating each parameter name x_i with the corresponding

value v_i, and then recursively evaluates *body* under the new alist, the same PA, and N-1.

We now illustrate EVAL. Let VA be an alist in which 'a has the value '(a b c), 'a0 has the value '(a b c . 0), and 'b has the value '(1 2 3).

```
VA = '((a . (a b c))
       (a0 . (a b c . 0))
       (b . (1 2 3))).
```

Let PA be an alist in which the atom 'app is defined as shown below:

```
PA = '((app (x y)
            (if (equal x nil)
                y
                (cons (car x) (app (cdr x) y))))).
```

Then here are two theorems about EVAL:

```
(EVAL '(app a b) VA PA N) = (IF (LESSP N 4)
                                (BTM)
                                '(a b c 1 2 3)),
(EVAL '(app a0 b) VA PA N) = (BTM).
```

The proof of the second theorem depends on the fact that the CDR of 0 is 0, which is not NIL.

We do not give the formal definition of EVAL here. The interested reader should see [Boyer & Moore 84].

The informal notion that the computation of X under VA and PA ''terminates'' is formalized by ''there exists a stack depth N such that (EVAL x VA PA N) #≠# (BTM).'' Conversely, the notion that the computation ''runs forever'' is formalized by ''for all N (EVAL x VA PA N) = (BTM).'' Thus, we see that the computation of '(app a b) under the VA and PA above terminates, but the computation of '(app a0 b) under VA and PA does not. Since our logic does not contain quantifiers, we will have to derive entirely constructive replacements of these formulas to formulate the Turing completeness of EVAL.

3. Turing Machines. Our formalization of Turing machines follows the description by Rogers [Rogers 67]. The ideal Turing machine interpreter takes three arguments: a state symbol ST, an infinite tape TAPE (with a distinguished ''cell'' marking the initial position of the Turing machine), and a Turing machine TM. If the computation terminates, the Turing machine interpreter returns the final tape. Otherwise, the ideal interpreter runs forever.

We cannot define the ideal interpreter directly as a function in our logic. Instead, we define the function TMI of four arguments, ST, TAPE, TM, and K. The fourth argument is the maximum number of state transitions permitted. TMI returns either the final tape (if the machine halts after executing fewer than K instructions) or (BTM). We say the computation of TM (starting in state ST) on TAPE ''terminates'' if ''there is a K such that (TMI ST TAPE TM K) #≠# (BTM)'' and ''runs forever'' if ''for all K (TMI ST TAPE TM K) = (BTM).''

The primary difference between our formalization of Turing machines and Rogers' description is that we limit our attention to infinite tapes containing only a finite number of 1's. This is acceptable since Rogers uses only such tapes to compute the partial recursive functions (page 14 of [Rogers 67]):

> Given a Turing machine, a tape, a cell on that tape, and an initial internal state, the Turing machine carries out a uniquely determined succession of operations, which may or may not terminate in a finite number of steps. We can associate a partial function with each Turing machine in the following way. To represent an input integer x, take a string of x+1 consecutive 1's on the tape. Start the machine in state q0 on the leftmost cell containing a 1. As output integer, take the total number of 1's appearing anywhere on the tape when (and if) the machine stops.

Our conventions for formalizing states, tapes, and Turing machines are made clear in Section .

4. The Statement of the Turing Completeness of LISP. We wish to formulate the idea that LISP can compute anything a Turing machine can compute. Roughly speaking we wish to establish a correspondence between Turing machines, their inputs, and their outputs on the one hand, and LISP programs, their inputs, and their outputs on the other. Then we wish to prove that (a) if a given Turing machine runs forever on given input, then so does its LISP counterpart; and (b) if a given Turing machine terminates on given input, then its LISP counterpart terminates with the same answer.

We will set up a correspondence between Turing machines and program alists (since it is there that LISP programs are defined). The mapping will be done by the function TMI.PA, which takes a Turing machine and returns a corresponding LISP program. We will also set up a correspondence between the state and tape ''inputs'' of the Turing machine and the s-expression being evaluated by LISP, since the s-expression can be thought of as the ''input'' to the LISP program defined on the program alist. That mapping is done by the function TMI.X. In our statement of the completeness theorem, we use the NIL variable alist.

Since the value of a terminating Turing machine computation is a tape, and tapes are formalized as list structures that can be manipulated by LISP programs, we will require that the corresponding LISP computation deliver the list structure identical to the final tape.

Our informal characterization of the Turing completeness of LISP is then as follows. Let TM be a Turing machine, ST a state, and TAPE a tape. Let PA_{TM} be the corresponding LISP program (i.e., the result of (TMI.PA TM)), and let $X_{ST,TAPE}$ be the corresponding LISP expression (i.e., the result of (TMI.X ST TAPE)). Then

(a) If the Turing machine computation of TM (starting in ST) on TAPE runs forever, then so does the LISP computation of $X_{ST,TAPE}$ under the NIL variable alist and the program alist PA_{TM}.

(b) If the Turing machine computation of TM (starting in ST) on TAPE terminates with tape Z, then the LISP computation of $X_{ST,TAPE}$ under NIL and PA_{TM} terminates with result Z.

To formalize (a) we replace the informal notions of ''runs forever'' by the formal ones. The result is:

$$\forall\ \mathbf{K\ (TMI\ ST\ TAPE\ TM\ K)=(BTM)}$$
$$\#\rightarrow\#$$
$$\forall\ \mathbf{N\ (EVAL\ X_{ST,TAPE}\ NIL\ PA_{TM}\ N)=(BTM).}$$

This is equivalent to:

$$\exists\ \mathbf{N\ (EVAL\ X_{ST,TAPE}\ NIL\ PA_{TM}\ N)\ \#\neq\#\ (BTM)}$$
$$\#\rightarrow\#$$
$$\exists\ \mathbf{K\ (TMI\ ST\ TAPE\ TM\ K)\ \#\neq\#\ (BTM).}$$

The existential quantification on N in the hypothesis can be transformed into implicit universal quantification on the outside. The existential quantification on K in the conclusion can be removed by replacing K with a term that delivers a suitable K as a function of all the variables whose (implicit) quantifiers govern the occurrence of K. The result is:

$$\mathbf{(a')\ (EVAL\ X_{ST,TAPE}\ NIL\ PA_{TM}\ N)\#\neq\#\ (BTM)}$$
$$\#\rightarrow\#$$
$$\mathbf{(TMI\ ST\ TAPE\ TM\ (TMI.K\ ST\ TAPE\ TM\ N))\#\neq\#\ (BTM).}$$

That is, we prove that a suitable K exists by constructively exhibiting one.

To formalize part (b) observe that it is equivalent to

If the computation of TM (starting in ST) on TAPE terminates within K steps with tape Z, then there exists an N such that $Z = (\mathrm{EVAL}\ X_{ST,TAPE}\ \mathrm{NIL}\ PA_{TM}\ \mathrm{N})$.

Thus, (b) becomes:

```
(TMI ST TAPE TM K) #≠# (BTM)
     #→#
∃ N (TMI ST TAPE TM K) = (EVAL X_ST,TAPE NIL PA_TM N).
```

We remove the existential quantification on N by replacing N with a particular term, (TMI.N ST TAPE TM K), that delivers an appropriate N as a function of the other variables.

Our formal statement of the Turing completeness of LISP is:

```
Theorem.  TURING.COMPLETENESS.OF.LISP:
(IMPLIES (AND (STATE ST)
              (TAPE TAPE)
              (TURING.MACHINE TM))
         (AND (IMPLIES (NOT (BTMP (EVAL (TMI.X ST TAPE)
                                        NIL
                                        (TMI.PA TM)
                                        N)))
                       (NOT (BTMP (TMI ST TAPE TM
                                       (TMI.K ST TAPE
                                              TM N)))))
              (IMPLIES (NOT (BTMP (TMI ST TAPE TM K)))
                       (EQUAL (TMI ST TAPE TM K)
                              (EVAL (TMI.X ST TAPE)
                                    NIL
                                    (TMI.PA TM)
                                    (TMI.N ST TAPE
                                           TM K)))))).
```

Before proving this theorem we must define the functions TMI.X, TMI.PA, TMI.K and TMI.N.

We find the above formal statement of the completeness result to be intuitively appealing. However, the reader should study it to confirm that it indeed captures the intuitive notion. Some earlier formalizations by us were in fact inadequate. For example, one might wonder why we talk of the correspondences set up by TMI.X and TMI.PA instead of stating part (b) simply as:

> For every Turing machine TM, there exists a program alist PA, such that for every state ST and tape TAPE, there exists an X such that: if the computation of TM (starting in ST) on TAPE terminates with tape Z then the computation of X under NIL and PA terminates with Z.

However, observe that the existentially quantified X -- the expression EVAL is to evaluate -- can be chosen after TM, ST, and TAPE are fixed. Thus, a suitable X could be constructed by running the ideal Turing machine interpreter on the given TM, ST, and TAPE until it halts (as it must do by hypothesis), getting the final tape, and embedding it in a 'quote form. Thus, when EVAL evaluates X it will return the final tape. Indeed, an EVAL that only knew how to interpret 'quote would satisfy part (b) of the above statement of Turing completeness! Rearranging the quantifiers provides no relief. If the quantification on PA is made the innermost, one can appeal to a similar trick and embed the correct final answer in a 'quote form inside the definition of a single dummy program.

Our statement does not admit such short cuts. Since TMI.PA is a function only of TM, one must invent a suitable LISP program from TM without knowing what its input will be. Since TMI.X is a function only of ST and TAPE, one must invent a suitable input for the LISP program without knowing which Turing machine will run it.

Proving the Turing completeness of EVAL is not entirely academic. The definition of EVAL requires about two pages. Its correctness as a formalization of Pure LISP is not nearly as obvious as the correctness of our formalization of Turing machines, which requires only half a page of definitions. The completeness result will assure us that EVAL -- whether a formalization of LISP or not -- is as powerful as the Turing machine paradigm.

For our proof of the Turing completeness of LISP we define (TMI.PA TM) to be a program alist that contains a LISP encoding of the ideal Turing machine interpreter together with a single dummy program that returns the Turing machine to be interpreted. Our definition of (TMI.X ST TAPE) returns the LISP s-expression that calls (the LISP version of) the ideal Turing machine interpreter on ST, TAPE, and the Turing machine TM. The heart of our proof is a ‘‘correspondence lemma’’ establishing that the LISP version of the ideal Turing machine interpreter behaves just like the ideal interpreter.

The remainder of this document presents formally the ideas sketched informally above. In the following sections we define Turing machines in our logic and develop the completeness proof. We simultaneously present the input necessary to lead our mechanical theorem-prover to the proof. Indented formulas preceded by the words ‘‘Definition,’’ ‘‘Theorem,’’ or ‘‘Disable’’ are actually commands to the theorem-prover to define functions, prove theorems, or cease to use certain facts. During our discussion we so present every command used to lead the system to the completeness proof, starting from the library of lemmas and definitions produced by our proof of the unsolvability of the halting problem [Boyer & Moore 84].

5. The Formalization of Turing Machines. We now describe our formalization of Turing machines, following [Rogers 67]. A tape is a sequence of 0’s and 1’s, containing a finite number of 1’s, together with some pointer that indicates which cell of the tape the machine is ‘‘on.’’ The symbol in that cell (a 0 or a 1) is the ‘‘current symbol.’’

We represent a tape by a pair. The CAR of the pair is a list of 0’s and 1’s representing the left half of the tape, in reverse order. The CDR of the pair a list of 0’s and 1’s representing the right half of the tape. The current symbol is the first one on the right half of the tape. To represent the infinite half-tape of 0’s we use 0, since both the CAR and the CDR of 0 are 0. The function TAPE, defined below, returns T or F according to whether its argument is a Turing machine tape.

```
Definition.
(SYMBOL X) = (MEMBER X '(0 1)).

Definition.
(HALF.TAPE X)
   =
(IF (NLISTP X)
    (EQUAL X 0)
    (AND (SYMBOL (CAR X))
         (HALF.TAPE (CDR X)))).

Definition.
(TAPE X)
   =
(AND (LISTP X)
     (HALF.TAPE (CAR X))
     (HALF.TAPE (CDR X))).
```

For example, to represent the tape below (containing all 0’s to the extreme left and right) with the machine on the cell indicated by the ‘‘M’’

```
                      M
... 0 1 0 1 1 0 1 1 1 0 1 1 1 1 0 1 1 1 1 1 0 ...
```

we can use the following:

```
'((1 1 1 0 1 1 0 1 . 0) . (0 1 1 1 1 0 1 1 1 1 1 . 0)).
```

The operations on a tape are to move to the left, move to the right, or replace the current symbol with a `0` or a `1`. The reader should briefly contemplate the formal transformations induced by these operations. For example, if the current tape is given by `(CONS` *left* `(CONS` *s right*`))` then the result of moving to the right is given by `(CONS (CONS` *s left*`)` *right*`)`.

A Turing machine is a list of 4-tuples which, given an initial state and tape, uniquely determines a succession of operations and state changes. If the current state is *q* and the current symbol is *s* and the machine contains a four-tuple *(q s op q')*, then *q'* becomes the new current state after the tape is modified as described by *op*; *op* can be `'L`, which means move left, `'R`, which means move right, or a symbol to write on the tape at the current position. If there is no matching four-tuple in the machine, the machine halts, returning the tape. The function `TURING.MACHINE`, defined below, returns `T` or `F` according to whether its argument is a well-formed Turing machine:

```
Definition.
(OPERATION X) = (MEMBER X '(L R 0 1)).

Definition.
(STATE X) = (LITATOM X).

Definition.
(TURING.4TUPLE X)
   =
(AND (LISTP X)
     (STATE (CAR X))
     (SYMBOL (CADR X))
     (OPERATION (CADDR X))
     (STATE (CADDDR X))
     (EQUAL (CDDDDR X) NIL)).
```

`(CAAR` *x*`)` is an abbreviation of `(CAR (CAR` *x*`))`, `(CADR` *x*`)` of `(CAR (CDR` *x*`))`, `(CADDR` *x*`)` of `(CAR (CDR (CDR` *x*`)))`, etc.

```
Definition.
(TURING.MACHINE X)
   =
(IF (NLISTP X)
    (EQUAL X NIL)
    (AND (TURING.4TUPLE (CAR X))
         (TURING.MACHINE (CDR X)))).
```

An example Turing machine in this representation is:

```
TM:  '((q0 1 0 q1)
       (q1 0 r q2)
       (q2 1 0 q3)
       (q3 0 r q4)
       (q4 1 r q4)
       (q4 0 r q5)
       (q5 1 r q5)
       (q5 0 1 q6)
       (q6 1 r q6)
       (q6 0 1 q7)
       (q7 1 l q7)
       (q7 0 l q8)
       (q8 1 l q1)
       (q1 1 l q1)).
```

This machine is the one shown on page 14 of [Rogers 67].

Our Turing machine interpreter, TMI, is defined in terms of two other defined functions. The first, INSTR, searches up the Turing machine description for the current state and symbol. If it finds them, it returns the pair whose CAR is the operation and whose CADR is the new state. Otherwise, it returns F. The second function, NEW.TAPE, takes the operation and the current tape and returns the new tape configuration.

```
Definition.
(INSTR ST SYM TM)
   =
(IF (LISTP TM)
    (IF (EQUAL ST (CAAR TM))
        (IF (EQUAL SYM (CADAR TM))
            (CDDAR TM)
            (INSTR ST SYM (CDR TM)))
        (INSTR ST SYM (CDR TM)))
    F).

Definition.
(NEW.TAPE OP TAPE)
   =
(IF (EQUAL OP 'L)
    (CONS (CDAR TAPE)
          (CONS (CAAR TAPE) (CDR TAPE)))
    (IF (EQUAL OP 'R)
        (CONS (CONS (CADR TAPE) (CAR TAPE))
              (CDDR TAPE))
        (CONS (CAR TAPE)
              (CONS OP (CDDR TAPE))))).
```

Our Turing machine interpreter is defined as follows:

```
Definition.
(TMI ST TAPE TM N)
   =
(IF (ZEROP N)
    (BTM)
    (IF (INSTR ST (CADR TAPE) TM)
        (TMI (CADR (INSTR ST (CADR TAPE) TM))
             (NEW.TAPE (CAR (INSTR ST (CADR TAPE) TM))
                       TAPE)
             TM
             (SUB1 N))
        TAPE)).
```

We now illustrate TMI. Let TM be the example Turing machine shown above. The partial recursive function computed by the machine is that defined by f(x)=2x.

Let TAPE be the tape '(0 . (1 1 1 1 1 . 0)), in which the current cell is the leftmost 1 in a block of 4+1 consecutive 1's. Then the following is a theorem:

```
(TMI 'q0 TAPE TM N)
   =
(IF (LESSP N 78)
    (BTM)
    '((0 0 0 0 . 0) . (0 0 1 1 1 1 1 1 1 1 . 0))).
```

Observe that there are 8 1's on the final tape.

Our TMI differs from Rogers' interpreter in the following minor respects. Rogers uses B (blank) instead of our 0. In his description of the interpreter, Rogers permits tapes containing infinitely many 1's. But his characterization of partial recursive functions does not use this fact and we have limited our machines to tapes containing only a finite number of 1's. Finally, Rogers' interpreter does certain kinds of syntax checking (e.g., insuring that *op* is either L, R, 0, or 1) that ours does not, with the result that our interpreter will execute more machines on more tapes than Rogers'.

6. The Definitions of TMI.PA, TMI.X, TMI.N, and TMI.K. In this section we define the LISP analogues of INSTR, NEW.TAPE, and TMI, and we then define TMI.PA, TMI.X, TMI.N, and TMI.K.

To define a LISP program on the program alist argument of EVAL one adds to the alist a pair of the form '(*name* . (*args body*)), where *name* is the name of the program, *args* is a list of the formal parameters, and *body* is the s-expression describing the computation to be performed. We wish to define the LISP counterpart of the ideal Turing machine interpreter. We will call the program 'tmi. It is defined in terms of two other programs, 'instr and 'new.tape. The three functions defined below return as their values the (*args body*) part of the LISP definitions.

```
Definition.
(INSTR.DEFN)
   =
'((st sym tm)
  (if (listp tm)
      (if (equal st (car (car tm)))
          (if (equal sym (car (cdr (car tm))))
              (cdr (cdr (car tm)))
              (instr st sym (cdr tm)))
          (instr st sym (cdr tm)))
      f)).

Definition.
(NEW.TAPE.DEFN)
   =
'((op tape)
  (if (equal op  (quote l))
      (cons (cdr (car tape))
            (cons (car (car tape)) (cdr tape)))
      (if (equal op (quote r))
          (cons (cons (car (cdr tape)) (car tape))
                (cdr (cdr tape)))
          (cons (car tape)
                (cons op (cdr (cdr tape)))))))).

Definition.
(TMI.DEFN)
   =
'((st tape tm)
  (if (instr st (car (cdr tape)) tm)
      (tmi (car (cdr (instr st (car (cdr tape)) tm)))
           (new.tape (car (instr st (car (cdr tape))
                                  tm))
                     tape)
           tm)
      tape)).
```

Observe that the LISP program 'tmi takes only the three arguments st, tape, and tm; it does not take the number of instructions to execute. Thus, the program 'tmi is the LISP analogue of the ideal Turing machine interpreter, rather than our function TMI, and on some inputs ''runs forever.''

```
Definition.
(KWOTE X)
   =
(LIST 'quote X).
```

(KWOTE 'a) is '(quote a). In general (KWOTE X) returns a LISP s-expression that when evaluated returns X.

Here now is the function, TMI.PA, that when given a Turing machine returns the corresponding LISP program environment:

```
Definition.
(TMI.PA TM)
   =
(LIST (LIST 'tm NIL (KWOTE TM))
      (CONS 'instr (INSTR.DEFN))
      (CONS 'new.tape (NEW.TAPE.DEFN))
      (CONS 'tmi (TMI.DEFN))).
```

Note that (TMI.PA TM) is a program alist containing definitions for 'instr, 'new.tape, and 'tmi, as well as the dummy program 'tm which, when called, returns the Turing machine TM to be interpreted. Thus, the Turing machine is available during the LISP evaluation of the s-expression returned by TMI.X. But it is not available to TMI.X itself.

Here is the function, TMI.X, that, when given a Turing machine state and tape, returns the corresponding input for the LISP program 'tmi.

```
Definition.
(TMI.X ST TAPE)
   =
(LIST 'tmi
      (KWOTE ST)
      (KWOTE TAPE)
      '(tm)).
```

The form returned by TMI.X is a call of (the LISP version of) the ideal interpreter, 'tmi, on arguments that evaluate, under (TMI.PA TM), to ST, TAPE, and TM.

We now define TMI.K, which is supposed to return a number of Turing machine steps sufficient to insure that TMI terminates gracefully if EVAL terminates in stack space N.

```
Definition.
(TMI.K ST TAPE TM N)
   =
(DIFFERENCE N (ADD1 (LENGTH TM))).
```

Finally, we define TMI.N, which returns a stack depth sufficient to insure that EVAL terminates gracefully if TMI terminates in K steps.

```
Definition.
(TMI.N ST TAPE TM K)
   =
(PLUS K (ADD1 (LENGTH TM))).
```

We have now completed all the definitions necessary to formally state TURING.COMPLETENESS.OF.LISP. The reader may now wish to prove the theorem himself.

7. The Formal Proof.

7.1. *Some Preliminary Remarks about* BTM. We wish to establish a correspondence between calls of the LISP program 'tmi and the function TMI. We will do so by first noting the correspondence between primitive LISP expressions, such as '(car (cdr x)), and their logical counterparts, e.g., (CAR (CDR X)). These primitive correspondences are explicit in the definition of EVAL. We then prove correspondence lemmas for the defined LISP programs 'instr and 'new.tape, before finally turning to 'tmi.

Unfortunately, the correspondence, even at the primitive level, is not as neat as one might wish because of the role of (BTM) in the LISP computations. For example, the value of the LISP program 'cdr applied to some object is the CDR of the object *provided* the object is not (BTM). Thus, if the LISP variable symbol 'x is bound to some object, X, then the value of the LISP expression '(car (cdr x)) is (CAR (CDR X)) only if X is not (BTM) and its CDR is not (BTM). If 'x were bound to the pair (CONS 3 (BTM)), then '(car (cdr x)) would evaluate to (BTM) while (CAR (CDR X)) would be 0, since the CAR of the nonlist (BTM) is 0. LISP programs cannot construct an object like (CONS 3 (BTM)), for if a LISP program tried to 'cons something onto (BTM) the result would be (BTM). But such ‘‘uncomputable’’ objects can be constructed in the logic and might conceivably be found in the input to EVAL.

Therefore, the correspondence lemmas for our programs 'instr, 'new.tape, and 'tmi contain conditions that prohibit the occurrence of (BTM) within the objects to which the programs are applied. These conditions are stated with the function CNB (read ‘‘contains no (BTM)’s’’). To use such correspondence lemmas when we encounter calls of these programs we must be able to establish that their arguments contain no (BTM)’s. This happens to be the case because in the main theorem 'tmi is called on a STATE, TAPE, and TURING.MACHINE and all subsequent calls are on components of those objects, none of which contains a (BTM).

The correspondence lemmas for recursive programs are complicated in a much more insidious way by the possibility of nontermination or mere ‘‘stack overflow.’’ The term

```
(LIST 'instr st sym tmc),
```

describes an arbitrary call of 'instr, when the logical variables st, sym and tmc are instantiated with LISP s-expressions. Let *st*, *sym*, and *tm* be the values of st, sym, and tmc, respectively, and suppose further that those values contain no (BTM)’s. Is the value of (LIST 'instr st sym tmc) equal to (INSTR *st sym tm*)? It depends on the amount of stack depth available. If the depth exceeds the length of *tm*, |*tm*|, then the answer computed is (INSTR *st sym tm*). If the depth is less than or equal to |*tm*| is the answer computed (BTM)? Not necessarily. It depends on where (and whether) the 4-tuple for *st* and *sym* occurs in *tm*. If it occurs early enough, 'instr will compute the same answer as INSTR even with insufficient stack space to explore the entire *tm*.

In stating the correspondence lemma for 'instr we could add a hypothesis that the amount of stack depth available exceeds |*tm*|. However, not all calls of 'instr arising from the computation described in the main theorem provide adequate stack space. Part (a) of the main theorem forces us to deal explicitly with the conditions under which 'tmi computes (BTM). Thus it is necessary to know not merely when 'instr computes INSTR but also when 'instr computes (BTM).

We therefore will define a ‘‘derived function,’’ INSTRN, in the logic that is like INSTR except that it takes a stack depth argument in addition to its other arguments and returns (BTM) when it exhausts the stack, just as does the evaluation of 'instr. We will then prove a correspondence lemma between 'instr and INSTRN. We will take a similar approach to 'tmi, defining the derived function TMIN that takes a stack depth and uses INSTRN where TMI uses INSTR. After proving the correspondence lemma between 'tmi and TMIN we will have eliminated all

consideration of EVAL from the problem at hand and then focus our attention on the relation between TMIN and TMI.

Before proving any correspondence lemmas we defined ‘‘contains no (BTM)’s’’ and had the machine prove a variety of general purpose lemmas about arithmetic, EVAL, and CNB. These details are merely distracting to the reader, as they have nothing to do with Turing machines. We have therefore put those events in the appendix of this document.

7.2. *The Correspondence Lemma for* 'instr.

```
Definition.
(INSTRN ST SYM TM N)
   =
(IF (ZEROP N)
    (BTM)
    (IF (LISTP TM)
        (IF (EQUAL ST (CAAR TM))
            (IF (EQUAL SYM (CADAR TM))
                (CDDAR TM)
                (INSTRN ST SYM (CDR TM) (SUB1 N)))
            (INSTRN ST SYM (CDR TM) (SUB1 N)))
        F)).
```

The proof of the correspondence between 'instr and INSTRN requires a rather odd induction, in which certain variables are replaced by constants. We tell the theorem-prover about this induction by defining a recursive function that mirrors the induction we desire.

```
Definition.
(EVAL.INSTR.INDUCTION.SCHEME st sym tmc VA TM N)
   =
(IF (ZEROP N)
    T
    (EVAL.INSTR.INDUCTION.SCHEME
         'st
         'sym
         '(cdr tm)
         (LIST (CONS 'st
                     (EVAL st VA (TMI.PA TM) N))
               (CONS 'sym
                     (EVAL sym VA (TMI.PA TM) N))
               (CONS 'tm
                     (EVAL tmc VA (TMI.PA TM) N)))
         TM
         (SUB1 N))).
```

This definition is accepted under the principle of definition because N decreases in the recursion. We discuss below the sense in which this definition describes an induction.

Here is the correspondence lemma for 'instr. Note that in the ‘‘Hint’’ we explicitly tell the machine to induct as suggested by EVAL.INSTR.INDUCTION.SCHEME.

```
Theorem.  EVAL.INSTR (rewrite):
(IMPLIES
     (AND (CNB (EV 'AL st VA (TMI.PA TM) N))
          (CNB (EV 'AL sym VA (TMI.PA TM) N))
          (CNB (EV 'AL tmc VA (TMI.PA TM) N)))
     (EQUAL (EV 'AL (LIST 'instr st sym tmc)
                    VA
                    (TMI.PA TM)
                    N)
            (INSTRN (EV 'AL st VA (TMI.PA TM) N)
                    (EV 'AL sym VA (TMI.PA TM) N)
                    (EV 'AL tmc VA (TMI.PA TM) N)
                    N)))
Hint:  Induct as for
(EVAL.INSTR.INDUCTION.SCHEME st sym tmc VA TM N).
```

Since the correspondence lemmas are the heart of the proof, we will dwell on this one briefly.

As noted, an arbitrary call of `'instr` has the form `(LIST 'instr st sym tmc)`, where `st`, `sym` and `tmc` are arbitrary LISP *expressions*. Suppose the call is evaluated in an arbitrary variable alist, `VA`, the program alist `(TMI.PA TM)`, and with an arbitrary stack depth `N` available. Let *st*, *sym*, and *tm* be the results of evaluating the expressions `st`, `sym`, and `tmc` respectively. Then the arbitrary call of `'instr` returns `(INSTRN` *st sym tm* `N)` provided only that *st*, *sym*, and *tm* contain no `(BTM)`'s.

The reader of [Boyer & Moore 84] will recall that `(EVAL X VA PA N)` is defined as `(EV 'AL X VA PA N)`, since `EVAL` is mutually recursive with a function called `EVLIST` which evaluates a list of s-expressions. Thus, formally, both `EVAL` and `EVLIST` are defined in terms of a more primitive function `EV` which computes either, according to a flag supplied as its first argument. We state our lemmas in terms of `EV` so they will be more useful as rewrite rules. (Terms beginning with `EVAL` will be expanded to `EV` terms by the definition of `EVAL`, and then our lemmas will apply.)

The program alist for the `'instr` correspondence lemma need not be `(TMI.PA TM)` for the lemma to hold; it is sufficient if the program alist merely contains the expected definition for `'instr`. However, as `(TMI.PA TM)` is the only program alist we will need in this problem it was simplest to state the lemma this way.

The base case of the suggested induction is `(ZEROP N)`, that is, `N` is `0` or not a natural number. In the induction step, we assume `N>0`. The induction hypothesis is obtained by instantiating the conjecture with the following substitution -- derived from the recursive call in the definition of `EVAL.INSTR.INDUCTION.SCHEME`:

variable replacement term

```
st          'st
sym         'sym
tmc         '(cdr tm)
VA          (LIST (CONS 'st
                        (EVAL st VA (TMI.PA TM) N))
                  (CONS 'sym
                        (EVAL sym VA (TMI.PA TM) N))
                  (CONS 'tm
                        (EVAL tmc VA (TMI.PA TM) N)))
TM          TM
N           (SUB1 N).
```

We invite the reader to do the theorem-prover's job and construct the proof of the lemma from the foregoing lemmas, definitions and hint.

Once proved, this lemma essentially adds 'instr to the set of ''primitive'' LISP programs with logical counterparts. That is, when the theorem-prover sees:

```
(EVAL '(instr st sym tm)
      VA (TMI.PA TM) N),
```

it will replace it by:

```
(INSTRN ST SYM TM N)
```

provided 'st, 'sym, and 'tm are bound in VA to ST, SYM, and TM (respectively) and ST, SYM, and TM contain no (BTM)'s. This eliminates a call of EVAL.

7.3. *The Correspondence Lemma for* 'new.tape.

```
Theorem.  EVAL.NEW.TAPE (rewrite):
(IMPLIES
 (AND (CNB (EV 'AL op VA (TMI.PA TM) N))
      (CNB (EV 'AL tape VA (TMI.PA TM) N)))
 (EQUAL (EV 'AL (LIST 'new.tape op tape)
                VA
                (TMI.PA TM)
                N)
        (IF (ZEROP N)
            (BTM)
            (NEW.TAPE (EV 'AL op VA (TMI.PA TM) N)
                      (EV 'AL tape VA
                          (TMI.PA TM) N))))).
```

7.4. *The Correspondence Lemma for* 'tmi. We now consider the correspondence lemma for 'tmi. As we did for 'instr, consider an arbitrary call of 'tmi, (LIST 'tmi st tape tmc). Let *st*, *tape*, and *tm* be the values of the st, tape, and tmc expressions. Suppose the stack depth available is N. At first glance, the evaluation of (LIST 'tmi st tape tmc) is just (TMI *st tape tm* N), since 'tmi causes EVAL to recurse down the stack exactly once for every time TMI decrements its step count N. But again we must carefully note that if N is sufficiently small, then 'instr will return (BTM) while trying to fetch the next op and state. We therefore define the derived function TMIN, which is just like TMI except that instead of using INSTR it uses INSTRN. We then prove the correspondence lemma between 'tmi and TMIN; but first we prove two simple lemmas about INSTRN and NEW.TAPE.

```
Theorem.  CNB.INSTRN (rewrite):
(IMPLIES (AND (NOT (BTMP (INSTRN ST SYM TM N)))
              (CNB TM))
         (CNB (INSTRN ST SYM TM N))).

Theorem.  CNB.NEW.TAPE (rewrite):
(IMPLIES (AND (CNB OP) (CNB TAPE))
         (CNB (NEW.TAPE OP TAPE))).

Disable NEW.TAPE.
```

When we disable a function name we tell the theorem-prover not to consider using the definition of the disabled function in future proofs unless the function symbol is applied to explicit constants.

Here is the derived function for the program 'tmi.

```
Definition.
(TMIN ST TAPE TM N)
   =
(IF (ZEROP N)
    (BTM)
    (IF (BTMP (INSTRN ST (CADR TAPE) TM (SUB1 N)))
        (BTM)
        (IF (INSTRN ST (CADR TAPE) TM (SUB1 N))
            (TMIN (CADR (INSTRN ST (CADR TAPE)
                                   TM (SUB1 N)))
                  (NEW.TAPE
                    (CAR (INSTRN ST
                                 (CADR TAPE)
                                 TM (SUB1 N)))
                    TAPE)
                  TM
                  (SUB1 N))
            TAPE))).
```

The induction scheme we use for the `'tmi` correspondence lemma is suggested by the following definition:

```
Definition.
(EVAL.TMI.INDUCTION.SCHEME st tape tmc VA TM N)
   =
(IF (ZEROP N)
    T
    (EVAL.TMI.INDUCTION.SCHEME
     '(car (cdr (instr st (car (cdr tape)) tm)))
     '(new.tape (car (instr st (car (cdr tape)) tm))
                tape)
     'tm
     (LIST (CONS 'st
                 (EV 'AL st VA (TMI.PA TM) N))
           (CONS 'tape
                 (EV 'AL tape VA (TMI.PA TM) N))
           (CONS 'tm
                 (EV 'AL tmc VA (TMI.PA TM) N)))
     TM
     (SUB1 N))).
```

Here is the correspondence lemma for `'tmi`, together with the hint for how to prove it.

```
Theorem.  EVAL.TMI (rewrite):
(IMPLIES
      (AND (CNB (EV 'AL st VA (TMI.PA TM) N))
           (CNB (EV 'AL tape VA (TMI.PA TM) N))
           (CNB (EV 'AL tmc VA (TMI.PA TM) N)))
      (EQUAL (EV 'AL
                 (LIST 'tmi st tape tmc)
                 VA
                 (TMI.PA TM)
                 N)
             (TMIN (EV 'AL st VA (TMI.PA TM) N)
                   (EV 'AL tape VA (TMI.PA TM) N)
                   (EV 'AL tmc VA (TMI.PA TM) N)
                   N)))
Hint:  Induct as for
       (EVAL.TMI.INDUCTION.SCHEME st tape tmc VA TM N).
```

7.5. EVAL *of* `TMI.X`.

EVAL occurs twice in the completeness theorem. The two occurrences are:

```
(EVAL (TMI.X ST TAPE) NIL (TMI.PA TM) N)
```

and

```
(EVAL (TMI.X ST TAPE) NIL
      (TMI.PA TM) (TMI.N ST TAPE TM K)).
```

But since `(TMI.X ST TAPE)` is just

```
(LIST 'tmi (LIST 'quote ST) (LIST 'quote TAPE) '(tm)).
```

we can eliminate both uses of EVAL from the main theorem by appealing to the correspondence lemma for `'tmi`.

```
Theorem.  EVAL.TMI.X (rewrite):
(IMPLIES (AND (CNB ST) (CNB TAPE) (CNB TM))
         (EQUAL (EV 'AL
                    (TMI.X ST TAPE)
                    NIL
                    (TMI.PA TM)
                    N)
                (IF (ZEROP N)
                    (BTM)
                    (TMIN ST TAPE TM N)))).
```

The test on whether `N` is `0` is necessary to permit us to use the correspondence lemma for `'tmi`. If `N=0` then the evaluation of the third argument in the call of `'tmi` returns `(BTM)` and we cannot relieve the hypothesis that the value of the actual expression contains no `(BTM)`.

```
Disable TMI.X.
```

7.6. *The Relation Between* `TMI` *and* `TMIN`. By using the lemma just proved the main theorem becomes:

```
(IMPLIES
 (AND (STATE ST)
      (TAPE TAPE)
      (TURING.MACHINE TM))
 (AND
```

```
(IMPLIES (AND (NOT (ZEROP N))
              (NOT (BTMP (TMIN ST TAPE TM N))))
         (NOT (BTMP (TMI ST TAPE TM
                         (TMI.K ST TAPE TM N)))))
(IMPLIES (NOT (BTMP (TMI ST TAPE TM K)))
         (EQUAL (TMI ST TAPE TM K)
                (TMIN ST TAPE TM
                      (TMI.N ST TAPE TM K)))))).
```

That is, EVAL has been eliminated and we must prove two facts relating TMIN and TMI. We seek a general statement of the relation between TMI and TMIN. It turns out there is a very simple one. Given certain reasonable conditions on the structure of TM,

```
(TMI ST TAPE TM K) = (TMIN ST TAPE TM
                           (PLUS K
                                 (ADD1 (LENGTH TM)))).
```

Note that once this result has been proved, the main theorem follows from the definitions of TMI.K and TMI.N. But let us briefly consider why this result is valid.

Clearly, if (TMI ST TAPE TM K) terminates normally within K steps, then TMIN will also terminate with the same answer with K+|TM|+1 stack depth. The reason TMIN needs more stack depth than TMI needs steps is that on the last step, TMIN will need enough space to run INSTRN all the way down TM to confirm that the current state is a final state. Of course, we use the facts that, given enough stack space, INSTRN is just INSTR and does not return (BTM):

```
Theorem.  INSTRN.INSTR (rewrite):
(IMPLIES (LESSP (LENGTH TM) N)
         (EQUAL (INSTRN ST SYM TM N)
                (INSTR ST SYM TM))).

Theorem.  NBTMP.INSTR (rewrite):
(IMPLIES (TURING.MACHINE TM)
         (NOT (BTMP (INSTR ST SYM TM)))).
```

It is less clear that if TMI returns (BTM) after executing K steps, then the TMIN expression also returns (BTM), even though its stack depth is K+|TM|+1. What prevents TMIN, after recursing K times to take the first K steps, from using some of the extra stack space to take a few more steps, find a terminal state, and halt gracefully? In fact, TMIN may well take a few more steps (if the state transitions it runs through occur early in TM so that it does not exhaust its stack looking for them). However, to confirm that it has reached a terminal state it must make an exhaustive sweep through TM, which will of necessity use up all the stack space. Another way of putting it is this: if TMIN is to terminate gracefully, INSTRN must return F, but INSTRN will not return F unless given a stack depth greater than |TM|.

```
Theorem.  INSTRN.NON.F (rewrite):
(IMPLIES (AND (TURING.MACHINE TM)
              (LEQ N (LENGTH TM)))
         (INSTRN ST SYM TM N)).
```

We therefore conclude:

```
Theorem.  TMIN.BTM (rewrite):
(IMPLIES (AND (TURING.MACHINE TM)
              (LEQ N (LENGTH TM)))
         (EQUAL (TMIN ST TAPE TM N) (BTM))).
```

So it is easy now to prove:

```
Theorem.  TMIN.TMI (rewrite):
(IMPLIES (TURING.MACHINE TM)
         (EQUAL (TMI ST TAPE TM K)
                (TMIN ST TAPE TM
                      (PLUS K (ADD1 (LENGTH TM)))))).
```

The reader is reminded that the lemmas being displayed here are not merely read by the theorem-prover. They are proved.

7.7. *Relating* CNB *to Turing Machines*.

All that remains is to establish the obvious connections between the predicates STATE, TAPE, and TURING.MACHINE and the concept of ‘‘contains no (BTM)’s.’’

```
Theorem.  SYMBOL.CNB (rewrite):
(IMPLIES (SYMBOL SYM) (CNB SYM)).

Disable SYMBOL.

Theorem.  HALF.TAPE.CNB (rewrite):
(IMPLIES (HALF.TAPE X) (CNB X)).

Theorem.  TAPE.CNB (rewrite):
(IMPLIES (TAPE X) (CNB X)).

Disable TAPE.

Theorem.  OPERATION.CNB (rewrite):
(IMPLIES (OPERATION OP) (CNB OP)).

Disable OPERATION.

Theorem.  TURING.MACHINE.CNB (rewrite):
(IMPLIES (TURING.MACHINE TM) (CNB TM)).

Disable TURING.MACHINE.
```

7.8. *The Main Theorem*. We are now able to prove that LISP is Turing complete.

```
Theorem.  TURING.COMPLETENESS.OF.LISP:
(IMPLIES
 (AND (STATE ST)
      (TAPE TAPE)
      (TURING.MACHINE TM))
 (AND
  (IMPLIES (NOT (BTMP (EVAL (TMI.X ST TAPE)
                            NIL
                            (TMI.PA TM)
                            N)))
           (NOT (BTMP (TMI ST TAPE TM
                           (TMI.K ST TAPE TM N)))))
  (IMPLIES (NOT (BTMP (TMI ST TAPE TM K)))
           (EQUAL (TMI ST TAPE TM K)
                  (EVAL (TMI.X ST TAPE)
                        NIL
                        (TMI.PA TM)
                        (TMI.N ST TAPE TM K))))))
```

Q.E.D.

8. Conclusion.

We have described a mechanically checked proof of the Turing completeness of Pure LISP. While the result is obvious to all LISP programmers, it was not obvious, even to us, that a mechanically checked proof could be constructed so easily.

It is unusual to see a completely formal characterization of any computational paradigm. We were particularly heartened to see how simply the Turing machine paradigm could be expressed formally in our constructive logic. Because we are constructivists at heart, we found the step counter intuitively appealing. Furthermore, it did not obstruct the proofs. In any case, one's formalism must be able to express the notion that the interpreter ''runs forever.'' We invite adherents of other mechanized logical systems to construct and mechanically check the entire proof -- from the definition of EVAL and Turing machines through the completeness theorem -- for comparison with this one.

Another aspect of the proof that delighted us was the fact that our Pure LISP interpreter -- defined with its stack depth parameter to fit it into our constructive logic -- was used to simulate the ideal Turing machine interpreter, not our TMI with its step counter.

It should be noted that our use of the ''derived functions'' INSTRN and TMIN to factor EVAL out of the proof is very similar to McCarthy's notion of functional semantics. Our derived functions capture the input/output semantics of those programs in addition to such intensional properties as their termination and stack overflow properties. It would have been much harder to construct the final sequence of proofs (in which we argued about termination conditions for 'instr and 'tmi) had EVAL been involved.

Finally we would like to remind the reader of our conjecture that it may be practical, with the help of automatic theorem-provers, to do much mathematics formally.

Some readers may feel that the proof was presented at such a low level of detail that it could have been checked by a far less sophisticated theorem-prover. The reader is urged to review the 17 named theorems displayed in Section and produce a *proof* of each. We stress the word ''proof'' because we have found that many people imagine that they have a proof of a formula when in fact all they have is an intuitive grasp of what the formula says and find themselves in agreement. We would be most interested if any reader successfully checks the entire proof here with another mechanical theorem-prover or proof checker.

Other readers may feel that the proof presented here was far from formal; that is true and exactly our point. The task required of the user in this proof was to describe the proof at roughly the level we are accustomed to presenting proofs. It was the machine that constructed the formal proof.

Appendix A. Some Elementary Lemmas used in the Proof. After defining the functions TMI.PA, TMI.X, TMI.N, and TMI.K used in the statement of the completeness theorem, but before proving the first correspondence lemma, we had the theorem-prover do some preliminary work with arithmetic, EVAL, CNB, and TMI.PA. In all, 22 commands are listed in this appendix: 16 lemmas, 1 definition, and 5 disable commands. All of the results are elementary and are unconcerned with Turing machines themselves. Indeed, as the reader will see, the arithmetic, EVAL, and CNB results are of a completely general and basic nature and will be of use in other proofs about EVAL. The one lemma about TMI.PA is strictly unnecessary for the completeness proof, but makes the output less voluminous. Nevertheless, in the interests of providing a complete record of the completeness proof, we list the commands here.

A.1. *Elementary Lemmas about Arithmetic*.

```
Theorem.  LENGTH.0 (rewrite):
(EQUAL (EQUAL (LENGTH X) 0)
       (NLISTP X)).
```

```
Theorem.  PLUS.EQUAL.0 (rewrite):
(EQUAL (EQUAL (PLUS I J) 0)
       (AND (ZEROP I) (ZEROP J))).

Theorem.  PLUS.DIFFERENCE (rewrite):
(EQUAL (PLUS (DIFFERENCE I J) J)
       (IF (LEQ I J) (FIX J) I)).

Disable DIFFERENCE.
```

A.2. *Elementary Lemmas about* EVAL. These lemmas all follow directly from the definition of EV. Indeed, taken together they are virtually equivalent to the definition. However, by making them available as rewrite rules we speed up the simplification of EV expressions in which the s-expression being evaluated is a constant. For example, after proving the lemmas, the EVAL expression

```
(EVAL '(if (listp x)
           (cons (car x) (cons op (cdr (cdr x))))
           f)
      (LIST (CONS 'x X) (CONS 'op OP))
      PA
      N)
```

is immediately simplified to:

```
(IF (LISTP X)
    (CONS (CAR X) (CONS OP (CDR (CDR X))))
    F)
```

provided only that OP, X, (CAR X), (CDR X), and (CDDR X) are not (BTM). Were these lemmas not available as rewrite rules the EVAL expression would still simplify as above, but it would take longer as it would involve the heuristics controlling the opening up (many times) of the recursive function EV.

```
Theorem.  EVAL.FN.0 (rewrite):
(IMPLIES (AND (ZEROP N)
              (NOT (EQUAL fn 'QUOTE))
              (NOT (EQUAL fn 'IF))
              (NOT (SUBRP fn))
              (EQUAL VARGS
                     (EV 'LIST args VA PA N)))
         (EQUAL (EV 'AL (CONS fn args) VA PA N)
                (BTM))).
```

This lemma says that the value of an arbitrary LISP call of the program fn on a list of expressions, args, is (BTM) if the stack depth is 0 and fn is not one of the LISP primitives.

```
Theorem.  EVAL.FN.1 (rewrite):
(IMPLIES
 (AND (NOT (ZEROP N))
      (NOT (EQUAL fn 'QUOTE))
      (NOT (EQUAL fn 'IF))
      (NOT (SUBRP fn))
      (EQUAL VARGS
             (EV 'LIST args VA PA N)))
 (EQUAL
    (EV 'AL (CONS fn args) VA PA N)
    (IF (BTMP VARGS)
        (BTM)
        (IF (BTMP (GET fn PA))
            (BTM)
            (EV 'AL
                (CADR (GET fn PA))
                (PAIRLIST (CAR (GET fn PA)) VARGS)
                PA
                (SUB1 N)))))).
```

This lemma characterizes the result of an arbitrary LISP call of a nonprimitive program `fn` on `args` when the stack depth is non-`0`. If either the evaluation of `args` produces `(BTM)` or no definition for `fn` is found in the program alist, the result is `(BTM)`. Otherwise, the result is obtained by evaluating the body of `fn` in an environment in which its formal parameters are bound to the values of the actuals and the stack depth is decremented by one.

The remaining `EV` lemmas characterize the behavior of `EVAL` on the primitive programs (e.g., the program names recognized by the function `SUBRP`, which include `cons`, `car`, and `cdr`, the programs `if` and `quote`, and atomic symbols) and characterize `EVLIST` on `NIL` and the general nonempty list.

```
Theorem.  EVAL.SUBRP (rewrite):
(IMPLIES (AND (SUBRP fn)
              (EQUAL VARGS
                     (EV 'LIST args VA PA N)))
         (EQUAL (EV 'AL (CONS fn args) VA PA N)
                (IF (BTMP VARGS)
                    (BTM)
                    (APPLY.SUBR fn VARGS)))).

Theorem.  EVAL.IF (rewrite):
(IMPLIES
     (EQUAL VX1 (EV 'AL x1 VA PA N))
     (EQUAL (EV 'AL (LIST 'if x1 x2 x3) VA PA N)
            (IF (BTMP vx1)
                (BTM)
                (IF vx1
                    (EV 'AL x2 VA PA N)
                    (EV 'AL x3 VA PA N))))).

Theorem.  EVAL.QUOTE (rewrite):
(EQUAL (EV 'AL (LIST 'quote x) VA PA N)
       x).
```

```
Theorem.  EVAL.NLISTP (rewrite):
(AND (EQUAL (EV 'AL 0 VA PA N) 0)
     (EQUAL (EV 'AL (ADD1 N) VA PA N)
            (ADD1 N))
     (EQUAL (EV 'AL (PACK X) VA PA N)
            (IF (EQUAL (PACK X) 't)
                T
                (IF (EQUAL (PACK X) 'f)
                    F
                    (IF (EQUAL (PACK X) nil)
                        NIL
                        (GET (PACK X) VA)))))).

Theorem.  EVLIST.NIL (rewrite):
(EQUAL (EV 'LIST NIL VA PA N) NIL).

Theorem.  EVLIST.CONS (rewrite):
(IMPLIES
   (AND (EQUAL VX (EV 'AL X VA PA N))
        (EQUAL VL (EV 'LIST L VA PA N)))
   (EQUAL (EV 'LIST (CONS X L) VA PA N)
          (IF (BTMP VX)
              (BTM)
              (IF (BTMP VL) (BTM) (CONS VX VL))))).
```

We now disable SUBRP and EV so that their definitions are not even considered. Such expressions as (SUBRP 'cons) and (SUBRP 'tmi) will still rewrite to T and F respectively, because even disabled definitions are evaluated on explicit constants. EV expressions will henceforth be rewritten only by the lemmas above.

```
Disable SUBRP.

Disable EV.
```

A.3. CNB.

```
Definition.
(CNB X)
   =
(IF (LISTP X)
    (AND (CNB (CAR X)) (CNB (CDR X)))
    (NOT (BTMP X))).
```

In the same spirit as our EVAL lemmas, we now prove a set of lemmas that let CNB expressions rewrite efficiently.

```
Theorem.  CNB.NBTM (rewrite):
(IMPLIES (CNB X) (NOT (BTMP X))).

Theorem.  CNB.CONS (rewrite):
(AND (EQUAL (CNB (CONS A B))
            (AND (CNB A) (CNB B)))
     (IMPLIES (CNB X) (CNB (CAR X)))
     (IMPLIES (CNB X) (CNB (CDR X)))).

Theorem.  CNB.LITATOM (rewrite):
(IMPLIES (LITATOM X) (CNB X)).

Theorem.  CNB.NUMBERP (rewrite):
(IMPLIES (NUMBERP X) (CNB X)).
```

```
Disable CNB.
```

Note how these lemmas are used by the simplifier. Suppose the simplifier encounters the test `(BTMP (CDR (CDR (CAR TM))))`, as it does in the proof of the correspondence lemma for `'instr`. Suppose the hypothesis of the conjecture being proved contains `(CNB TM)`. Then the `BTMP` expression above is simplified to `F` by backwards chaining through the lemmas just proved:

```
(CNB TM)
  #→# (CNB (CAR TM))
       #→# (CNB (CDR (CAR TM)))
            #→# (CNB (CDR (CDR (CAR TM))))
                 #→# (BTMP (CDR (CDR (CAR TM)))) = F.
```

A.4. *Shutting Off Some Verbose Output.*

Consider `(TMI.PA TM)`. It is equal to a `LIST` expression containing three large list constants, namely, the constants defined to be `(INSTR.DEFN)`, `(NEW.TAPE.DEFN)`, and `(TMI.DEFN)`. Left to its own devices, the theorem-prover chooses to eliminate all mention of `TMI.PA`, `INSTR.DEFN`, `NEW.TAPE.DEFN`, and `TMI.DEFN` by replacing them by their definitions. Thus, `(TMI.PA TM)` expands into a very large expression and this expression is printed many times during the display of the machine's proofs.

However, the only use that is ever made of the expression is to use `GET` to fetch the definitions of `'tm`, `'new.tape`, `'instr`, and `'tmi`. Rather than suffer through pages of output devoted to `(TMI.PA TM)`, we here prove the four relevant properties of it, namely that `GET` returns the appropriate constant when given one of the known program names, and then disable `TMI.PA` so that it is no longer expanded into its verbose form. Again, these lemmas are not necessary to the proof, but they make the production of the proof much less taxing on the user.

```
Theorem.  GET.TMI.PA (rewrite):
(AND (EQUAL (GET 'tm (TMI.PA TM))
            (LIST NIL (KWOTE TM)))
     (EQUAL (GET 'instr (TMI.PA TM))
            (INSTR.DEFN))
     (EQUAL (GET 'new.tape (TMI.PA TM))
            (NEW.TAPE.DEFN))
     (EQUAL (GET 'tmi (TMI.PA TM))
            (TMI.DEFN))).

Disable TMI.PA.
```

BIBLIOGRAPHY

[--- 77] American National Standards Institute, Inc.
Code for Information Interchange.
Technical Report ANSI X3.4-1977, American National Standards Institute, Inc., 1430 Broadway, N.Y. 10018, 1977.

[Boyer & Moore 79]
R. S. Boyer and J S. Moore.
A Computational Logic.
Academic Press, New York, 1979.

[Boyer & Moore 81]
R. S. Boyer and J S. Moore.
Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures.
In R. S. Boyer and J S. Moore (editors), *The Correctness Problem in Computer Science*. Academic Press, London, 1981.

[Boyer & Moore 84]
R. S. Boyer and J S. Moore.
A Mechanical Proof of the Unsolvability of the Halting Problem.
JACM 31(3):441-458, 1984.

[Boyer&Moore 84]
R. S. Boyer and J S. Moore.
Proof-Checking, Theorem-Proving, and Program Verification.
In W.W. Bledsoe and D.W. Loveland (editors), *Automated Theorem Proving: After 25 Years*, pages 119-132. American Mathematical Society, Providence, R.I., 1984.

[Church 41] A. Church.
The Calculi of Lambda-Conversion.
In *Annals of Mathematical Studies*. Princeton University Press, Princeton, New Jersey, 1941.

[McCarthy 65] J. McCarthy, et al.
LISP 1.5 Programmer's Manual.
The MIT Press, Cambridge, Massachusetts, 1965.

[Rogers 67] H. Rogers, Jr.
Theory of Recursive Functions and Effective Computability.
McGraw-Hill Book Company, New York, 1967.