A User's Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover

Matt Kaufmann[1]

Technical Report #60 August 1987

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

[1]Supported by ONR Contract N00014-81-K-0634 ,Department of the Navy Contract N00039-85-K-0085, and IBM grant award "ICSCA--Research in Hardware Verification, Various Purposes"

SECTION 0: PREFACE

NOTE: Most users can get by with reading no further than Section 1 of this manual, at least until they desire to utilize the potential of this system more fully. That said .....

This manual accompanies a system for checking the provability of terms in the Boyer-Moore logic, as described in [1] and (more recently) updated in [2]. This system is loaded on top of the Boyer-Moore Theorem Prover, as explained below, and is integrated with that prover. Thus, the user can give commands at a low level (such as deleting a hypothesis) or at a high level (such as calling the Boyer-Moore Theorem Prover). As with a variety of proof-checking systems, this system is goal-directed: a proof is completed when the main goal and all subgoals have been proved. A notion of *macro commands* lets the user create compound commands, in the spirit of the *tactics* and *tacticals* of LCF [3]. Upon completion of an interactive proof, the lemma with its proof may be stored as a Boyer-Moore *event* which can be added to the user's current library of events (i.e. definitions and lemmas). An on-line help facility is provided.

We assume a little familiarity with the Boyer-Moore Theorem Prover, especially, with *definition* events **(DEFN ....)** and *lemma* events **(PROVE-LEMMA ....)**.

The manual is organized into sections as follows. Section 1 gives an introduction to the system, including a short annotated transcript of a sample session. Section 2 contains a reasonably careful explanation of the notion of a proof "state" and an introduction to the various commands and what they do. The third section is a presentation of helpful tips. The final section contains a description of the *macro command* facility together with a more detailed description of the top-level loop. This manual concludes with three appendices. The first appendix contains an annotated transcript of a sample session which is somewhat more realistic than the one presented in Section 1, so that beginning users may gain some additional feel for how the system might be used. The second appendix presents a soundness argument for this system. The final appendix lists information printed by the "short" help facility.

ACKNOWLEDGEMENTS. An early version of part of this system was written by J Moore, who I also thank for suggesting this project. I also thank David Goldschlag, Carl Pixley, and Bill Young for their helpful feedback in the development of this system. Finally, I truly appreciate the congenial and stimulating atmosphere that has been present during my year at the Institute for Computing Science at the University of Texas. It's great to love your job!

SECTION 1: INTRODUCTION TO THE SYSTEM

This section is divided into four parts. The first part contains the essentials needed to get started with the system. In the second part, a few words are said about the organization of the system into goals, states, and the state stack. The third part describes the four types of commands (change, help, meta, and macro) that the user may give. The final part (which is perhaps the most important) gives a basic introduction to the use of the system by way of an example. Although more precise details are given in later sections, the four parts of this section should provide an adequate introduction for most users.

Getting started

Let us assume that you have been given instructions on how to set up the files for the interactive proof-checker and that you have loaded them all in, after loading the Boyer-Moore system. (Presumably such instructions have been provided on a separate sheet.) You now have a proposed theorem to verify. Then submit that theorem as an argument to the Lisp function **VERIFY**; for example, to proof-check the associativity of **APPEND**, submit the form

```
(VERIFY (EQUAL (APPEND (APPEND X Y) Z)
               (APPEND X (APPEND Y Z))))
```

You will see the prompt "->: ", indicating that the system is ready to accept *proof commands*, i.e. commands which alter the *state* of the system. The idea then is to give various commands, some of which may introduce subgoals. Upon completion of the session, you may give an **EXIT** command, which may be used to cause the goal to be stored as a Boyer-Moore *event* in case the proof is complete; more on this later. However, whenever you leave the interactive proof-checker (either by an **EXIT** command or by aborting out), you may re-enter where you left off simply by submitting the form

```
(VERIFY)
```

to Lisp.

At any point during the proof, you may review the available commands by submitting **HELP** or **HELP-LONG**, which give lists of the main commands, or **(HELP <command$_1$> <command$_2$> ... <command$_n$>)**, (similarly for **HELP-LONG**), which give descriptions of the indicated commands. The principle here is that most users will be content with the help provided by **HELP** rather than **HELP-LONG**, but **HELP-LONG** may be used to get more precise specifications of the commands (which can be useful in obscure cases) and more details about unusual arguments[2] that are allowed for various commands. More precisely, the difference between **HELP** and

[2]By *arguments* to a command we mean, for example, that the numbers 3 and 4 are arguments to **DIVE** in the instruction **(DIVE 3 4)**.

HELP-LONG tends to be that **(HELP <command$_1$> <command$_2$> ... <command$_n$>)** prints out enough information for most purposes and generally gives examples, while the corresponding use of **HELP-LONG** rarely gives examples but instead gives more dry and often more complete specifications of the commands.

During the course of a session, the user will submit a number of commands which will alter the state of the system. Some of these commands will create new goals to be proved. The session is complete when all goals have been proved, in the sense that their conclusions have been reduced to **T** (*true*). (The notions of *goal* and *conclusion*, among others, are explained in the next subsection.) At that time the user may create an event by submitting an appropriate **EXIT** command. For example, the the final subsection of Section 1 (below) will display an interactive session for proving the associativity of the **APPEND** function. Upon completion of that session, the user will type an **EXIT** command in order to "create" the following Boyer-Moore event:

```
(PROVE-LEMMA ASSOCIATIVITY-OF-APPEND
             (REWRITE)
             (EQUAL (APPEND (APPEND X Y) Z)
                    (APPEND X (APPEND Y Z)))
             ((INSTRUCTIONS (INDUCT (APPEND X Y))
                            PROMOTE
                            (DIVE 1 1)
                            X UP X NX X TOP
                            (DIVE 1 2)
                            = TOP S S)))
```

The **INSTRUCTIONS** hint is a record of all the (successful) proof commands given during the session. This event may be submitted like any other Boyer-Moore event when running events in *batch mode*, i.e. when submitting events to Lisp rather than creating them through the interactive proof checker. Since the Lisp machine prints this event out in the Lisp window, the user will probably want to have output "dribbling" into a "dribble buffer", where he can then grab this event and copy it into his list of events.

Organization: goals, states, and the state stack

This section gives an informal introduction to the organization of the system. Details are postponed until Section 2.

The history of an interactive session is stored as a *state stack*. This stack consists of *proof states* (or, *"states"* for short). A *state* contains a collection of *goals*, where each *goal* has a list of *hypotheses* and a *conclusion*. Each of the goal's hypotheses can either be active or hidden; hidden hypotheses are generally ignored by proof commands unless (and until) they are made active (again). Dependencies are recorded between goals: the goals are stored in a directed acyclic graph, where an arc joins one goal to another if the former depends on the latter. At the start of an interactive session, only one state is on the state stack, namely the one corresponding to the user's input.

Let us consider an example. Suppose the user enters the system with the goal of proving the associativity of **APPEND**:

```
(verify (equal (append (append x y) z)
               (append x (append y z))))
```

Then the unique state on the state stack contains only one goal, namely the goal whose conclusion is the argument given to **verify** above and whose hypothesis list is empty. Now various commands may be given to create new states by modifying this goal, possibly introducing subgoals in the process. *The idea is that when invoking a proof command, the goal follows from its modified version together with the subgoals that are created.* Note that all goals are viewed as (implicitly) universally quantified; for example, the initial goal asserts the equality of the two **APPEND** expressions shown above for *all* values of **x**, **y**, and **z**.

Continuing with this example, notice that the initial goal follows by the Boyer-Moore induction principle from the following two subgoals, which one might call the "base step" and "induction step" respectively:

```
(IMPLIES (NOT (LISTP X))
         (EQUAL (APPEND (APPEND X Y) Z)
                (APPEND X (APPEND Y Z))))

(IMPLIES (AND (LISTP X)
              (EQUAL (APPEND (APPEND (CDR X) Y) Z)
                     (APPEND (CDR X) (APPEND Y Z))))
         (EQUAL (APPEND (APPEND X Y) Z)
                (APPEND X (APPEND Y Z))))
```

Now this induction is "dual to" the recursion in the definition of the function **APPEND**. Hence the following command may be invoked to generate these subgoals. (This command is in complete analogy to the giving of this as a hint to the Boyer-Moore **PROVE-LEMMA** command.)

```
(induct (append x y))
```

At any rate, the original goal follows from the two subgoals above (when all goals are viewed as universally quantified). So, when the above command is executed, a new state is pushed onto the state stack, where the new state contains two new goals (one for each of the subgoals displayed above). Moreover, since the original goal follows from these two goals, its conclusion may be replaced by **T** (*true*) in the new state. That is, the property mentioned earlier does indeed hold (and we restate it now for emphasis):

(*) *When invoking a proof command, the goal follows from its modified version together with the subgoals that are created.*

Now in fact, a *state* also has a *current goal* as well as a pointer to the *current subterm* of the current goal's conclusion. Here is another example of creating a new state to be pushed onto the state stack. Suppose that the current goal has hypotheses as follows:

```
H1.  (LISTP X)
H2.  (EQUAL (APPEND (APPEND (CDR X) Y) Z)
            (APPEND (CDR X) (APPEND Y Z)))
```

Also suppose that the conclusion of the current goal is as follows, where the current subterm (cf. first paragraph of this subsection) has been "highlighted" with asterisks:

```
(EQUAL (CONS (CAR X)
             (*** (APPEND (APPEND (CDR X) Y) Z)
                  ***))
       (CONS (CAR X)
             (APPEND (CDR X) (APPEND Y Z))))
```

Now the second hypothesis equates the current subterm with the term **(APPEND (CDR X) (APPEND Y Z))**, and we might wish to make a substitution of this new term for the current subterm. It turns out that the command = does just that. But more precisely, this = command pushes a new state on top of the state stack, where the new state is obtained from the old state by making the substitution indicated by the second hypothesis (for the current subterm in the conclusion of the current goal). Unlike the previous example, no new subgoals are generated, and the modified version of the current goal has a conclusion that is not yet **T** (*true*). But again, the key property (*) above is maintained, since the current version of the goal follows from the modified version (in fact it follows by making the reverse of the equality substitution that was used).

Suppose instead that the second hypothesis had not been present in the example above. Then a different version of the = command would be appropriate here, namely **(= * (APPEND (CDR X) (APPEND Y Z)) 0)**. (This and other commands are introduced in the example here and in the first appendix, and are explained by the help facility.) In this case, the substitution would still have been made but also a new subgoal would have been generated. This new subgoal would have had the same hypotheses as the current goal, but its conclusion would have been the equality of the current subterm with the term to be substituted for it. Again the key property (*) would be maintained: the current goal follows from the new version (obtained by substitution) together with the goal stating the equality of the relevant terms.

The four types of commands: change, help, meta, and macro

Recall that the purpose of an interactive proof session is ultimately to create a state in which every goal has conclusion equal to **T** (*true*). The commands that push new states on top of the state stack, such as the **INDUCT** and = commands described in the subsection above, are called *change* commands. These are the only "official" commands, in the sense that each **PROVE-LEMMA** event created upon **EXIT** from the system will store these commands in the **INSTRUCTIONS** hint, as shown in the first subsection above.

However, there is a vast difference between the result of an interactive session and the session itself. That is, even though one's aim is to arrive at a sequence of change commands which result in all goals having conclusions of **T**, one would certainly like helpful support toward achieving that end. The other three types of commands provide such support.

The *help* commands display useful information while making no change whatsoever in the state of the system. Probably the most commonly used help command is **P**: print the current subterm. Some other commonly used help commands include **PP** (like **P**, but with a more raw syntax), **HYPS** (print the current goal's hypotheses), **SHOW-REWRITES** (show the rewrite rules which apply to the current subterm), and **HELP** and **HELP-LONG** (print information about the commands); a complete list is given in the final appendix.

The *meta* commands allow one to manipulate the entire state stack to (potentially) create a new state stack. However, the new state stack will still correspond to a sequence of change commands.[3] Probably the most commonly used meta command is **UNDO**, which may be used to pop the state stack (i.e. revert to a previous state). In addition, the previous state stack is stored to that the meta command **RESTORE** may be used to undo the effect of an **UNDO** command. Most of the other meta commands are used to create *macro* commands, which are the remaining type of command.

The idea behind *macro* commands are that they enable the user to extend the system. For example, suppose that one wants a command which prints the value of an arbitrary Lisp form **<exp>**. Now the meta command **LISP** evaluates an arbitrary Lisp form, so one could simply submit the command **(LISP (PRINT <exp>))**. But perhaps the user requires this sort of thing frequently and is tired of typing **(LISP (PRINT ...))**. Then he can define a macro command which does just that. In fact, some macro commands are provided in the system that is initially loaded, including a macro command **PRINT**. Macro commands are discussed at length in Section 4, including a detailed description of the top-level evaluation mechanism and how to define macro commands. So we'll keep the discussion here short. Even at this stage though it is worth pointing out that the help facility does print information about the predefined macro commands, so the user can begin using them right away. It is also worth pointing out that, as the terminology implies, a macro command is actually expanded (textually) into its body, and then the resulting command is resubmitted. (The resulting command may however also be a macro command, which is in

[3]In fact, the command creating a state is one of the fields of the *state* record, so the commands actually exist in the state stack. Actually, a malicious user could create "invalid" state stacks using e.g. the **LISP** meta command; however, we claim that the non-malicious user would not get into this trouble. Moreover, even the malicious user cannot violate soundness, in the sense that we do not consider events to have been completely checked until they have been run back through the system. Since only *change* commands may be given as **INSTRUCTION** hints to the **PROVE-LEMMA** events, malicious **LISP** commands will be ignored when these events are run back through.

turn resubmitted, and so on.) So for example, in the example described above where we defined the macro command **(PRINT X)** to expand to **(LISP (PRINT X))**, if the user submits the macro command **(PRINT <exp>)** then the top-level interactive loop expands this command into the command **(LISP (PRINT X))**, which (being a meta command rather than a macro command) is then executed.

The full story of macro commands is presented in Section 4. That section is however *not* necessary reading for the user to be able to use macro commands.

An introduction by way of example

Here is a annotated display of a short interactive session corresponding to the **PROVE-LEMMA** event shown above. Comments will be enclosed in curly braces {} and italicized. Other than the initial **verify** command, user input is preceded on each line by the prompt "->: "; the rest is printed by the system. Some extra blank lines have been added for readability.

A slightly more realistic example appears in Appendix 1.

```
(verify (equal (append (append x y) z)
               (append x (append y z))))

->: p  {Print the current subterm}

(EQUAL (APPEND (APPEND X Y) Z)
       (APPEND X (APPEND Y Z)))

->: (induct (append x y))  {Similar to the Boyer-Moore INDUCT hint}

Creating 2 new subgoals, (MAIN . 1) and (MAIN . 2).

The proof of the current goal, MAIN, has been completed.  However, the
following subgoals of MAIN remain to be proved: (MAIN . 1) and (MAIN . 2).
Now proving (MAIN . 1).
{Note:  The proof of this goal is "completed" because there's
nothing left to do except to prove its subgoals.}

->: p  {Print the current subterm}

(IMPLIES (AND (LISTP X)
              (EQUAL (APPEND (APPEND (CDR X) Y) Z)
                     (APPEND (CDR X) (APPEND Y Z))))
         (EQUAL (APPEND (APPEND X Y) Z)
                (APPEND X (APPEND Y Z))))
```

```
->: hyps  {Print the current hypotheses}

*** Active top-level hypotheses:
There are no top-level hypotheses to display.

*** Active governors:  {Governors are discussed later}
There are no governors to display.

->: promote  {Turn the left side of the implication into top-level hypotheses}

->: hyps  {Print the current hypotheses}

*** Active top-level hypotheses:
H1.  (LISTP X)
H2.  (EQUAL (APPEND (APPEND (CDR X) Y) Z)
            (APPEND (CDR X) (APPEND Y Z)))

*** Active governors:
There are no governors to display.

->: p  {Print the current subterm}

(EQUAL (APPEND (APPEND X Y) Z)
       (APPEND X (APPEND Y Z)))

->: (dive 1 1)  {Point to the first argument of the current subterm and
        then to that subterm's first argument}

->: p  {Print the current subterm}

(APPEND X Y)


->: pp-top  {Print the entire conclusion, highlighting the current subterm}

(EQUAL (APPEND (*** (APPEND X Y) ***) Z)
       (APPEND X (APPEND Y Z)))

->: x  {Expand the function call in the current subterm, namely in
       (APPEND X Y), and simplify the result.}

->: p  {Print the current subterm -- Notice that the expansion using the X command
      simplified away the IF test in the body of the definition of APPEND.}

(CONS (CAR X) (APPEND (CDR X) Y))

->: pp-top  {Print the entire conclusion, highlighting the current subterm}

(EQUAL (APPEND (*** (CONS (CAR X) (APPEND (CDR X) Y))
                    ***)
               Z)
       (APPEND X (APPEND Y Z)))

->: up  {Move up to the enclosing term}
```

```
->: pp-top {Print the entire conclusion, highlighting the current subterm}

(EQUAL (*** (APPEND (CONS (CAR X) (APPEND (CDR X) Y))
                    Z)
            ***)
       (APPEND X (APPEND Y Z)))

->: x {Expand the function call in the current subterm and simplify}

->: pp-top {Print the entire conclusion, highlighting the current subterm}

(EQUAL (*** (CONS (CAR X)
                  (APPEND (APPEND (CDR X) Y) Z))
            ***)
       (APPEND X (APPEND Y Z)))

->: nx {Move to the next argument in the current subterm}

->: pp-top {Print the entire conclusion, highlighting the current subterm}

(EQUAL (CONS (CAR X)
             (APPEND (APPEND (CDR X) Y) Z))
       (*** (APPEND X (APPEND Y Z)) ***))

->: x {Expand the function call in the current subterm and simplify}

->: pp-top {Print the entire conclusion, highlighting the current subterm}

(EQUAL (CONS (CAR X)
             (APPEND (APPEND (CDR X) Y) Z))
       (*** (CONS (CAR X)
                  (APPEND (CDR X) (APPEND Y Z)))
            ***))

->: top {Move to the top of the goal's conclusion}

->: hyps {Print the current hypotheses}

*** Active top-level hypotheses:
H1.  (LISTP X)
H2.  (EQUAL (APPEND (APPEND (CDR X) Y) Z)
            (APPEND (CDR X) (APPEND Y Z)))

*** Active governors:
There are no governors to display.

->: p {Print the current subterm}

(EQUAL (CONS (CAR X)
             (APPEND (APPEND (CDR X) Y) Z))
       (CONS (CAR X)
             (APPEND (CDR X) (APPEND Y Z))))

->: (dive 1 2) {Point to the first argument of the current subterm and
       then to that subterm's second argument}
```

```
->: pp-top {Print the entire conclusion, highlighting the current subterm}

(EQUAL (CONS (CAR X)
             (*** (APPEND (APPEND (CDR X) Y) Z)
                  ***))
       (CONS (CAR X)
             (APPEND (CDR X) (APPEND Y Z))))

->: hyps

*** Active top-level hypotheses:
H1.  (LISTP X)
H2.  (EQUAL (APPEND (APPEND (CDR X) Y) Z)
            (APPEND (CDR X) (APPEND Y Z)))

*** Active governors:
There are no governors to display.

->: = {Make a substitution for the current subterm, using an equality
    among the current hypotheses and governors.}

->: p {Print the current subterm}

(APPEND (CDR X) (APPEND Y Z))

->: top

->: p {Print the current subterm}

(EQUAL (CONS (CAR X)
             (APPEND (CDR X) (APPEND Y Z)))
       (CONS (CAR X)
             (APPEND (CDR X) (APPEND Y Z))))

->: s {Simplify}

The current goal, (MAIN . 1), has been proved, and has no dependents.
Now proving (MAIN . 2).

->: p {Print the current subterm}

(IMPLIES (NOT (LISTP X))
         (EQUAL (APPEND (APPEND X Y) Z)
                (APPEND X (APPEND Y Z))))

->: s {Simplify}

The current goal, (MAIN . 2), has been proved, and has no dependents.

*!*!*!*!*!*!*  All other goals have also been proved!  *!*!*!*!*!*!*
You may wish to EXIT -- type (HELP EXIT) for details.
```

```
->: (exit associativity-of-append (rewrite))
```
{As described previously}

```
The indicated goal has been proved.  Here is the desired event:

(PROVE-LEMMA ASSOCIATIVITY-OF-APPEND
             (REWRITE)
             (EQUAL (APPEND (APPEND X Y) Z)
                    (APPEND X (APPEND Y Z)))
             ((INSTRUCTIONS (INDUCT (APPEND X Y))
                            PROMOTE
                            (DIVE 1 1)
                            X UP X NX X TOP
                            (DIVE 1 2)
                            = TOP S S)))
Do you want to submit this event?
Y (Yes), R (Yes and replay commands), or N (No) ? Yes
```
{User response}
```
[ 0.2 0.0 0.0 ]
ASSOCIATIVITY-OF-APPEND
```
{The event has been stored in the Boyer-Moore database of events, i.e. it now shows up in the Lisp variable **CHRONOLOGY***.}*

Remark. The triple of numbers above the event name "ASSOCIATIVITY-OF-APPEND" above is meaningless when printed as a response to the prompt at the end of an interactive session. However, when the event is submitted to Lisp (in what we refer to as *batch mode*), the numbers have the following meaning (which agrees with the usual meaning). The sum of the second and third numbers is the amount of time spent inside the Theorem Prover, and the third number alone is that portion of the sum which is spent inside the I/O routines. The first number is what is left of the total, i.e. the sum of the three numbers is the total time spent on the event. To summarize, we have three times:

[Miscellaneous_time Proof_time I/O_time(inside Prover)]

SECTION 2: A MORE DETAILED DESCRIPTION OF THE SYSTEM

In this section we present detailed accountings of the notions of *goal*, *state*, and *state stack* that were introduced in Section 1, together with some related notions. At the same time we also give an overview of the *commands* that the user may give; more complete details of the commands may be found by using the help facility, i.e. by typing **(HELP <command$_1$> <command$_2$> ... <command$_n$>)**, or similarly with **HELP-LONG**. (Information is also included in the final appendix.) Finally, top-level matters such as creation of **PROVE-LEMMA** events and aborting and re-entering an interactive session are dealt with in detail in the final subsection.

Goals

A *goal* is a record with the fields discussed below. It is not important for the user to know the names of these

fields (or of the fields for the *state* record to be presented next), but the concepts underlying them are important for using the system. It is also important to keep in mind that a goal is viewed as the universal closure[4] of the implication whose antecedent is the term formed by conjoining the goal's hypotheses and whose consequent is the goal's conclusion (or, of just the conclusion in case there are no hypotheses). Recall though the convention from [1] that a term **<exp>** may be used as a formula by identifying it with the negation of **[EXP = F]**.

A *goal* **consists of:**
CONC, HYPS, DEPENDS-ON, GOAL-NAME, and ORIG-CONC-AND-HYPS

CONC is called the *conclusion* of the goal, and is a Boyer-Moore term.

HYPS is called the *hypotheses* of the goal, or sometimes the *top-level hypotheses*, and is a list of Boyer-Moore terms. The hypotheses of the current goal can be found by employing the help command **HYPS**. But actually --

HYPS is *really* a list of *pairs*, where the first element of the pair is a hypothesis and the second element is either the atom **A** or the atom **H**, indicating that the hypothesis is *active* or *hidden* (respectively). All of the *change* commands are set up so that they "ignore" hidden hypotheses. For example, the **S** (simplify) command uses only the active hypotheses in simplifying the current subterm. The change commands **HIDE-HYPS** and **SHOW-HYPS** are used to hide and activate hypotheses, so that one can hide hypotheses temporarily and then bring them back. There is also a **DROP** change command which eliminates hypotheses.[5]

There are several other change commands which modify the **HYPS**. Here are brief, simplified descriptions; use the help facility or see the final appendix for more details. The **CLAIM** command allows one to add additional hypotheses that follow from the existing (active) hypotheses. The command **PROMOTE** modifies a current goal with conclusion **(IMPLIES $TERM_1$ $TERM_2$)** by replacing its conclusion with **$TERM_2$** while adding **$TERM_1$** to its hypotheses. (More accurately, as with all change commands it pushes a new state on the state stack which agrees with the old state except that the current goal has been modified approximately as indicated.) On the other hand, the command **DEMOTE** is (roughly) an inverse to **PROMOTE**. The **CONTRADICT** command exchanges a hypothesis and the conclusion while negating each of them. Finally, **USE-GOAL** and **USE-LEMMA** allow one to add hypotheses that are instances of another goal or of a lemma from the chronology, respectively. In the former case, it is the *original* form of the used goal that is actually used; see the description of **ORIG-CONC-AND-HYPS** below.

---

[4]The *universal closure* of a formula in first-order logic is obtained by prefixing it with a sequence of all quantifiers of the form **(FORALL X)** as **X** ranges over the free variables of the formula.

[5]Except one can always return to a previous state with the **UNDO** command; more on **UNDO** later.

DEPENDS-ON is a list of Lisp objects (S-expressions): the names of the goals that the given goal depends on. (Intuitively, goal X depends on goal Y if Y is related to X by the transitive closure of the subgoal relation, i.e. Y is a subgoal of X or a subgoal of a subgoal of X or ....) The dependents of all the goals are shown with the help command **GOAL-NAMES** (or macro command **GOALS**). Several commands modify the **DEPENDS-ON** field. For example, the command **USE-GOAL** mentioned in the paragraph above will add the used goal's name to the **DEPENDS-ON** field of the current goal. In fact, this command will fail if the current goal's name is in the **DEPENDS-ON** field of the goal which is to be used; we maintain the invariant that the dependency graph has no cycles, i.e. we avoid circular reasoning. A very useful command that creates a dependency is **PUSH**, which replaces the current subterm **<exp>** by **T** (*true*).[6] In this case, the current goal is further modified by adding the name of a new goal to its **DEPENDS-ON** field, where the new goal has the current goal's active hypotheses and governors as its (top-level) hypotheses and has **<exp>** as its conclusion. The **INDUCT** command creates subgoals as discussed in the demo in Section 1. Other change commands that create dependents are **GENERALIZE**, **SPLIT**, and sometimes **CLAIM**, **REWRITE**, =, and **USE-GOAL**.

GOAL-NAME is a Lisp object that we call the *name* of the goal. When the system generates subgoals of a given goal named **<name>** it does so by creating new names of the form **(<name> . N)**, where **N** is a positive integer. (Exceptions: the commands **PUSH** and **GENERALIZE** allow one to specify the new subgoal.)

Finally, **ORIG-CONC-AND-HYPS** is a list whose first element is the conclusion of the goal at the state where it first existed and whose other elements are its original hypotheses. By the way, the hypotheses in **ORIG-CONC-AND-HYPS** are indeed terms rather than the term-atom pairs found in **HYPS**.

States

Next, we consider the *state* record type.

```
A state consists of:
INSTRUCTION, CURRENT-TERM, GOVERNORS, CURRENT-ADDR-R, GOAL,
   OTHER-GOALS, CUMULATIVE-LEMMAS-USED, FREWRITE-DISABLED-RULES,
   and ABBREVIATIONS
```

INSTRUCTION is the change command that created the given state from the previous state. (However, **INSTRUCTION** is **START** for the initial state, i.e. the state created by the call of **VERIFY** on the term to be proved.)

---

[6] Exception: if **<exp>** is not known either to be boolean or to be in a position where only propositional equivalence needs to be maintained, then it is replaced by **(IF <exp> <exp> T)**.

The next three fields -- **CURRENT-TERM**, **GOVERNORS**, and **CURRENT-ADDR-R** -- are all based on the notion of a pointer to a subterm of the current goal's conclusion. This pointer can be thought of as a list of positive integers which gives directions for diving in to the conclusion. For example, the *address* would be **(2 3 1)** for the subterm **(PLUS X Y)** of:

```
(TIMES (ADD1 X)
       (IF (EQUAL X Y)
           Y
           (SUB1 (PLUS X Y))))
```

since: we are diving to the second argument of the **TIMES** term, then the third argument of that **IF** term, and then finally the first argument of the **SUB1** term. The **CURRENT-TERM**, usually called the *current subterm*, is in this case **(PLUS X Y)**, while the **CURRENT-ADDR-R**, i.e. current-address-reversed, is **(1 3 2)**. The **GOVERNORS** are, roughly speaking, the IF-tests accumulated on the way to diving down to the current subterm. More precisely, the **GOVERNORS** is a list of all terms which *govern* the current subterm, in the sense of the definition of *governs* on the top of page 45 of [1]. For example, the **GOVERNORS** of **(PLUS X Y)** in the term displayed above is the one-element list **((NOT (EQUAL X Y)))**. NOTE: the **GOVERNORS** are defined only with respect to the **IF**-structure of the term. So for example, there are no governors of **X** in the term **(IMPLIES Y X)**. (There are of course other ways of using the hypothesis, as discussed in the "Proving Implications" discussion in Section 3.)

Let us discuss the commands which are particularly related to these notions of **CURRENT-TERM**, **GOVERNORS**, and **CURRENT-ADDR-R**. The commands **P** and **PP** both print the current subterm, the difference being that **P** introduces some notational conventions for terms with top function symbols among **CAR**, **CDR**, **CONS**, **AND**, **OR**, **PLUS**, **TIMES**. So for example, the command **P** prints the term **(PLUS X (PLUS Y Z))** as **(PLUS X Y Z)**, while **PP** prints it as is. In other words, the command **P** prints the current subterm just as the Boyer-Moore Theorem Prover would print it, while **PP** simply prints the term according to its actual structure.[7] Why ever use **PP** rather than the (prettier) **P**? Because the command **DIVE** may be used to move to a subterm of the current subterm according to a specified list of addresses (as explained in the example in Section 1), and on a few occasions it thus helps to see the term displayed in the more "raw" form given by the **PP** command. Other change commands besides **DIVE** which change the current subterm are **UP**, **TOP**, **NX**, and **BK**, all of which are of course explained by the help facility (and in the final appendix).

We continue with our description of the fields of a **STATE**. **GOAL** is the current goal, in the sense of "goal"

[7] An exception is that both commands print explicit value terms using the quote notation, which is explained in detail in [4]. So for example, **(LIST 'A 'B)** is printed as **'(A B)** with both commands.

described in the previous subsections (i.e. a record consisting of a **CONC**, **HYPS**, **DEPENDS-ON**, **GOAL-NAME**, and **ORIG-CONC-AND-HYPS**). The user may change the current goal by using the change command **CHANGE-GOAL**. Also, when a goal has been completed, i.e. its conclusion is **T** (*true*), the system automatically chooses an uncompleted goal as the current goal (unless of course there are no further goals to prove, in which case it informs the user of that situation).

OTHER-GOALS is a list of all the goals in the **STATE** other than the current **GOAL** together with information about the current subterm of each goal.[8]

CUMULATIVE-LEMMAS-USED is a list of atoms to be used when forming the dependencies for a **PROVE-LEMMA** event which results from an interactive session. So for example, any function symbol which has a call expanded by the **S** (simplify) command will be added to this field in the new state, as will any function symbol or name of lemma used in a call to the Theorem Prover by the **PROVE** command (or any of several other commands).

FREWRITE-DISABLED-RULES is a list of atoms which are names of function symbols or rewrite rules which are to be ignored by the so-called "fast rewriter", which is called by the **S** (simplify) and **X** (expand) commands. This field is updated when the **ENABLE** and **DISABLE** commands are executed.

Finally, **ABBREVIATIONS** is a list of abbreviations in the following sense. An *abbreviation* is a pair consisting of an atom and a term, where the atom begins with the ampersand '@' character. The user interface is set up so that both printing of terms and (generally) reading of terms in commands is done with respect to this list of abbreviations. Unabbreviating takes place from the outside in. So, for example, if the abbreviations consist of the pairs

```
((@W . (ADD1 X))
 (@V . (PLUS (ADD1 X) Y)))
```

then the term **(TIMES Z (PLUS (ADD1 X) Y))** would be printed as **(TIMES Z @V)** rather than as **(TIMES Z (PLUS @W Y))**. Notice that this is true whether that term is printed as part of the current term or as one of the hypotheses (*via* the **HYPS** command). The **ADD-ABBREVIATION** and **REMOVE-ABBREVIATIONS**

[8]More precisely, each member of the list **OTHER-GOALS** is a list of the form **(goal' current-term' governors' current-addr-r')**, where each member of this list is of the type that one would expect from its name. The idea here is that when **goal'** becomes the current goal, then the other three members of the list will become the current subterm, governors, and current-address-reversed (respectively).

commands modify the **ABBREVIATIONS** field of the state, or more precisely, they push a new state onto the state stack which obtained from the previous top state by changing the **ABBREVIATIONS** field appropriately. The current abbreviations can be viewed using the help command **SHOW-ABBREVIATIONS**.

State stacks and other related matters

The next topic for this section is that of the *state stack*. As we already discussed in Section 1, a stack of states is maintained -- in fact it is stored in the global Lisp variable **STATE-STACK** -- such that execution of a change command pushes a new state on top of this stack. More precisely, when the user submits a change command then one of two things can happen. If the command is not "allowed", for example if the **S** (simplify) command fails to make any changes in the current subterm, then **STATE-STACK** is unchanged. However, if the command "succeeds", then the appropriate new state is pushed on top of the state stack.[9]

UNDO is a meta command which can be used to pop states off the state stack. In this way, abortive "branches" in an interactive proof effort can be undone. The **UNDO** and **BOOKMARK** (see below) commands interact in that the argument to **UNDO** can be a bookmark. (As usual, we defer the details to the final appendix or to the user's inquiry of the help facility.) Generally, though, the user will give **UNDO** a numeric argument which indicates the number of states to be popped. (The default of **1** is used when no arguments to **UNDO** are given.) The help command **COMMANDS** may be used to list the **INSTRUCTION** fields of the states in **STATE-STACK**. They are listed in reverse order in order to aid use of the **UNDO** command.

On occasion, a user may wish to undo an **UNDO** command. The meta command **RESTORE** has been provided for this purpose. What **RESTORE** *really* does is swap the value of **STATE-STACK** with the value of another global Lisp variable, **OLD-SS**. Each time an **UNDO** command is executed, the variable **OLD-SS** is set to the existing value of **STATE-STACK** while **STATE-STACK** is in turn reset to be the popped state stack; and that is why **RESTORE** will undo an **UNDO**.

Two change commands push new states on the state stack in particularly trivial ways. The command **COMMENT** simply inserts comments: it creates a new state from the previous top state by simply inserting an appropriate comment into the **INSTRUCTION** field. The command **BOOKMARK** is similar in that it simply creates an instruction of the form **(BOOKMARK x)** which is understood by the meta command **UNDO**.

---

[9]Actually, there is a third possibility in the current system, namely that an error is caused. Most or all errors are however caused "on purpose". For example, if the second argument to **ADD-ABBREVIATION** contains a function symbol that is unknown in the current Boyer-Moore database, then an error is caused by the Boyer-Moore Lisp function **TRANSLATE**. In this case the user can return to the top-level interactive loop simply by submitting the Lisp form **(VERIFY)**.

We omit mention of the other commands here, referring the reader once again to the help facility or the final appendix.

Top-level matters

In this final subsection of Section 2, we discuss carefully the relation between this system and the Boyer-Moore system. Let us begin by recalling some such matters which have been explained above. To enter the system, one submits

```
(verify <term>)
```

to Lisp. This puts the user into a read-eval-print loop signified by the prompt "->: ", where one can submit the various proof-checker commands. However, the user may for some reason wish to leave this loop and return to the top level of Lisp, either by aborting out or by giving the command **EXIT**. By submitting the form

```
(verify)
```

to Lisp, the user will return to the system's read-eval-print loop (and receive the prompt "->: "), where the global state (i.e. **STATE-STACK** and **OLD-SS**) is exactly as it was when the system was exited[10]. **(VERIFY)** also initializes certain parameters to match the current state of the Theorem Prover's database. The **(VERIFY)** feature is quite useful, for example, in conjunction with calling the Theorem Prover (using the command **PROVE**, for example). For, if the Prover goes down a "bad path" one may wish to abort the proof attempt, which will throw the user back to the top level of Lisp; but using **(VERIFY)** the user can return to where he was just before submitting the **PROVE** command. More generally, aborting during the execution of any *change command* should result in no change to the global state (i.e. the state stack), in which case the execution of **(VERIFY)** will return the user to that global state. The user can ascertain whether the command has completed, by submitting the help command **COMMANDS**; we believe that it is not possible for a change command to complete only partially.

We have already mentioned that a command of the form

```
(exit <event-name> <lemma-types>)
```

will let the user create a Boyer-Moore event by simply answering "Y" to the prompt (as illustrated by the example in the subsection above). Generally this form of the **EXIT** command will only be used once all the goals have been proved (i.e. their conclusions have all been reduced to **T** *(true)*). However, it is possible to create an event which records progress made during an interactive session. If there are remaining goals to prove, then upon execution of the **EXIT** command displayed above the system will first print a warning and then will print an appropriate event.

[10] unless, of course, changes were made using **SETQ** or **RPLACA** or such evilness in the top level of Lisp!

This event will have a conclusion of **T** (*true*), and thus the event has no logical content. It does have operational content, however: the Lisp function **RE-ENTER** may be used to get back in to the environment recorded by the exit process above.

For example, suppose that one is proving commutativity of times and has given commands which result in a state where a goal still remains to be proved. Here is the information which might be printed in such a situation, upon invocation of the **VIEW** macro command:

```
*** Active top-level hypotheses:
H1.  (NOT (ZEROP X))

*** Active governors:
There are no governors to display.

The current subterm is:
(EQUAL (PLUS Y (TIMES Y (SUB1 X)))
       (TIMES Y X))
```

At this point, one might choose to exit the interactive loop:

```
->: (exit times-comm-progress nil)

WARNING:  The appropriate goal has not been proved.
The following event notes progress made during this (fresh) session.

(PROVE-LEMMA TIMES-COMM-PROGRESS NIL T
             ((START-GOAL (EQUAL (TIMES X Y) (TIMES Y X)))
              (INSTRUCTIONS (INDUCT 1)
                            PROVE PROMOTE
                            (DIVE 1)
                            X
                            (DIVE 2)
                            =
                            (HIDE-HYPS 2)
                            TOP)))
Do you want to submit this event?
Y (Yes), R (Yes and replay commands), or N (No) ? Yes
[ 0.0 0.0 0.0 ]
TIMES-COMM-PROGRESS
```

Now one might prove a lemma which can be used to finish the proof of the goal displayed above:

```
(prove-lemma times-add1 (rewrite)
  (equal (times x (add1 y))
         (plus x (times x y))))
```

Finally, one can re-enter the previously recorded interactive environment. This is done by supplying **RE-ENTER** with the name of the interactive event displayed above:

```
(re-enter times-comm-progress)
```

At this point the user will see the prompt "->: " and can proceed with the interactive proof. It turns out that the single command **PROVE** finishes the proof. After execution of this **PROVE** command the user can exit:

```
->: (exit times-comm (rewrite))

The indicated goal has been proved.  Here is the desired event:

(PROVE-LEMMA TIMES-COMM
             (REWRITE)
             (EQUAL (TIMES X Y) (TIMES Y X))
             ((PREVIOUS-EVENT TIMES-COMM-PROGRESS)
              (INSTRUCTIONS PROVE)))
Do you want to submit this event?
Y (Yes), R (Yes and replay commands), or N (No) ? Yes
[ 0.1 0.0 0.0 ]
TIMES-COMM
```

The user doesn't have to understand the meaning of the hints above, since the system prints this event and the user need only put it in his event file (or retrieve it using the Theorem Prover's Lisp function **PPE**, e.g. **(PPE 'TIMES-COMM)**). Actually, though, the hints mean what they say: the **PREVIOUS EVENT** from which the proof was continued is called **TIMES-COMM-PROGRESS**, and the only instruction needed to complete the proof was the single instruction **PROVE**.

Now in fact, the user had at least one other option in the way the proof was managed above. Instead of leaving the system the first time with the command **(EXIT TIMES-COMM-PROGRESS NIL)**, the user could have simply typed **EXIT** (or aborted). Then no event would be stored in the Theorem Prover's database. The user could then submit the lemma **TIMES-ADD1** as shown above. Finally, the re-entry mechanism could have been used that was shown earlier: one simply executes **(VERIFY)**. Then one would re-enter the interactive system at the same state in which it was left, and could give the final **PROVE** command and then submit **(EXIT TIMES-COMM (REWRITE))** to conclude the proof. There is a danger with this approach. Imagine re-playing the events created in this manner. Since the lemma **TIMES-ADD1** would precede **TIMES-COMM**, the **TIMES-ADD1** rewrite rule would be available for calls to the Theorem Prover that take place during the replay of the proof of **TIMES-COMM**. This approach could (in rare cases) cause the replay to fail, since calls to the prover which precede the place where re-entry took place were not originally made with **TIMES-ADD1** present. Probably such failures are rare, and the user can of course check the replayability by answering "R" to the exit prompt instead of "Y". Thus, it should not often be necessary to use the premature exit feature described in the previous paragraph.

There is also a mechanism for storing a goal other than the main (i.e. original) goal, though we seriously doubt whether anyone will use this feature. Submit **(HELP-LONG EXIT)** for instructions regarding the use of this mechanism.

SECTION 3: HELPFUL TIPS

When I sit down to use the Theorem Prover and am ready to prove a lemma, I generally see first if it can be proved automatically (possibly with appropriate hints). If the first few seconds either give virtually no output or give output that's discouraging, I'm likely to decide then to use the interactive system by submitting **(VERIFY <lemma to be proved>)**. Sometimes I'll start with an **INDUCT** command and then try **PROVE** on each of the goals thus created. Often the base case(s) will present no problem (using **PROVE** or even **S**). Then sometimes the inductive step(s) can be proved by opening up various function calls using **X**, **X-DUMB**, or **S-PROP** and then messing around. Occasionally this approach reveals that one or more rewrite rules are really called for, in which case I may exit the interactive system, prove some rewrite rules, and then go back in with **(VERIFY)**. Occasionally the proof can actually now be done automatically in the presence of these rules.

Some people may wish to use the interactive system only as a tool for finding automatic proofs. The strategy outlined above can be used to test induction strategies. That is, suppose one thinks that a theorem should be true by induction according to **`(foo x y z)`**. One might invoke **VERIFY** and then give the command **(INDUCT (FOO X Y Z))**. By using **CHANGE-GOAL** repeatedly together with the printing command **P**, one can look at all the goals and decide if they are all true. Once convinced that they are all true, the user might then try simplifying or proving each of them, as described above -- or one might choose to focus on a particularly worrisome inductive step.

Here are some other random observations.

The Syntax of Commands

There is a general rule which can help the user remember whether a command expects an arbitrary number of arguments or a single list of arguments. Macro commands always expect a fixed number of arguments, as in

```
(USE ((APPEND-ASSOC (X A) (Y B)) (TIMES-0)))  .
```

Primitive commands, however, expect a fixed number of arguments *unless* all arguments are of the same "type". For example, we have

```
(DIVE 2 3 1)
```

rather than **(DIVE (2 3 1))**, and

```
(PROVE (DISABLE TIMES APPEND) (INDUCT (PLUS X Y)))
```

rather than **(PROVE (DISABLE TIMES APPEND) (INDUCT (PLUS X Y)))**, since **PROVE** expects a list all of whose members are hints. Compare though with

```
(GENERALIZE (((PLUS X Y) A)) MAIN-SUBGOAL)
```

which has two arguments, namely a list of term-variable pairs and a new goal name. Now in fact the new goal name is optional; however, for consistency one still would use **(GENERALIZE (((PLUS X Y) A)))** rather than **(GENERALIZE ((PLUS X Y) A))**.

Of course, the help facility is the final authority on the syntax of commands. However, this subsection is intended to save the user some time in some cases. One final note on this subject: the meta command **BIND** is an exception to the rule, since it takes an arbitrary number of arguments but the first is of a different type. We did this to make **BIND** have a syntax similar to Lisp's **LET**.

Brief Introduction to Several Useful Macro Commands

More detailed descriptions of these commands may of course be obtained by using the help facility.

The command **H** is like **HELP** except that it clears the screen first.

The command **VIEW** (equivalently, **TH**) is very handy, as it shows the hypotheses, governors, and current subterm after clearing the screen. The command **VIEW-TOP** is similar: it shows the entire conclusion instead of just the current subterm, and it highlights the current subterm.

The command **TAUT** checks to see whether the current goal is essentially a tautology. (The primitive command **SPLIT** can also be used for this purpose, though unlike **TAUT** it does not expand **ZEROP** and **NLISTP**.)

The command **GOALS** is like **GOAL-NAMES** but hides certain information which is thought to be mostly useless.

The command **ELIM** is used to perform elimination. For example, if one has a goal in which **(SUB1 X)** occurs, one may wish to use the **ELIM** to eliminate **X** in favor of **(ADD1 Z)** (for some variable **Z**).

The command **PRINT** may be used to print the value of an arbitrary Lisp form.

The command **THEN** applies a given command and then applies another command to each of the new subgoals. For example, suppose that you wish to **REWRITE** within the consequent of an implication. Try **(THEN**

REWRITE TAUT), which is best said in English as "Rewrite then Taut". This will cause **REWRITE** to be applied, and then **TAUT** will be applied to each new subgoal. Those subgoals which are trivial will be proved using **TAUT**, leaving you only with the "real" new goals. A more common use of **THEN** might be for calling the Theorem Prover on each subgoal created by **REWRITE** (or any other command that creates subgoals). The syntax would be **(THEN REWRITE PROVE+)**. But in fact, **THEN** is allowed one argument with the second defaulting to **PROVE+**, and hence **(THEN REWRITE)** is acceptable syntax.

The command **INTRO-APPLY$** may be of interest to those who use the "second-order" function **APPLY$** on the left side of rewrite rules. For, one may then wish to introduct an **APPLY$** into the current subterm so that such rules will be applicable.

The commands **HYP** and **WRAP** are explained in Section 4. Finally, the commands **CLAIM+**, **PROVE+**, and **=+** are described in the following subsection, while **S*** is discussed in the one after that.

Calling the Theorem Prover

The change commands which may call the Theorem Prover are **CLAIM**, **PROVE**, and **=**. In each of these cases, the Prover is called with a context that ignores the interactive session, i.e. the same context that existed at the start of the session. In particular, and **ENABLE** and **DISABLE** commands given during that session are irrelevant. Often, though, the user would expect that such commands would in fact be respected by the Prover. For that purpose we have provided the macro commands **CLAIM+**, **PROVE+**, and **=+**, which do respect the context provided during the interactive session. The user may wish to use these on any non-trivial calls to the Prover.

Simplification: S, S-PROP, and S*

When one wants to simplify the current subterm, each of these three commands can perhaps be used -- but which is appropriate? **S*** is a macro command which alternates between **S** and **S-PROP**, and is hence the safest to use if one wants to be sure that every "fast" simplification possible is made. However, in most cases either **S** or **S-PROP** is in fact the preferable command -- but which one? Both commands perform propositional simplification, eliminating the propositional functions **AND**, **OR**, **IMPLIES**, and **NOT** in favor of **IF** (unless these are disabled or some arguments are given to **S-PROP**). However, **S** goes further in that it expands all nonrecursive function symbols (except the disabled ones) and uses the so-called *fast rewrite rules* -- use **(HELP-LONG S)** for details about this. On the other hand, while **S-PROP** is not as powerful in that sense, it has the advantage that it normalizes expressions in the sense that **IF** is pushed to the outside. To be precise, after applying **S-PROP** it will be the case that for every proper subterm of the current subterm which has the form **(IF x y z)**, it is in fact the second or

third argument (*not* the *test*) of a term of that form. This movement of **IF**-expression to the outside is often quite helpful. For example, consider the term:

```
(EQUAL (FIX X) (IF (ZEROP X) 0 X))
```

To prove this term, one might give the command **S**; but this only results in the term

```
(EQUAL (IF (NUMBERP X) X 0)
       (IF (EQUAL X 0)
           0
           (IF (NUMBERP X) X 0)))
```

which however can then be proved with **S-PROP**. If instead **S-PROP** is done first, one obtains the simpler term

```
(IF (ZEROP X)
    (EQUAL (FIX X) 0)
    (EQUAL (FIX X) X))
```

which in fact can be proved using **S**.[11]

There are times when one wants to expand all calls of one or more function symbols without having to move to the individual relevant subterms first. In this case the command S-PROP with arguments (as explained by the **HELP-LONG** command) is useful. Finally, use **(S-PROP NIL)** when *all* you want to do is to normalize **IF**-expressions.

Case Splitting

A common strategy in all sorts of proof methodologies is the splitting of a goal into cases. Usually, one will probably want to do this with **(CLAIM <term> 0)**, where **<term>** is the term on which one want to case (according to whether or not it is **F**) and the argument **0** indicates that the theorem-prover should not attempt to prove **<term>**. The current goal will then have **<term>** as an additional hypothesis, but a new goal will be created which is identical to the current goal except that instead **(NOT <term>)** is added as a hypothesis. Moreover, the current goal now depends on the new goal (which in turn has no dependents). Other case splitting methods include using the macro command **CASE** (use the help facility for a specification) and, if **<term>** is an equality to be used for substitution, a version of the **=** command.

Proving Implications

There are at least two basic approaches to proving terms of the form **(IMPLIES <hyp> <conc>)**. The preferability of one or the other approach may well be mainly a matter of taste. One approach is to begin with the

[11] In fact, S* and PROVE both prove the original goal; this example was chosen merely to illustrate the differences between **S** and **S-PROP**.

command **PROMOTE**, which adds **<hyp>** to the hypotheses (after flattening its **AND** structure, if it is an **AND** expression). The other approach is to invoke **S** or **S-PROP** or **S***, which will create a (possibly rather large) **IF** tree and then **DIVE**, which when given no arguments will put you at a non-**T** branch of the **IF**-tree (if there is one). Often the two approaches are equivalent except that the second approach makes the assumptions governors rather than top-level hypotheses. I suppose that the first approach has the advantage of keeping the hypotheses out of the way (at the top level) and eliminating the need for **DIVE**, while the second approach has the advantage of allowing you to do simplification in the assumptions without doing the contortions involving **DEMOTE** that are ilustrated in the subsection "Simplifying Hypotheses" below. Usually I prefer the former approach, but again, that's probably just a matter of taste.

However, if there are more than a couple of cases implicit (or explicit) in the implication, I recommend the use of **SPLIT**. This command will create a goal for each case generated, roughly speaking, by **OR**-terms in the hypotheses and **AND**-terms in the conclusion (and by other appropriate propositional functions).

Substitution

A lot of the work in an interactive proof may involve substituting equals for equals. Often this is accomplished by simplification or by expanding function calls (**S**, **S-PROP**, **X**, **X-DUMB**). However, when the underlying equality seems fairly straightforward but requires some proof, it may be worthwhile using the = command. For example, suppose that the current term has the form **(IF <test> <branch1> <branch2>)**, where the user sees that **<test>** equals **T**. Although one might be tempted to expand the outermost function call of **<test>** and to do various other low-level operations, there may be an easier way: simply submit the command **(DIVE 1)** (or equivalently, just **1**) in order to make **<test>** the current term, and then submit the command **(= T)**. If the Theorem Prover is able to prove this equality, then the substitution will be made.[12] Then of course one can submit **UP** followed by **S** or **S-PROP** to replace **(IF T <branch1> <branch2>)** with simply **<branch1>** (or a simplified version thereof).

Managing Goals

Suppose that one is trying to prove a rather complicated goal. It may be that this amounts to proving that various subterms of the conclusion are equal to **T**. Perhaps one or two of these subterms seem(s) difficult to simplify. Then it may be helpful for bookkeeping purposes to use the **PUSH** command. Roughly speaking, the **PUSH** command replaces the current subterm by **T** and creates a new goal to prove the current subterm (under the current active hypotheses and governors). In this manner one can prove the goal that was under consideration

[12] If not, then the second paragraph below suggests a different form of the = command, which here would be: **(= * T 0)**.

without getting lost in details, and then go prove the goals that were **PUSH**ed in the process. Of course, one may further **PUSH** subgoals of the new goals, and so on.

A similar modularization is accomplished by using versions of **CLAIM** and **=** which allow one to postpone proof attempts; see the help facility. The **REWRITE** command is also useful for proof control, and in particular for controlling backward chaining via application of conditional rewrite rules.

When a "significant" goal is proved by pushing difficult subgoals, you may wish to insert a **BOOKMARK** instruction to make it easy to undo back to the point at which the difficult goal's proof was begun (in case the proofs of the messy subgoals get gnarled up). One can also insert comments using the **COMMENT** command.

Exiting Temporarily

Suppose that you wish to exit an interactive session without creating an event but with a record of what you have done. One way to do this is to submit the command **(EXIT <any name> NIL)** and then answer **N** to the query about adding an event. You can then grab this "event" from the dribble-buffer and run it later, followed by **(UBT)** (so that you don't actually create an event) and then **(VERIFY)** (in order to resume the proof).[13] Alternatively, you can just enter the interactive system later with **(VERIFY <term>)** and submit **(PLAY <command$_1$> ... <command$_k$>)**, where these are the change commands that were put in the **INSTRUCTIONS** hint.

What should you do if you exit the interactive system, prove some rewrite lemmas, and then go back in where you were? (By the way, this is often an excellent approach when you are confronted with a goal which follows from a relatively simple and general fact.) The issue here is that if the interactive session had involved calls to the theorem-prover, those calls may not work later in the batch re-play of events because of the presence of these rewrite rules. As mentioned in Section 2, one can create events noting the progress of an interactive session. While that solves this problem, it probably isn't necessary in a majority of cases: usually the added rewrite rules won't cause a problem. If you suspect that there isn't going to be a problem but want to make sure, the command **REPLAY** will play back all the instructions and let you know if one or more of them didn't work. Alternatively, you can finish the interactive proof and then answer **R** ("replay") to the query regarding adding the event.

Bringing in Useful Information

Suppose that you are working on a goal and you need a certain fact. How can you make that fact explicit?

---

[13] An obscure but sometimes helpful fact is that the global Lisp variable **EV0** stores the most recent **PROVE-LEMMA** expression printed in response to the user's answer to the query caused by the **EXIT** command.

There are a few ways. One is to **CLAIM** it, i.e. submit **(CLAIM <fact>)**. This will invoke the theorem-prover. If the proof fails then a new goal will be generated as though you had submitted **(CLAIM <fact> 0)** (which was alluded to above; it avoids calling the Prover); or, you may wish (or need) to submit **(CLAIM <fact> 0)** anyhow in this case. Then later, when considering the new goal, you can invoke the **CONTRADICT** command to move the **<fact>** to the conclusion, then (optionally) hide the resulting new hypothesis, and finally proceed from there to prove **<fact>** from the other (original) hypotheses.

If the fact is an instance of a lemma in the chronology, you may use the command **USE-LEMMA** or the macro command **USE**. On the other hand, if the fact is an instance of the original form of another goal, you may use **USE-GOAL**. All of these are of course described in the help facility.

Simplifying Hypotheses

Suppose that during the course of an interactive proof, one has a situation in which one wants to simplify (or make any sort of substitution in) a hypothesis. Now this system is designed for working on the *conclusion* of the current goal, and especially on the current subterm of that conclusion; so what can one do? One approach is to use **(CLAIM <new_version> 0)** to add the "new version" of the hypothesis to the hypothesis list, and then prove the correctness of this operation later (by proving the resulting subgoal). Here we describe another approach.

A simple approach is to shift a hypothesis from the hypothesis list into the conclusion, by making it the antecedent for an implication concluding with the current conclusion; one then may manipulate the hypothesis, and then move it back. (In fact there is a macro command **HYP** which uses **CONTRADICT** for this purpose; it is described in Section 4. However, let us proceed with a manual approach based on **DEMOTE**.) Here's an example.

```
->: view

*** Active top-level hypotheses:
H1.  (NUMBERP X)
H2.  (EQUAL (PLUS 0 X) Y)

*** Active governors:
There are no governors to display.

The current subterm is:
<some term>

->: (demote 2)
```

```
->: view

*** Active top-level hypotheses:
H1.  (NUMBERP X)

*** Active governors:
There are no governors to display.

The current subterm is:
(IMPLIES (EQUAL (PLUS 0 X) Y)
         <some term>)

->: 1

->: p
(EQUAL (PLUS 0 X) Y)

->: s

->: p
(EQUAL X Y)

->: top

->: view

*** Active top-level hypotheses:
H1.  (NUMBERP X)

*** Active governors:
There are no governors to display.

The current subterm is:
(IMPLIES (EQUAL X Y) <some term>)

->: promote

->: view

*** Active top-level hypotheses:
H1.  (NUMBERP X)
H2.  (EQUAL Y Z)

*** Active governors:
There are no governors to display.

The current subterm is:
<some term>
```

So now the hypothesis **(EQUAL (PLUS 0 X) Y)** has been replaced by the simpler hypothesis **(EQUAL X Y)**, and we can proceed with the proof however we choose to.

Pesky Abbreviations

As things stand currently, the functions **LEQ**, **GREATERP**, and **GEQ** are immediately expanded upon translation. This means that they will never be seen except as user input. In fact, they won't even appear in events which have been created by **EXIT**, even if you originally entered **VERIFY** with a term having such function symbols in it. Now of course you can edit this event before making it part of your events file so that the original term is the body of the event, and the proof should replay in batch mode just fine, but admittedly that's a nuisance. Of course, you can just leave the event so that your pretty **(LEQ A B)** has been replaced by **(IF (LESSP B A) F T)** in the body of your lemma, and that will work too. Sorry about the pair of undesirable choices.

SECTION 4: MACRO COMMANDS AND THE TOP LEVEL LOOP

The system described above may well be adequate for many users' needs. However, the system is in fact extensible by way of a facility called *macro commands*. The idea behind these commands is that a single macro command generates zero or more "real" commands, i.e. change, help, and meta commands. Several macro commands are currently provided by the system, and a mechanism exists for adding additional macro commands and even for redefining the given ones. The soundness of the system is not affected by macro commands, in the following sense: macro commands are ignored by the **PROVE-LEMMA** command. That is, a user may give macro commands freely during an interactive session, but in order to be secure in the correctness of the proofs constructed, the resulting events should be run through once more. The **INSTRUCTIONS** hints to **PROVE-LEMMA** events should contain no macro commands, since these are all "expanded out" during the interactive proof session. But even if a malicious user tries to sneak some such commands in to the **INSTRUCTIONS**, no harm will arise: the system will simply complain, since only *change* commands are recognized by **PROVE-LEMMA**.

This section is organized into three subsections. We begin by giving an introduction to the *use* of macro commands by describing informally how to use some of the predefined macro commands. The second subsection gives a detailed definition of the execution of the top-level loop. We conclude with examples which illustrate the *writing* of macro commands.

How to use macro commands: some examples

Suppose that you wish to call the Theorem Prover with the **PROVE** command, but you have already disabled or enabled various functions and rewrite rules during the current interactive session (with the **DISABLE** and **ENABLE** commands). Perhaps you want to call the Prover with the corresponding "environment". For example, perhaps you have begun the current session with the commands **(DISABLE TIMES-COMMUTATIVITY)** and

(ENABLE APPEND), and you are ready to call the Prover. But suppose for some reason that you want **TIMES-COMMUTATIVITY** disabled and **APPEND** enabled (even though they are presumably the other way around, say, in the global Theorem Prover state). Now one way to do this, as explained by invoking **(HELP PROVE)**, is to submit the command

```
(PROVE (DISABLE TIMES-COMMUTATIVITY) (ENABLE APPEND)) .
```

However, there is a better way. The macro command **PROVE+** has been defined to accomplish the same effect. Here is the documentation provided by submitting **(HELP PROVE+)**:

```
->: (help prove+)
Macro Command  [Use HELP-LONG to see the definition]
(PROVE+ &OPTIONAL HINTS)

Call the theorem prover with hints that match the local
disable-enable environment in the current interactive
session, as updated (optionally) by the HINTS argument.
Notice that this command has the same syntax as PROVE,
except that the hints are put into a list; compare
(PROVE (DISABLE PLUS TIMES)) versus
(PROVE+ ((DISABLE PLUS TIMES))).
```

Thus, in the example above, the command **PROVE+** will actually generate the command displayed above, namely **(PROVE (DISABLE TIMES-COMMUTATIVITY) (ENABLE APPEND))**. This **PROVE** command is then the one that is actually executed by the system: if the proof is successful, a new state will be pushed onto the state stack and its **INSTRUCTION** field will be the **PROVE** command shown above, while if the proof is not successful, there will be no change in the state stack.

The information provided above by **(HELP PROVE+)** also mentions an optional argument **HINTS**, which may be provided using essentially the same syntax as for the **PROVE** command. Consider then the example above but where we instead give the command

```
(PROVE+ ((DISABLE PLUS DIFFERENCE)
         (USE (PLUS (X A)))))
```

In this case, the command generated would be

```
(PROVE (USE (PLUS (X A)))
       (DISABLE TIMES-COMMUTATIVITY PLUS DIFFERENCE)
       (ENABLE APPEND))
```

and this would be the one that is actually executed.

Let us consider another example. Suppose that hypothesis number 2 is of the form **(IF T X Y)**, which one

would like to simplify to **X**. Now the command **S** only works on the conclusion; hence one approach is to move that hypothesis into the conclusion somehow, then simplify it there, and then move it back into its former position. The following sequence of commands accomplishes this task. (We provide some comments in italics.)

```
(CONTRADICT 2)      {Swap <hyp> hypothesis number 2, with the conclusion}
(DIVE 1)      {Dive to the first argument of (NOT <hyp>)}
S      {Simplify}
TOP      {Move to the top of the conclusion}
(CONTRADICT 2)      {Swap back the second hypothesis with the conclusion}
```

As this sequence of commands is really independent of the contents of the second hypothesis and the conclusion, one can imagine a macro command that generates this sequence of commands.[14] Such a command is in fact a predefined macro command called **HYP**. Here's its description, as provided by the help facility:

```
->: (help hyp)
Macro Command  [Use HELP-LONG to see the definition]
(HYP N INSTR)

Applies instruction INSTR to hypothesis number N.
If this "fails", then there is no change made to
the state stack, the command "fails", and a
message to that effect is printed; otherwise it
"succeeds".
```

We'll talk about "success" and "failure" in the next subsection; roughly, the submission of any command (whether change, meta, help, or macro) either "succeeds" or "fails" (we always put these notions in double-quotes to emphasize that they are technical terms). In the case of change commands, "success" means creation of a new state to push on top of the state stack, while for the other commands, the **HELP-LONG** facility gives the definitions of "success". At any rate, the sequence of commands given above (for simplifying hypothesis number 2) is in fact generated by the invocation of: **(HYP 2 S)**.

Now suppose that one gives the command **(HYP 2 S)** in the example above, but then decides that one wants to undo this command. The command **UNDO** is inadequate, since in fact five actual change commands have been given, as can be seen by giving the command **COMMANDS**. But it's not always convenient to look through the commands, and that doesn't always make it clear just how far back one wants to undo. For this purpose it is handy to use the predefined macro command **WRAP** to put bookmarks around the use of a macro command when that command generates several change commands. Here is the information provided by the help facility:

---

[14] Actually, a macro command generates a single command, but there are several *meta* commands to use as sequencers. For example, the above sequence could be put in the single command **(DO-ALL (CONTRADICT 2) (DIVE 1) S TOP (CONTRADICT 2))**.

```
->: (help wrap)
Macro Command  [Use HELP-LONG to see the definition]
(WRAP INSTR)

The difference between (WRAP instr) and instr is
that in case of success, the former will insert
two bookmarks:  (BOOKMARK (BEGIN name)) before the
first new primitive command put on the state
stack, and (BOOKMARK (END name)) after the last.
Success is the same for both, however.
```

If one submits the command **(WRAP (HYP 2 S))** in place of **(HYP 2 S)** in the example above, one obtains the same five change commands together with **(BOOKMARK (BEGIN HYP))** preceding these five and **(BOOKMARK (END HYP))** following these five. Now if one wants to undo these commands, one can submit either **(UNDO 7)** or (and here's the point of using **WRAP**) **(UNDO (BEGIN HYP))**. One final note regarding this use of **UNDO**, however: when using this (bookmark) form of **UNDO**, the undoing only goes back *up to* (and not including) the given bookmark; hence the atomic command **UNDO** still needs to be given. But have no fear -- there is also a predefined macro command **UNDO+** which undoes up to *and including* a specified bookmark!

The examples above are intended to give enough of an introduction to the use of macro commands so that the user can use the predefined macro commands with ease. One other useful thing to know is that the global Lisp variable **BASIC-MACRO-COMMAND-NAMES** stores a list of names of particularly useful macro commands, essentially the ones discussed in Section 3 above. (The user can of course use Lisp to set this variable to anything he chooses.) This variable is used by the help facility: **HELP** and **H** do not mention any macro commands other than these. Several of these are very useful right from the start (especially **H** and **VIEW**).

The top-level loop

Roughly speaking, the top-level loop has the following operational-style semantics. First, the instruction is checked to see if it is a call of a macro command, and if so, it is repeatedly expanded until the result is no longer a call of a macro command. Now the resulting instruction is executed. If the resulting command is a meta command then its execution may in fact result in the execution of any number of other commands (of any kinds); this sequencing is controlled by notions of "success" and "failure". Let us now be more precise.

It is convenient to refer to the main Lisp procedures by their names. The key ones here are called **INTERACTIVE-SINGLE-STEP** and **READ-INSTR**. Here are their specifications.[15] The discussion below can also be viewed as documentation for the code.

[15]We omit details concerning the "identification" of atomic instructions with instructions that are lists of length 1, e.g. <cmd> vs. (<cmd>).

READ-INSTR takes a (potential) instruction, which might be a call of a macro command, and returns either a call of a primitive (non-macro) command or else returns **NIL** (indicating that the instruction did not expand into a call of a primitive command). In the latter case, explanatory comments are printed to the screen.

More precisely, **READ-INSTR** is defined as follows. First, for each argument **<arg$_i$>** of the potential instruction **(**<**cmd**> <**arg$_1$**> **...** <**arg$_n$**>**)**, if that argument has the form **(! x)** then replace it by the result of evaluating **x** in Lisp. More precisely, there is a global variable called **COMMAND-BINDINGS** which contains a list of pairs each of the form **(variable . S-expression)**, and the form **x** is evaluated in this environment.[16] Even in the case that **READ-INSTR** is applied recursively (for macro commands, as explained below), arguments of the form **(! x)** are always evaluated in the sense above. In a nutshell: **READ-INSTR** causes evaluation of the "(! x)" arguments of the given instruction as explained above, and then returns an instruction as follows:

- If the instruction is a positive integer **N**, return **(DIVE N)**.
- If the instruction is a call of a macro command but the arity of the macro command does not match the number of arguments, cause an error.
- If the instruction is a call of a macro command in which the arity *does* match the number of arguments, then textually substitute the arguments for the formals into the body of the macro command's definition. (Optional arguments which are omitted are treated as **NIL** for this purpose. This substitution procedure will be discussed in more detail in the next subsection.) Then, recursively apply **READ-INSTR** to the result.
- If the instruction is a call of a primitive command (either a change, meta, help or exit command), then return the instruction.
- Otherwise, return **NIL** (and print to the screen that the given instruction is not a valid instruction).

INTERACTIVE-SINGLE-STEP is evaluated for side-effect, especially to the **STATE-STACK**. It takes an arbitrary S-expression and immediately applies **READ-INSTR** to that form. If the result of **READ-INSTR** is **NIL**, then nothing further happens (and **READ-INSTR** is responsible for printing helpful information to the screen). Otherwise, **READ-INSTR** returns a form which is a call of a legitimate command (and not a macro command). What happens next depends on the kind of command.

- If the instruction is a call of a change command, then the current state is fed to an appropriate function, named **CHANGE-STATE-WITH-<command_name>**. That function either returns a new state, which is then pushed on top of the **STATE-STACK**, or else returns **NIL** (in which case the state stack is not changed).
- If the instruction is a call of a help command, then an appropriate function (named **PRINT-HELP-WITH-<command_name>**) is called to print information to the screen.
- If the instruction is a call of a meta command, then the function **CHANGE-STATE-STACK-WITH-<command_name>** is called. That function may itself call **INTERACTIVE-SINGLE-STEP**. For example, the meta command **DO-ALL** invokes the function

[16] The meta command **BIND** modifies this environment.

CHANGE-STATE-STACK-WITH-DO-ALL, which in turn sequentially calls **INTERACTIVE-SINGLE-STEP** on each instruction in its list of arguments.

- If the instruction is **EXIT**, the top-level loop is exited.
- If the instruction is **(EXIT <name> <lemma-types>)** (or a three-argument version), the top-level loop is exited in the manner explained in the example in Section 1.
- Otherwise, the instruction is not legitimate, and there is no change in the **STATE-STACK**.

An important consideration was left out of the specifications above. Each call of **INTERACTIVE-SINGLE-STEP** returns a value which we think of as denoting "success" or "failure", according to whether that value is non-**NIL** or is **NIL** (respectively). For change commands, "success" occurs (i.e. a non-**NIL** value is returned) if and only if a new state is indeed pushed on to the state stack. (The user can tell if this is the case by checking, using the instruction **COMMANDS**, whether or not the new instruction was stored.) There is no specification of "success" and "failure" for help commands. For calls of meta and macro commands, the "success" or "failure" depends on the particular command (as specified by the **HELP-LONG** facility).[17] Various meta and macro commands cause multiple calls to **INTERACTIVE-SINGLE-STEP** which are however guarded by the "success" or "failure" of subcalls. Another important use of the "success" notion is made by the meta command **PROTECT**: **(PROTECT <instruction>)** runs the given instruction, but reverts the system to the global state existing at call time in case that instruction fails (by restoring the values of the Lisp variables **STATE-STACK** and **OLD-SS**).

Defining macro commands

In this section we present the syntax for defining macro commands. In the process, we review carefully the evaluation mechanism presented in the immediately preceding subsection above. The ideas are presented with some examples.

The syntax for defining macro commands is:

```
(DEFINSTR <command_name> <formals> <body> <documentation>)
```

where **<documentation>** is optional. The arguments to **DEFINSTR** may be described briefly as follows:

- **<command_name>** is the name that one wishes to assign to the command (e.g. **PROVE+**). It may not be the name of a primitive (i.e. change, meta, help, or exit) command.
- **<formals>** is the list of arguments to the macro command. The syntax should be a list of distinct symbols, one of which may be the symbol **&OPTIONAL**. We define the *upper arity* of the command to be the number of symbols in **<formals>** other than the (optional) symbol **&OPTIONAL**, while the *lower arity* is the number of symbols in **<formals>** which occur strictly before **&OPTIONAL** (unless **&OPTIONAL** is not in the list, in which case the lower arity is defined to equal the upper arity).

[17] As "success" and "failure" are considered "advanced" notions, the long help must be used to get such information about meta commands.

- **<body>** is an arbitrary S-expression. However, it should be an S-expression which makes sense when formal parameters are textually replaced by actual parameters (as explained in the specification of **READ-INSTR** in the subsection above).
- **<documentation>** is a list of atoms to be printed by the help facility (both **HELP** and **HELP-LONG**). This printing is done with code written by Boyer and Moore, and certain conventions apply. The atom **CR** denotes a carriage return (with line feed), while **|#|** denotes a tab. Lower-case words and punctuation need to be enclosed in vertical bars. Finally, this printing routine is intelligent about line length and punctuation. The examples below will make all of this clearer.

The most recent definition of **<command_name>** may be removed by invoking

```
(UNDEFINSTR <command_name>)
```

Let us consider some examples. We begin with the definition of a simple macro command which changes goals except when its argument is already the current goal. Explanation follows.

```
(DEFINSTR MAYBE-CHANGE-GOAL (NAME)
  (IF (EQ (QUOTE NAME) (GOAL-NAME))
      SKIP
      (CHANGE-GOAL NAME))
  (|Change| |to| |the| |goal| |with| |the| |indicated|
            |name| |,| |unless| |we| |are| |already|
            |at| |that| |goal| |.|))
```

Here, **IF** and **SKIP** are themselves macro commands, with the following documentation:

- (IF TEST INSTR1 INSTR2): Evaluate the test in the environment obtained from COMMAND-BINDINGS. Execute INSTR1 if the result is not NIL and INSTR2 otherwise.
- SKIP: Does nothing, but "succeeds".

Suppose that we are at the prompt "->: ", and we submit the instruction **(MAYBE-CHANGE-GOAL (MAIN . 1))**. Let us trace through the calls of **INTERACTIVE-SINGLE-STEP** and **READ-INSTR**, assuming that the current goal's name is not **(MAIN . 1)**. Notice that here **(GOAL-NAME)** is the call of a Lisp function **GOAL-NAME** which returns the **GOAL-NAME** of the current **GOAL** of the **STATE** on top of the state stack (cf. Section 2). (All corresponding functions returning the fields of the current state and goal are also defined.) Also important is the definition of the macro command **IF**.[18] (We omit the documentation here, as it's shown above.)

```
(DEFINSTR IF (TEST INSTR1 INSTR2)
  (BIND ((INSTR
           (IF TEST
               (QUOTE INSTR1)
               (QUOTE INSTR2))))
        (DO-ALL (! INSTR))))
```

[18]The reason for **(DO-ALL (! INSTR))** rather than simply **(! INSTR)** is the semantics of **BIND**. The extension of the environment which is given in the first argument of **BIND** is only available *inside* calls of its arguments. Unfortunately, there are several somewhat obscure places like this to make errors. Perhaps someday we will improve the macro command facility.

Now we are ready for the trace. Since anyone reading this far had better be familiar with Lisp, we make no apology for presenting the trace in a traditional Lisp format. Comments are inserted in curly braces in italics, *{<comment>}*.

```
1 Enter INTERACTIVE-SINGLE-STEP (MAYBE-CHANGE-GOAL (MAIN . 1))

| 1 Enter READ-INSTR (MAYBE-CHANGE-GOAL (MAIN . 1))

|  2 Enter READ-INSTR (IF (EQ (QUOTE (MAIN . 1)) (GOAL-NAME))
                          SKIP
                          (CHANGE-GOAL (MAIN . 1)))

|  | 3 Enter READ-INSTR (BIND ((INSTR (IF (EQ (QUOTE (MAIN . 1)) (GOAL-NAME))
                                          (QUOTE SKIP)
                                          (QUOTE (CHANGE-GOAL (MAIN . 1))))))
                             (DO-ALL (! INSTR)))

|  | 3 Exit READ-INSTR (BIND ....)

|  2 Exit READ-INSTR (BIND ....)

| 1 Exit READ-INSTR (BIND ....)

| 1 Enter CHANGE-STATE-STACK-WITH-BIND (BIND ....)

|  1 Enter CHANGE-STATE-STACK-WITH-DO-ALL (DO-ALL (DO-ALL (! INSTR)))
       {roughly, because BIND is defined in terms of DO-ALL}

|  | 2 Enter INTERACTIVE-SINGLE-STEP (DO-ALL (! INSTR))

|  |  1 Enter READ-INSTR (DO-ALL (! INSTR))

|  |  1 Exit READ-INSTR (DO-ALL (CHANGE-GOAL (MAIN . 1)))
       {by using the binding of INSTR in COMMAND-BINDINGS}

|  |  2 Enter CHANGE-STATE-STACK-WITH-DO-ALL (DO-ALL (CHANGE-GOAL (MAIN . 1)))

|  |  | 3 Enter INTERACTIVE-SINGLE-STEP (CHANGE-GOAL (MAIN . 1))

|  |  |  1 Enter READ-INSTR (CHANGE-GOAL (MAIN . 1))

|  |  |  1 Exit READ-INSTR (CHANGE-GOAL (MAIN . 1))

|  |  |  1 Enter CHANGE-STATE-WITH-CHANGE-GOAL (CHANGE-GOAL (MAIN . 1)) <state>

|  |  |  1 Exit CHANGE-STATE-WITH-CHANGE-GOAL <new state>

|  |  | 3 Exit INTERACTIVE-SINGLE-STEP <a state stack>

|  |  2 Exit CHANGE-STATE-STACK-WITH-DO-ALL T

|  | 2 Exit INTERACTIVE-SINGLE-STEP T

|  1 Exit CHANGE-STATE-STACK-WITH-DO-ALL T

| 1 Exit CHANGE-STATE-STACK-WITH-BIND T
```

1 Exit INTERACTIVE-SINGLE-STEP T

Finally, let us look at and analyze definitions of macro commands which were introduced earlier. First, **PROVE+**:

```
(DEFINSTR PROVE+ (&OPTIONAL HINTS)
  (DO-ALL
    (LISP (PROGN (CHK-ACCEPTABLE-HINTS (QUOTE HINTS)) T))
    (BIND ((HNTS (MAKE-LOCAL-HINTS (QUOTE HINTS) STATE-STACK)))
          (IF HNTS (! (CONS (QUOTE PROVE) HNTS)) PROVE))))
```

How does this macro command work? The outer command **DO-ALL** is simply a sequencer. The first command is **LISP**, which submits the form to Lisp (in an environment augmented by **COMMAND-BINDINGS**). **LISP** is called simply to check that the (optional) hints to the **PROVE+** command are appropriate; in fact, **CHK-ACCEPTABLE-HINTS** is a Lisp function written by Boyer and Moore for the Theorem Prover which checks the hints to **PROVE-LEMMA**, causing an error if it finds something it doesn't like.[19] Then the symbol **HNTS** is bound in **COMMAND-BINDINGS** to an appropriate hint list, which is the result of evaluating **(MAKE-LOCAL-HINTS (QUOTE <hints>) STATE-STACK)** for the particular **<hints>** supplied in the instruction.[20] Inside the execution of **BIND** we have a call of the macro command **IF**, which we have seen before. Here, this **IF** command evaluates **HNTS**, i.e. it considers **(MAKE-LOCAL-HINTS (QUOTE HINTS) STATE-STACK)**. If this is not **NIL** then it executes the command **(PROVE** <**hints**>**)**, where **<hints>** is the result of evaluating this call of **MAKE-LOCAL-HINTS**; otherwise it executes the command **PROVE**.

The next example presented earlier in this section was the macro command **HYP.** In fact **HYP** is defined in terms of a macro command **HYP0**; here are the definitions.

---

[19]The reason for the call of **PROGN** is simply that we want this call of **LISP** to "succeed", and as the **HELP-LONG** facility explains, this command "succeeds" if and only if the result of the Lisp evaluation is not **NIL**. But why do we want this to "succeed"? Because as the **HELP-LONG** facility explains, **DO-ALL** "succeeds" if and only if all of its subcommands "succeed"; thus in our case, that's equivalent to the **BIND** command "succeeding". That in turn is equivalent (by the specification of **BIND**) to the "success" of the **IF** command, which in turn is simply a call for execution of one of its two branches, i.e. a call to **PROVE**. And that's what we desire, i.e. the "success" of a call to **PROVE+** should be equivalent to the "success" of the corresponding call to **PROVE** (as explained by the help facility).

[20]Currently, the file "macro-commands-aux" contains definitions of Lisp functions written for the macro commands, which themselves are in the file "macro-commands". The user should feel free to add to these files; as long as they aren't loaded in the final check of a sequence of events, soundness won't be affected. In fact, unless functions like **PROVE-LEMMA** are redefined in these files, soundness won't be affected even if they are loaded.

```
(DEFINSTR HYP (N INSTR)
  (ORELSE
    (HYP0 N INSTR)
    (PROG2 (PRINT "HYP failed")
           FAIL)))

(DEFINSTR HYP0 (N INSTR)
  (PROTECT
    (DO-STRICT
      ((CONTRADICT-DUMB N)
       (DIVE 1)
       (DO-ALL (! (SUBST (QUOTE (DO-ALL TOP 1))
                         (QUOTE TOP)
                         (QUOTE INSTR))))
       TOP
       (CONTRADICT? N)))))
```

The reader is invited to use the help facility (especially **HELP-LONG**) to learn about the macro commands called inside the definitions of **HYP** and **HYP0**. But here is a brief explanation of these commands. FIrst, **(HYP N INSTR)** calls for the execution of **(HYP0 N INSTR)**. **ORELSE** is defined so that if this call of **HYP0** "succeeds", then so does the call of **HYP** and nothing further happens. Otherwise, the macro command **PRINT** is called to let us know that the call to **HYP** in fact failed (and the entire call "fails" because of the definition of **PROG2**). Next consider **HYP0**. **HYP0** has a call of **PROTECT** wrapped around its body, so that if it does not "succeed" then no change is made to the global state. Inside that, a call of **DO-STRICT** is made to sequence the commands so that once a command "fails", no further execution is made of commands in the sequence and the entire sequence is deemed to have "failed". Then five commands are given in the list fed to **DO-STRICT**, and behave as explained earlier in this section. The first and last of the five commands are appropriate versions of **CONTRADICT**, and the call of **DO-ALL** is there to create an instruction which is the same as the given instruction, except that the atom **TOP** is replace everywhere by **(DO-ALL TOP 1)**. The idea here is that if the conclusion is of the form **(NOT HYP)**, where **HYP** was a hypothesis before invocation of a **CONTRADICT** command, and if one then wants to run **INSTR** on **HYP**, one wants any occurrence of **TOP** to lift one only to the top of **HYP**, not to the top of **(NOT HYP)**. Such mundane considerations often arise when writing macro commands, which is why this entire section has been considered optional (as explained in the first sentence of the report).

Appendix A
A small but somewhat realistic example

Here's a sample session showing how I actually use this system. I'll use it to verify the commutativity of **TIMES**. This is a good example because a novice user of the Boyer-Moore Theorem Prover is likely to find it to be a bit awkward to prove this lemma. The interactive mode helps one discover the facts needed to prove this lemma (after which one could, in fact, get the proof through without the interactive enhancement).

```
(verify (equal (times x y) (times y x)))

->: induct

Inducting according to the scheme:

      (AND (IMPLIES (ZEROP X) (p X Y))
           (IMPLIES (AND (NOT (ZEROP X)) (p (SUB1 X) Y))
                    (p X Y)))

Creating 2 new subgoals, (MAIN . 1) and (MAIN . 2).

The proof of the current goal, MAIN, has been completed.  However, the
following subgoals of MAIN remain to be proved: (MAIN . 1) and
(MAIN . 2).
Now proving (MAIN . 1).

->: view  {A "macro command" which clears the screen first}

*** Active top-level hypotheses:
There are no top-level hypotheses to display.

*** Active governors:
There are no governors to display.

The current subterm is:
(IMPLIES (ZEROP X)
         (EQUAL (TIMES X Y) (TIMES Y X)))
```

```
->: prove
```
{We call the Boyer-Moore theorem prover, since this looks easy.}

```
***** Now entering the theorem prover *****:

This formula simplifies, expanding the functions ZEROP, EQUAL, and
TIMES, to two new conjectures:
```

{And so on -- I'll omit the output of the theorem-prover. It does an induction and then two sub-inductions, but it only takes a few seconds.}

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


The current goal, (MAIN . 1), has been proved, and has no dependents.
Now proving (MAIN . 2).

->: view

*** Active top-level hypotheses:
There are no top-level hypotheses to display.

*** Active governors:
There are no governors to display.

The current subterm is:
(IMPLIES (AND (NOT (ZEROP X))
              (EQUAL (TIMES (SUB1 X) Y)
                     (TIMES Y (SUB1 X))))
         (EQUAL (TIMES X Y) (TIMES Y X)))

->: promote

->: view

*** Active top-level hypotheses:
H1.  (NOT (ZEROP X))
H2.  (EQUAL (TIMES (SUB1 X) Y)
            (TIMES Y (SUB1 X)))

*** Active governors:
There are no governors to display.

The current subterm is:
(EQUAL (TIMES X Y) (TIMES Y X))

->: 1
```
{Move to the first argument of **EQUAL***}*

```
->: x
```
{Recall that **X** *simplifies}*

```
->: p
(PLUS Y (TIMES (SUB1 X) Y))

->: 2
```

```
->: p
(TIMES (SUB1 X) Y)

->: =
```
{Apply the inductive hypothesis}
```

->: p
(TIMES Y (SUB1 X))

->: top

->: p
(EQUAL (PLUS Y (TIMES Y (SUB1 X)))
       (TIMES Y X))



->: hyps

*** Active top-level hypotheses:
H1.  (NOT (ZEROP X))
H2.  (EQUAL (TIMES (SUB1 X) Y)
            (TIMES Y (SUB1 X)))

*** Active governors:
There are no governors to display.

->: (hide-hyps 2)

->: prove


***** Now entering the theorem prover *****:

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

      (IMPLIES (AND (NOT (EQUAL X 0)) (NUMBERP X))
               (EQUAL (PLUS Y (TIMES Y (SUB1 X)))
                      (TIMES Y X))).
```

{I'll omit the rest of the theorem-prover's output. It does a straightforward proof by induction.}

```
The current goal, (MAIN . 2), has been proved, and has no dependents.

*!*!*!*!*!*!*  All goals have been proved!  *!*!*!*!*!*!*
You may wish to EXIT -- type (HELP EXIT) for details.
```

Appendix B
Soundness

In this appendix we argue that if the interactive system certifies a term to be a theorem of the Boyer-Moore logic, i.e. the appropriate interactive **PROVE-LEMMA** event completes successfully, then that term is indeed a theorem of the logic (relative to the existing list of events, of course). We say "argue" rather than "prove" because we sluff over some important issues:

- correctness of Boyer and Moore's code for their Theorem Prover;
- correctness of the code for the interactive system;
- details involving preservation of the invariant (*) from Section 1 (reviewed below)

However, we do feel that there is value in at least *stating* the theorem below and in presenting an outline that could in principle be fleshed out to a rigorous proof. First we need a definition.[21]

DEFINITION. A *valid state stack* is a state stack which can be produced from an interactive session which begins with a call of the form **(VERIFY <term>)** and then results from the execution of a sequence of change commands.

And now, one more definition.

DEFINITION. Let's say that a goal is *provable* if its hypotheses (hidden and active) provably imply its current conclusion, i.e. the appropriate implication is provable. (If there are no hypotheses then we simply mean that the conclusion is provable.)

We wish to prove the following theorem:

THEOREM. Suppose that **G** is a goal of the top state in a valid state stack, and suppose that **G** and all of its dependents have conclusions of **T**. Then the original version of **G** is provable.

In particular, if a sequence of change commands results in a state in which every goal has a conclusion of **T**, then the original term given to **VERIFY** is provable.

---

[21]Again, we beg the reader to forgive us. The following definition is clearly much more operational than might be desired. However, we believe it to be a routine (though very tedious) exercise to give a purely logical specification of each command's action so that this definition can itself be purely logical.

To prove this theorem we begin by recalling property (*) from Section 1. As we mentioned above, we will omit the proof of this property, which we believe to be without deep content but rather to depend on careful tedious thinking about the code.

(*) *When invoking a proof command, the goal follows from its modified version together with the subgoals that are created.*

Here, a goal is said to "follow from" others if the provability of the others implies the provability of the goal. Now property (*) trivially implies the following key lemma, which says (roughly) that each state-changing operation preserves validity in reverse.

LEMMA. Fix a valid state stack. Let **G** be a goal in a given state **S** of that state stack, let **S'** be the next state in the stack (thus created from **S** by a change command), and let **G'** be the version of **G** in **S'**. Suppose that all subgoals of **G'** in **S'** are provable in their original forms[22] and that **G'** is provable. Then **G** is provable.

We may now complete the proof of the theorem, which we do by contradiction. Fix a valid state stack for which the theorem fails. By acyclicity of the goal graph (at the top state in the stack), there is a goal **G** for which all original versions of its (final) subgoals are provable yet its original version is not provable. Now the set of subgoals of any previous version of **G** is contained in the set of subgoals of the final version of **G**, and hence the lemma above implies (by a trivial induction argument) that every previous version of **G** is provable. In particular, the original version of **G** is provable, which contradicts our choice of **G**.

[22]that is, in the sense of the field **ORIG-CONC-AND-HYPS** described in Section 2: the goal as it was when it was created

Appendix C
A Listing of the Help Facility

Here is a list of the messages printed in response to commands of the form **(HELP <command>)**. For more details (which probably will not be necessary too often), the user is advised to use **(HELP-LONG <command>)**. This appendix is organized as shown by invoking HELP-LONG (with no arguments):

```
STATE CHANGE (PROOF) COMMANDS:
  Subterm manipulation commands:
           DIVE UP NX BK TOP
  Goal manipulation commands:
           CHANGE-GOAL HIDE-HYPS HIDE-GOVS SHOW-HYPS SHOW-GOVS CLAIM
           PROMOTE DEMOTE DROP CONTRADICT USE-GOAL USE-LEMMA
  Goal creation commands:
           PUSH GENERALIZE INDUCT SPLIT
           (and sometimes CLAIM, REWRITE, =, and USE-GOAL)
  Simplification/replacement commands:
         X X-DUMB SUBV S S-PROP = PROVE REWRITE
  Fast rewrite enable/disable commands:
         ENABLE DISABLE
  Bookmark command:
           BOOKMARK COMMENT
  Abbreviation commands:
         ADD-ABBREVIATION REMOVE-ABBREVIATIONS
HELP COMMANDS:
         PP P PP-TOP GOAL-NAMES PRINT-GOAL HYPS SHOW-REWRITES
         SHOW-INDUCTIONS COMMANDS SHOW-ABBREVIATIONS HELP HELP-LONG
META COMMANDS:
         UNDO RESTORE LISP REPLAY PLAY BIND DO-ALL PROG2 NEGATE
         PROTECT SUCCEED
Exit command:  EXIT

(We omit macro commands in this appendix.  Some of the more useful
predefined macro commands are discussed in Section 3.)

-----------------------------------------------
-----------------------------------------------
```

(DIVE N): For N>0, move to the Nth argument.
EXAMPLE: If the current subterm is
(TIMES (PLUS A B) C),
then after (DIVE 1) it is
(PLUS A B).

DIVE: Dive down to the leftmost unproved branch in the IF-structure (from the top of the conclusion, unless the current term is an IF term).
EXAMPLE: If the conclusion is
(IF A (IF B T (G X)) Y),
then DIVE puts the current subterm at (G X).

NOTE: N is an abbreviation for (DIVE N), so we could have typed 1 instead of (DIVE 1) in the first example above.

OTHER FORMS (see HELP-LONG): (DIVE N1 ... Nk).

UP: Move up to enclosing subterm.
EXAMPLE: If the conclusion is
(G (H X (FOO Y)) Z)
and the current subterm is
(FOO Y),
then after executing UP, the current subterm will be
(H X (FOO Y)).

NX: Move forward one argument in the enclosing term.
EXAMPLE: If the conclusion is
(G (H X (FOO Y)) Z)
and the current subterm is X, then after executing NX, the current subterm will be (FOO Y).

BK: Move backword one argument in the enclosing term.
EXAMPLE: If the conclusion is
(G (H X (FOO Y)) Z)
and the current subterm is (FOO Y), then after executing BK, the current subterm will be X.

TOP: Move to the top of the current conclusion.
EXAMPLE: If the conclusion is
(G (H X (FOO Y)) Z)
and the current subterm is (FOO Y), then after executing TOP, the current subterm will be
(G (H X (FOO Y)) Z).

(CHANGE-GOAL <goal-name>): Change to the goal with the name <goal-name>, i.e. make it the current goal.
EXAMPLE: (CHANGE-GOAL (MAIN . 1))

CHANGE-GOAL: Change to the first unproved goal (in the list shown by the command GOAL-NAMES).

HIDE-HYPS: Hide all of the hypotheses.

(HIDE-HYPS ...): Hide the indicated hypotheses, referenced by number. To obtain these numbers, type HYPS or (HYPS ALL NIL).
EXAMPLE: (HIDE-HYPS 2 4).

HIDE-GOVS: Hide all of the governors.

(HIDE-GOVS ...): Hide the indicated governors, referenced by number. To obtain these numbers, type HYPS or (HYPS NIL ALL).

SHOW-HYPS: Activate all of the top-level hypotheses.

(SHOW-HYPS ...): Activate the indicated top-level hypotheses, referenced by number. To obtain these numbers, type (HYPS ALL NIL).
EXAMPLE: (SHOW-HYPS 2 4).

SHOW-GOVS: Activate all of the governors.

(SHOW-GOVS ...): Activate the indicated governors, referenced by number. To obtain these numbers, type (HYPS NIL ALL).
EXAMPLE: (SHOW-GOVS 2 4).

(CLAIM exp): attempt to prove exp from the currently active hypotheses (using the theorem prover). If this can be done, then add exp as a hypothesis. Otherwise, add it as a hypothesis anyhow, but also also add a new subgoal which is identical to the (previous) current goal except for the added assumption that exp is false.
EXAMPLE: (CLAIM (NUMBERP (PLUS X Y))) will simply add (PLUS X Y) as a hypothesis in the current goal. However, if one submits the command (CLAIM (NUMBERP X)) and if this cannot be proved by the prover (from the other hypotheses), then in addition to adding this hypothesis to the current goal, a new subgoal will be created with the extra hypothesis (NOT (NUMBERP X)).

OTHER FORMS (see HELP-LONG):
(CLAIM exp (hint1 hint2 ... hintK)) {to give hints to the prover}
(CLAIM exp TAUT) {to replace the prover call with a tautology check}
(CLAIM exp 0) {to turn off all proof checking, thus forcing a case split}

PROMOTE: Replace (IMPLIES hyps exp) with simply exp, adding hyps to the list of hypotheses (after flattening its AND structure, if it is an AND expression).
EXAMPLE: If the conclusion is (IMPLIES (AND X Y) Z), then after execution of PROMOTE, the conclusion will be Z and the terms X and Y will be hypotheses.

(DEMOTE n1 n2 ... nk): Replace the current conclusion with the goal of proving that the conjunction of the indicated hypotheses implies the current conclusion, and drop these hypotheses. Note that this command may only be used when at the top of the conclusion.

DEMOTE: As above, but demote all the active hypotheses.
EXAMPLE: If the active hypotheses are
H1. (LESSP X Y)
H2. (NOT (ZEROP Z))
and the conclusion and current subterm are both
(LESSP (TIMES X Z) (TIMES Y Z)),
then after executing DEMOTE, there will be no active hypotheses and the conclusion will be

```
(IMPLIES (AND (LESSP X Y) (NOT (ZEROP Z)))
         (LESSP (TIMES X Z) (TIMES Y Z)))
```


DROP: Drop all the top-level hypotheses.

(DROP n1 ... nk): Drop the top-level hypotheses with the indicated indices.
EXAMPLE: (DROP 2 5).

(DROP HIDDEN): Drop the hidden top-level hypotheses.

(CONTRADICT n): Negate the current conclusion and make it the n-th hypothesis, while negating the current n-th hypothesis and making it the current conclusion.
EXAMPLE: If the active hypotheses are
H1. (LESSP X Y)
H2. (NOT (ZEROP Z))
and the conclusion and current subterm are both
(LESSP (TIMES X Z) (TIMES Y Z)),
then after executing (CONTRADICT 2), the hypotheses are
H1. (LESSP X Y)
H2. (NOT (LESSP (TIMES X Z) (TIMES Y Z)))
and the conclusion is
(ZEROP Z).

(USE-GOAL goal-name ((var1 term1) ... (varK termK))): Add the indicated instance of the ORIGINAL version of the indicated goal as a hypothesis of the current goal (eliminating hypotheses of the indicated goal that are already known). However, this must not introduce a cycle in the dependency structure of the goals. EXAMPLE: If (MAIN . 3) is the name of a goal with hypothesis (P X) and conclusion (Q X), then after executing (USE-GOAL (MAIN . 3) ((X A))), the implication (IMPLIES (P A) (Q A)) would be added to the list of hypotheses of the current goal. However, there would be no change if (MAIN . 3) either is the current goal or depends on the current goal. Also, if (P A) is already an active hypothesis of the current goal, then the hypothesis (Q A) would be added instead.

(USE-GOAL goal-name): Same as (USE-GOAL goal-name ()).

OTHER FORM (see HELP-LONG): USE-GOAL.

(USE-LEMMA name ((var1 term1) ... (varK termK))): Add the given lemma or axiom from the chronology to the list of hypotheses, after instantiating it according to the given substitution.
EXAMPLE: Invocation of (USE-LEMMA TIMES-0 ((X A))) will add the hypothesis (EQUAL (TIMES A 0) 0) if the lemma TIMES-0 (from the chronology) is (EQUAL (TIMES X 0) 0).

USE-LEMMA: Same as (USE-LEMMA ()).

PUSH: Replace the current subterm with T (assuming it's boolean), and create a new goal to prove that term under the active hypotheses and governors.
EXAMPLE: Suppose that we have the single active hypothesis and single active governor
H1. (LESSP X Y)
G1. (NOT (ZEROP Z)),
and conclusion
(IF (ZEROP Z) T (LESSP (TIMES X Z) (TIMES Y Z)))
with current subterm
(LESSP (TIMES X Z) (TIMES Y Z)).
Then after the execution of PUSH, the current subterm is T, the new conclusion is
(IF (ZEROP Z) T T),
and the new subgoal has hypotheses
H1. (LESSP X Y)
H2. (NOT (ZEROP Z))
and conclusion
(LESSP (TIMES X Z) (TIMES Y Z)).

OTHER FORMS (see HELP-LONG): (PUSH name).

(GENERALIZE ((term1 V1) ... (termn Vn))): Replace each of the given terms by the indicated new variable (which must not occur anywhere in the current goal). A question mark (?) may be used in place of any or all variables Vi.
EXAMPLE: (GENERALIZE (((PLUS X Y) ?))).

OTHER FORMS (see HELP-LONG):
(GENERALIZE ((term1 V1) ... (termn Vn)) new-goal-name).

(INDUCT (g v1 ... vn)): Induct as in the corresponding INDUCT hint given to the theorem prover, creating new subgoals for the base and induction steps.
EXAMPLE: If the current conclusion is of the form (P X) and there are no hypotheses, then after (INDUCT (TIMES X Y)), there are two new subgoals with conclusions
(IMPLIES (ZEROP X) (P X)) and
(IMPLIES (AND (NOT (ZEROP X)) (P (SUB1 X))) (P X))

INDUCT: Induct according to a scheme chosen by the theorem prover's heuristics, creating subgoals for the base and induction steps.

SPLIT: Replace the current goal by the cases generated from its propositional structure. This command can only be used at the top of the conclusion. Roughly speaking, this command does OR-splitting in the hypotheses and AND-splitting in the conclusion.
EXAMPLE: Suppose that there is a hypothesis of the form (IMPLIES (AND P Q) R). Then a subgoal would be generated replacing this hypothesis by (NOT P), and similarly there would be one for (NOT Q) and for R. Of course, there could be more subgoals generated if P, Q, or R had additional propositional structure or if other hypotheses or the conclusion generated further cases.

X: Expand function call at current subterm, and simplify.
EXAMPLE: If the current subterm is
(APPEND (CONS A X) Y),
then after executing X the current subterm will be
(CONS A (APPEND X Y)).

X-DUMB: Expand function call at current subterm without simplifying the result.
EXAMPLE: If the current subterm is
(APPEND (CONS A X) Y),
then after executing X the current subterm will be

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


(SUBV (v1 t1) ... (vn tn)): Do parallel substitution of each term ti for the respective variable vi into the current subterm, if justified by the current active hypotheses and governors.
EXAMPLE: If the equalities (EQUAL X (PLUS A B)) and (EQUAL (TIMES C D) Y) are among the active hypotheses, and if the current subterm is
(DIFFERENCE X Y),
then after executing
(SUBV (X (PLUS A B)) (Y (TIMES C D))),
the current subterm will be
(DIFFERENCE (PLUS A B) (TIMES C D)).

SUBV: As above, but every such substitution will be performed.
EXAMPLE: In the example above, SUBV would have had the same effect.

S: Simplify the current subterm, expanding nonrecursive function calls and doing various nice things. (For details, use (HELP-LONG S).)
EXAMPLE: (ZEROP X) simplifies to
(IF (EQUAL X 0) T (IF (NUMBERP X) F T))
if there are no active hypotheses or governors, but in the presence of the hypotheses (NOT (EQUAL X 0)) it simplifies to
(IF (NUMBERP X) F T).

OTHER FORMS (see HELP-LONG): (S ALL).

S-PROP: Expand all calls of IMPLIES, AND, OR, and NOT in the current subterm, and then push all calls of IF to the top of the current subterm.

OTHER FORMS (see HELP-LONG): (S-PROP NIL), (S-PROP x1 ... xn).

= : Make a substitution for the current subterm, using an equality which is explicitly among the current active hypotheses and governors.
EXAMPLE: If the current subterm is (PLUS X Y) and the equality (EQUAL (TIMES U V) (PLUS X Y)) is an active hypothesis, then after executing the = command, the current subterm will be (TIMES U V) (unless perhaps some other hypothesis equates (PLUS X Y) with something).

(= Exp): Replace the current subterm by Exp, if they can be proved equal by the theorem prover under the current active hypotheses and governors.
EXAMPLE: (= T) will replace the current subterm with T if the prover can prove their equality.

(= Exp1 Exp2): Replace Exp1 by Exp2 in the current subterm, if they can be proved equal by the theorem prover under the current active hypotheses and governors.

OTHER FORMS (see HELP-LONG): In the following one may use * in place of Exp1 if it's the current subterm.
(= Exp1 Exp2 (hint1 hint2 ... hintK)): for giving hints to the prover) ;
(= Exp1 Exp2 <atom>): Use tautology checking instead of calling prover, and create new goal if this fails.

PROVE: Attempt to prove the current goal.
(PROVE hint1 ... hintk): as above, where each hint is as in PROVE-LEMMA and the theorem prover is to use these hints as it does with PROVE-LEMMA.
EXAMPLE:
(PROVE (DISABLE TIMES APPEND) (INDUCT (PLUS X Y))).

(REWRITE N): Replace the current subterm with a new term by using the Nth rewrite rule, as displayed by the command SHOW-REWRITES.
EXAMPLE: Suppose that the current subterm is (REVERSE (REVERSE X)). The command SHOW-REWRITES might result in the following being printed to the screen:

```
1. REVERSE-NLISTP
New term: NIL
Hypotheses: ((NOT (LISTP (REVERSE X))))
2. REVERSE-REVERSE
New term: X
Hypotheses: ((PLISTP X))
```

Then (REWRITE 2) results in a new current subterm of X together with a new subgoal of proving (PLISTP X) under the current goal's active hypotheses.
OTHER FORMS (see HELP-LONG):
One may specify the name of the rewrite rule instead of the number. One may also specify a substitution for the free variables, specify a subterm to rewrite, or specify the result of the rewrite. Also, REWRITE (with no arguments) may be used to apply the first hypothesis-free rewrite rule (or, just the first rule if all have hypotheses to relieve), as displayed by SHOW-REWRITES. Use (HELP-LONG REWRITE) to get the details.

(ENABLE <name1> ... <nameK>): Enable the given fast rewrite rules and function definitions for the fast rewriter. (These terms are explained with (HELP-LONG S).)
EXAMPLE USE: (ENABLE APPEND ZEROP).

ENABLE: Enable all fast rewrite rules and function definitions.

(DISABLE name1 ... nameK): Disable the given fast rewrite rules and function definitions for the fast rewriter. (These terms are explained with (HELP-LONG S).)
EXAMPLE USE: (DISABLE APPEND ZEROP).

DISABLE: Disable all fast rewrite rules and function definitions.

(BOOKMARK x): Makes no change in the state except to insert this instruction, for the purpose of possible UNDOing later.
EXAMPLE: The command (BOOMARK HOWDY) creates an instruction such that if later one submits (UNDO HOWDY), then the state stack will be unwound back to (but not including) this bookmark.

(COMMENT ....): A no-op, except that the instruction field of the new state contains the given instruction.
EXAMPLE: (COMMENT HI THERE HUMAN) makes this instruction the new instruction, as shown e.g. by executing COMMANDS.

OTHER FORMS (see HELP-LONG):
COMMENT makes a comment which displays the current subterm.

(ADD-ABBREVIATION <var> <exp>): Add the abbreviation which displays <exp> as <var>.
NOTE: The variable <var> should start with the character @.
EXAMPLE: (ADD-ABBREVIATION @V (TIMES X Y)) causes future occurrences of (TIMES X Y) to be printed as @V. Moreover, user input can henceforth contain the symbol @V as an abbreviation for (TIMES X Y).

REMOVE-ABBREVIATIONS: Remove all abbreviations.

(REMOVE-ABBREVIATIONS var1 var2 ... varK): Remove the indicated abbreviations.
EXAMPLE: (REMOVE-ABBREVIATIONS @X @Y) will cause future proof states to be unaware of @X and @Y as abbreviations (unless they are restored by future invocations of ADD-ABBREVIATION).

PP: Prettyprint current subterm without any special use of pretty-printing abbreviations. So for example, (AND x y z) is printed as (AND x (AND y z)); compare with P.

P: Prettyprint current subterm in the usual manner. So for example, (AND x y z) is printed rather than (AND x (AND y z)); compare with PP.

PP-TOP: Prettyprint entire term, highlighting current subterm. Printing is with respect to the same conventions as for the command PP.
EXAMPLE: If the conclusion is (EQUAL (PLUS X X 0) (TIMES X 2)) and the current subterm is (PLUS X X 0), then PP-TOP will cause the printing of:

```
(EQUAL (*** (PLUS X (PLUS X 0)) ***)
       (TIMES X 2))
```


GOAL-NAMES: Print the name of each goal, current goal first, showing dependencies.

OTHER FORMS (see HELP-LONG): (GOAL-NAMES goal-name)

(PRINT-GOAL goal-name): Print the original and the current conclusion and hypotheses of the given goal-name.

PRINT-GOAL: similarly for the current goal.

HYPS: Print the current active hypotheses, both top-level and governors.

(HYPS ALL ALL) As above, but show the hidden hypotheses and governors as well.
EXAMPLE: If the current hypotheses are (EQUAL X1 Y) and (EQUAL X2 Y) (in that order), where the first of these is hidden but the second is active, and there are no governors, then (HYPS ALL ALL) will display:

```
*** Top-level hypotheses (with "*" when hidden):
*H1.  (EQUAL X1 Y)
H2.  (EQUAL X2 Y)

*** Governors (with "*" when hidden):
There are no governors to display.
```

while HYPS will simply display:

```
*** Active top-level hypotheses:
H2.  (EQUAL X2 Y)

*** Active governors:
There are no governors to display.
```

OTHER FORM (see HELP-LONG): (HYPS (H1 H2 ... Hk) (G1 G2 ... Gn))

SHOW-REWRITES: Show all the rewrite rules which apply to the current subterm, including the name, resulting subterm, and hypotheses whose relieving will require the generation of new subgoals.
EXAMPLE: Suppose that the current subterm is (LESSP A (PLUS A 3)) and that there is a rewrite rule for transitivity of LESSP. Then SHOW-REWRITES might display the following:

```
1. LESSP-TRANS
New term: T
Hypotheses: ((LESSP A $Y) (LESSP $Y (PLUS A 3)))
```

The dollar sign indicates a free variable, i.e. a variable which occurs in the hypotheses of the rewrite rule (or, in rare cases, the right side of the conclusion of the rule) but does not occur in the left side of the conclusion of the rule. Submit (HELP REWRITE) to see the role of free variables in applying the REWRITE command.

OTHER FORM (see HELP-LONG): (SHOW-REWRITES term).

SHOW-INDUCTIONS: Show all the heuristically-chosen induction schemes.

COMMANDS: Show all of the previous commands, in reverse order.
EXAMPLE: Here is some possible output upon execution of COMMANDS:

```
The commands thus far (in reverse order, i.e. last one first) have been:
1. (CLAIM (LESSP A B))
2. (REWRITE LESSP-TRANS)
3. PROMOTE
4. START
```

Thus, there have been three commands so far, with the CLAIM command being the most recent.

(COMMANDS n): Show the last n previous commands, in reverse order.

SHOW-ABBREVIATIONS: Show all abbreviations.
EXAMPLE OUTPUT:

```
@Y corresponds to
     (TIMES B C)

@X corresponds to
     (PLUS A @Y)
       i.e. to
     (PLUS A (TIMES B C))
```

OTHER FORM (see HELP-LONG): (SHOW-ABBREVIATIONS var1 var2 ... varK).

HELP: Print names of selected instructions, including a selected set of predefined macro commands. NOTE: To get the names of all instructions and all macro commands, use HELP-LONG.

(HELP instr1 ...): Print short help on instr1 ..., generally with examples. NOTE: Use HELP-LONG to get more complete information, but without examples.

HELP-LONG: Print names of all instructions, including all macro commands. For a shortened list appropriate for people new to this system, use HELP instead.

(HELP-LONG instr1 ...): Print help on instr1 .... NOTE: Use HELP to get less detailed information, but with examples, which is perhaps more appropriate for people who are just beginning to use this system.

(UNDO N): Undo N instructions.
EXAMPLE: (UNDO 3) undoes the last three instructions.

UNDO: Undo one instruction.

OTHER FORM (see HELP-LONG): (UNDO x), where x is a bookmark.

RESTORE: Restores the state stack to its value just before the immediately preceding RESTORE or UNDO, if any.
EXAMPLE: After successfully executing (UNDO 3), the execution of RESTORE will put the state-stack back to its value before that UNDOing, provided that there is no intervening invocation of UNDO or RESTORE.

(LISP form): Evaluate the given Lisp form.
EXAMPLE: (LISP (PROGN (PRINT (+ 2 5)) (TERPRI NIL))) will cause the numeral 7 to be printed, followed by a newline.

REPLAY: This command is used to check that everything is in order, by re-running all the instructions from the start.

(PLAY instr1 instr2 ...): Play the given CHANGE instructions from the current state, stopping if any fails and otherwise noting if all instructions run successfully.
EXAMPLE: (PLAY INDUCT (CHANGE-GOAL (MAIN . 2)) S) will cause the three given instructions to be run, as long as they each create new states.

(BIND bindings instr1 instr2 ...): This instruction is only of interest to writers of macro commands -- execute (HELP-LONG BIND) if you really care.

(DO-ALL instr1 instr2 ...): Run all of the given instructions.
EXAMPLE: (DO-ALL UNDO DIVE (PLAY S S-PROP)).

(PROG2 instr1 instr2): This instruction is only of interest to writers of macro commands -- execute (HELP-LONG PROG2) if you really care.

(NEGATE instr): This instruction is only of interest to writers of macro commands -- execute (HELP-LONG NEGATE) if you really care.

(PROTECT instr): Treats instr as an atomic entity, in the sense that if it is a macro command, then either instr "succeeds" or else the global state reverts to what it was before invocation of the PROTECT command.
EXAMPLE: (PROTECT (DO-ALL DIVE PUSH TOP S)) will either cause execution of all instructions DIVE, PUSH, TOP, and S (in that order), or else will be a no-op on the global state (in case any of these four instructions does not create a new state).

(SUCCEED instr): This instruction is only of interest to writers of macro commands -- execute (HELP-LONG SUCCEED) if you really care.

(EXIT name lemma-types): Exit the proof checker, printing out the appropriate PROVE-LEMMA event with the given name and types (e.g. (REWRITE) or NIL).
EXAMPLE: (EXIT TIMES-ASSOCIATIVITY (REWRITE)).

EXIT: Quit the proof checker without making an event. In this case, (VERIFY) may be executed to get back in at the same state, as long as there hasn’t been an intervening use of the proof-checker.

References

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

2. Robert S. Boyer and J Strother Moore, ‘‘The User’s Manual for A Computational Logic (Draft)’’, Tech. report 55, Institute for Computing Science, University of Texas at Austin, March 1987.

3. M. J. Gordon, A. J. Milner, and C. P. Wadsworth, *Edinburgh LCF,* Springer-Verlag, New York, 1979.

4. R.S. Boyer and J S. Moore, *Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures,* Academic Press, 1981, pp. 103-185.

Table of Contents