The Boyer-Moore Theorem Prover and Its Interactive Enhancement

Robert S. Boyer
University of Texas, Austin
Computational Logic, Inc.

Matt Kaufmann
Computational Logic, Inc.

J Strother Moore
Computational Logic, Inc.

July 12, 1993

Abstract. The so-called "Boyer-Moore Theorem Prover" (otherwise known as "Nqthm") has been used to perform a variety of verification tasks for two decades. We give an overview of both this system and an interactive enhancement of it, "Pc-Nqthm," from a number of perspectives. First we introduce the logic in which theorems are proved. Then we briefly describe the two mechanized theorem proving systems. Next, we present a simple but illustrative example in some detail in order to give an impression of how these systems may be used successfully. Finally, we give extremely short descriptions of a large number of applications of these systems, in order to give an idea of the breadth of their uses. This paper is intended as an informal introduction to systems that have been described in detail and similarly summarized in many other books and papers; no new results are reported here. Our intention here is merely to present Nqthm to a new audience.

This research was supported in part by ONR Contract N00014-91-C-0130 and the Advanced Research Projects Agency, ARPA Order 7406. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Office of Naval Research, the Advanced Research Projects Agency, or the U.S. Government.

Introduction

In 1972, at the Metamathematics Unit of the University of Edinburgh, Scotland, Boyer and Moore began work on what has become known as the *Boyer-Moore Theorem Prover*. The mathematical logic behind our work was largely inspired by McCarthy's seminal papers on the logic of Lisp, including [McC60b], [McC62b], and [McC63]. See also [Bur69], [Goo64], and [Sko67] for related work on quantifier free logics for arithmetic. Since Lisp may be viewed simultaneously as a logic and an applicative programming language, it is a natural vehicle for the expression of theorems about computations and constructive mathematics in general. Our theorem proving style has been inspired by the work of W. W. Bledsoe, most notably the work reported in [Ble71] and [BBH72].

The first version of the Boyer-Moore system was released in 1973 and we have continued to improve the system and periodically release new versions ever since. Such improvements may involve changes to the logic in which theorems are proved, changes to the heuristics or proof techniques employed, and changes to the user interface. Nevertheless, the 1973 release would be recognized by today's users as a primitive version of the system. By 1980 the system, then called "Thm," was quite similar to the present version. By 1986 the system had also become known as *Nqthm* (pronounced *en-queue-thum*), an acronym for "New, Quantified THeoreM Prover," the little-considered name of the disk directory on which a rapidly evolving improvement resided that included a limited quantification capability.

The 1978 book, *A Computational Logic*[ACL] is still a largely accurate and comprehensive description of how the theorem prover works. That version of the system has been reimplemented by at least three groups working from[ACL]. The book *A Computational Logic Handbook*[ACLH] (often called the "Nqthm handbook") contains a precise description of the Nqthm logic as it stood in 1988. In addition, [ACLH] indicates by appropriate bibliographic citations how to reconstruct the 1988 version of Nqthm from [ACL] and numerous journal articles about subsequent improvements. However, the handbook is primarily devoted to a description of how to use the Nqthm logic and its mechanization.

We expect to release a new version of Nqthm in 1993. When it is necessary to distinguish this new release from previous versions of Nqthm, we call the new version *Nqthm-1992*. Nqthm-1992 differs from the older release primarily by supporting the introduction of undefined but constrained function symbols and the use of a derived rule of inference permitting the instantiation of function symbols, giving Nqthm-1992 a "higher-order" feel [BGKM]. Some performance improvements were made, a few new user commands were added, and many minor bugs were fixed. None of the bugs affected soundness. Since 1989 we have been working on a new theorem prover, called *Acl2*, designed from the ground up and heavily influenced by our experience with Nqthm. This paper is about Nqthm-1992, not Acl2.

The system *Pc-Nqthm* ("Proof-Checker" Nqthm) is an interactive enhancement of Nqthm. *Pc-Nqthm-1992* is the version of the enhancement for Nqthm-1992. The Pc-Nqthm user can give commands at a low level (such as deleting a hypothesis, diving to a subterm of the current term, expanding a function call, or applying a rewrite rule) or at a high level (such as invoking the Boyer-Moore Theorem Prover). Commands also exist for displaying useful information and for controlling the progress of the proof, and for helping the user create compound commands.

We say more about these two systems in Sections 2 and 3, respectively. Section 4 provides a detailed illustration of their use. We conclude in Section 5 with a long list of applications of these systems, each accompanied by a very brief description.

Acknowledgements

We thank Andrew Ireland of the MRG, Department of Artificial Intelligence, University of Edinburgh, for useful comments on a draft of this paper.

1 Overview of the Logic

The Nqthm logic is a first order, quantifier free logic of recursive functions.

The logic includes axioms defining many useful primitive functions. Most fundamental, perhaps, are the axioms introducing the functional analogues of the propositional calculus connectives and the equality relation. The axioms describe two distinct objects, called `T` and `F` which play the roles of truth values in our propositional functions. An "if-then-else" function is axiomatized to return its second or third argument depending on whether its first argument differs from `F`. Using `IF` such familiar propositional functions as `AND`, `OR`, `NOT` and `IMPLIES` are defined. The logic also includes the function `EQUAL` which returns `T` or `F` according to whether its two arguments are equal. With these functions we have essentially embedded propositional calculus and equality into the term structure of the logic. Among the other primitives defined are those for the construction and elementary manipulation of natural numbers, integers, ordered pairs, and symbols.

The logic includes an induction principle based on the well-foundedness of the "less than" relation on the ordinals up to ε_0 and extension principles allowing the user to introduce recursively defined functions, axiomatically constrained functions, and new inductively defined data types. Successful use of the extension principles require the proving of certain theorems that guarantee the conservation of the consistency of the logic.

A precise description of the Nqthm logic may be found in Chapter 4 of the Nqthm handbook [ACLH].

The syntax of our logic resembles that of the Lisp programming language. For example, our definition of the Peano addition function is

DEFINITION.
```
(PLUS I J)
   =
(IF (ZEROP I)
    (FIX J)
    (ADD1 (PLUS (SUB1 I) J))).
```

Roughly speaking, this may be read "If `I` is 0, the sum of `I` and `J` is the natural number `J`; otherwise it is one more than the sum of `I-1` and `J`." Observe how the embedding of propositional calculus into the term structure of the logic allows us to define recursive functions in a computational style. The definitional principle requires the proof of a theorem exhibiting an ordinal measure of the arguments that, in the recursive call, decreases according to the ordinal less-than relation. A suitable measure of the arguments in this case is simply the natural number `I` itself. Put another way, every recursive definition must be proved to terminate. This (along with some syntactic restrictions) guarantees the existence of a unique function satisfying the definitional equation. It also means that such functions can be evaluated to concrete results when concrete input is provided.

An example theorem is

THEOREM.
```
(EQUAL (PLUS (PLUS I J) K)
       (PLUS I (PLUS J K)))
```

which states the associativity of Peano addition. Observe that the logic is quantifier free. There are no quantifiers in the language and all theorems are implicitly universally quantified on the far outside. Thus, one may think of the above theorem as saying, "for all `I`, `J`, and `K`, `(PLUS (PLUS I J) K)` is `(PLUS I (PLUS J K))`."

The logic is untyped. While it is a theorem that `(PLUS 2 2)` equals `4`, it is also a theorem that `(PLUS T 2)` equals `2`. The proof of the latter theorem is based on the fact that our `PLUS` function is defined to coerce to `0` any argument that is not a natural number. In particular, recalling the definition of `PLUS` above, `(ZEROP I)` is `T` if either `I` is the natural number 0 or is not a natural number, and `(FIX J)` is `J` if `J` is a natural number and 0 otherwise. In general, all Nqthm primitive functions are axiomatized to coerce "unexpected" arguments to selected values in the "intended domain" so that all functions are total and behave in a predictable way outside of the intended domain. User defined functions generally inherit this predictable behavior. The result is that many theorems are more simply stated than would be the case otherwise. While many outsiders disparage this aspect of Nqthm, most Nqthm users recognize that by eliminating restrictive hypotheses and potentially explosive case analysis it makes a valuable contribution to the ease with which theorems can be proved and subsequently used.

While the logic is technically first order, the combination of certain of the extension principles and a derived rule of inference make it feel higher order. It is possible to introduce a new function symbol by *constraint* so that the symbol is supposed to satisfy a given formula. To be admissible, such an extension must exhibit one function (a *witness*) satisfying the constraint. Theorems may then be proved about the constrained function. Later, a derived rule of inference permits those theorems to be *functionally instantiated*, that is, the constrained function symbols may be replaced by other function symbols, provided it can be proved that the incoming symbols satisfy the constraints on the replaced ones. See [BGKM].

2 Introduction to the Nqthm System

The Nqthm system is a Common Lisp[Ste84] program of roughly one million characters. It is currently available by ftp from Internet host `ftp.cli.com`. There is no fee, but, at the insistence of our sponsors, a license agreement must be signed. The currently released version of Nqthm was first released in July of 1988, and no bugs affecting soundness have been reported as of the time of this writing. The chapter of the Nqthm handbook [ACLH] on installation describes in complete detail how to bring up Nqthm from the sources. It is our intention to release a new version of Nqthm, called Nqthm-1992, in 1993. Included with that release will be a description of the formal logic, the user's reference guide, installation instructions, and almost 12 megabytes of input files created by many users, as well as a license agreement that we does not require a signature, we expect.

2.1 Commands and the Data Base

When Nqthm is started up, the user is confronted with a standard Common Lisp interactive loop. Represented within that Common Lisp, mainly via "property lists" associating data with function symbols, is the initial Nqthm logic. By executing commands the user can extend the logic, for example by invoking the definitional principle command, `DEFN`. The data base also contains all the theorems proved thus far in that Nqthm session and new ones can be added by successful invocations of the `PROVE-LEMMA` command. We illustrate commands later. Commands which extend the data base are called "event commands." Still other commands allow the user to inspect the data base, for example to display the definition of a previously defined function or the statement of a previously proved theorem. A third class of commands remove items from the data base. For example, the `UBT` command, which stands for "undo back through," rolls the data base back to a previous extension. Finally, there are commands for saving the data base to a file and reinstating that data base so work can be saved from day to day or moved from one machine or user to another.

An example event command is

```
(DEFN PLUS (I J)
 (IF (ZEROP I)
```

```
(FIX J)
(ADD1 (PLUS (SUB1 I) J)))).
```

This command submits the above-mentioned definition of Peano addition to the definitional principle. If the DEFN completes without error, it extends the data base so as to contain the new definition. In fact, if a user tried to submit the above DEFN command an error would result and a message would be printed informing the user that the function symbol PLUS is already defined and hence may not be redefined.

As noted above, the Nqthm user actually types commands to Common Lisp. That is, Nqthm commands are just Common Lisp programs the user invokes, using the available Common Lisp interaction protocols. We personally most often use Nqthm from within a Gnu Emacs text editor[Sta87] "shell buffer" running Common Lisp. We thus have available both the text processing convenience of Gnu Emacs and the computational power of Common Lisp to help us review what is happening in an Nqthm session and to create and record commands. We personally find this an appealing aspect of the Nqthm interface: a powerful programming language and decades of design and evolution make the Lisp interactive environment very convenient and efficient for the experienced user. But again, this is a point of contention. Critics of Nqthm assert, with some justification, that it has no user interface. Our choice of interface illustrates a fundamental philosophical position: we do not want to obstruct the experienced user from getting theorems proved simply to provide the novice with a restrictive but "user-friendly" intermediary. The user is presumed interested in proving theorems and must come to the table prepared.

2.2 The Theorem Prover

The theorem prover is a symbol manipulation program that attempts to prove the submitted formula by applying the logic's rules of inference. The behavior of the theorem prover is largely determined by the data base and hundreds of heuristics for controlling the use of the rules of inference, axioms, definitions, and previously proved theorems. The theorem prover has seven main proof techniques.

- *Simplification* coordinates the application of rewrite rules derived from axioms, definitions, and previously proved theorems with decision procedures for propositional calculus, equality, and linear arithmetic. We say more about rewriting below.

- *Destructor elimination* replaces variables by terms to explicate implied structures and thus eliminate some function applications. For example, if the term denoting "`I mod J`" occurs in the conjecture, and suitable theorems are in the data base, the system will replace `I` by `X+J*Y`, where `X<J`. After this rerepresentation of `I`, the term `I mod J` can be replaced by `X`, eliminating the use of the mod function.

- *Cross-fertilization* is a heuristic for using equality hypotheses that is especially effective at using inductive hypotheses.

- *Generalization* replaces terms by new variables, possibly of restricted type, so as to generalize the conjecture being proved. This is often necessary in inductive proofs, though Nqthm's generalization heuristic is weak and little used.

- *Elimination of irrelevance* attempts to discard unnecessary hypotheses from a conjecture and is another form of generalization.

- *Induction* analyzes the uses of recursively defined functions in the conjecture and attempts to find a suitable application of the induction principle. This task is made easier by the duality of recursion and induction; the theorems proved at definition time, establishing that the recursion is well-founded, are easily converted into inductive schemas. Heuristics are used to discard inappropriate induction "suggestions" and to combine "compatible" suggestions. It should be noted that the absence of quantifiers also contributes to the success of the induction heuristic. Often, merely to state the desired theorems, users have to define explicitly some recursive function that strongly suggests the induction for the proof of the conjecture. We should note that in contrast to our approach, there has also been promising research on the mechanization of induction in the presence of quantifiers; see for example [BSvHIS].

When theorems are added to the data base, the user must specify how they are to be used later. The various proof techniques query different parts of the data base for relevant theorems. The most common way to use a previously proved theorem is as a "rewrite rule." The command below invokes the theorem prover on the associativity of addition. It specifies that, if proved, the theorem is to be stored for future use as a rewrite rule and referred to by the name ASSOC-OF-PLUS.

```
(PROVE-LEMMA ASSOC-OF-PLUS (REWRITE)
  (EQUAL (PLUS (PLUS I J) K)
         (PLUS I (PLUS J K))))
```

The data base is not extended if the system fails to prove the formula. Thus, while the user controls the behavior of the system by suggesting theorems and their use as rules, only valid rules are entered into the data base. If the system proves ASSOC-OF-PLUS as above, then every time the system subsequently encounters an instance of the left-hand side above, e.g., (PLUS (PLUS X 3) (SQ Z)), it replaces it by the corresponding instance of the right-hand side, e.g., (PLUS X (PLUS 3 (SQ Z))). Thus, by proving the associativity of addition as stated above, the user causes the system to right-associate addition expressions. By swapping the two sides of the equality above, the user would cause addition expressions to be left-associated. If both rules were proved, an "infinite" loop would result. Heuristics are present to prevent some such loops (but not gross ones like this because users are presumed to be more careful).

Rewrite rules need not be simple equalities. For example, the following statement of the uniqueness of prime factorizations,

THEOREM.
```
(IMPLIES (AND (PRIMES L1)
              (PRIMES L2)
              (EQUAL (PROD L1) (PROD L2)))
         (PERM L1 L2))
```

when used as a rewrite rule, causes instances of (PERM L1 L2) to be replaced by T provided the system can prove that (the instances of) L1 and L2 are lists of primes whose products are equal. Such "backchaining" invites infinite regress and heuristics are present to limit such backchaining. As noted previously, details are given in [ACL] and [ACLH] and the papers cited in the latter.

2.3 Hard Theorems Proved and the Importance of the User

Among the hard theorems proved by Nqthm are

- the existence and uniqueness of prime factorizations [ACL]
- the invertibility of the RSA public key encryption algorithm [BM84c]
- Wilson's theorem [Rus85],
- Gauss's law of quadratic reciprocity [Rus92],
- the tautology theorem (that every propositional tautology has a proof in Shoenfield's propositional logic) [Sha85],
- Gödel's incompleteness theorem (for Shoenfield's first order logic extended with Cohen's axioms for hereditarily finite set theory, Z2) [Sha86], and
- the correctness of many algorithms, computer programs, and digital hardware designs including simple compilers, operating systems, the Berkeley C string library as compiled by the gcc compiler for the Motorola MC68020, and a fabricated microprocessor.

Later in the paper we enumerate the current set of example files to be distributed with Nqthm-1992 in which we include all those examples mentioned above with appropriate bibliographic citations and acknowledgment of the authors.

This brings us to a crucial aspect of an informal understanding of Nqthm. In one sense, Nqthm is an automatic theorem prover: once it is set loose on a conjecture the user cannot influence its behavior. Because of this, and the fact that Nqthm is quite capable of automatically finding proofs of many elementary theorems—even theorems requiring induction and the discovery of additional inductively proved lemmas—it is easy to fall into the trap of thinking that Nqthm proved Gödel's theorem automatically. But in truth, Nqthm would flounder if simply presented with Gödel's theorem: its success at "automatically" discovering a proof of that theorem was entirely due to the care with which the user had used Nqthm to construct an appropriate data base. Thus, in a practical sense, Nqthm is a proof checker: it is essentially led to the proofs of hard theorems by the user, who "trains" it by formulating an appropriate sequence of intermediate results each of which is within its competence at the time it encounters it. The proofs of hard theorems constructed by Nqthm are, in every case, actually the intellectual work of the user. Nqthm's contributions to the proofs are not those of discovery and creativity but of care and plodding precision.

It is hard, perhaps impossible, to use Nqthm effectively without investing a substantial amount of time learning how to use it. Almost all of the successful users of Nqthm have taken a course at the University of Texas at Austin on proving theorems in the Nqthm logic. The Nqthm handbook [ACLH] was written primarily to teach users how to use Nqthm's logic and the Nqthm system; the handbook describes in great detail the Nqthm commands and how to use them to control the system's behavior. It also discusses successful styles of Nqthm use. Prospective users of Nqthm might also find the description of its heuristics in [ACL] to be of use, even though the description there is very low level. Finally, an online "users group" mailing list exists and is often helpful, especially to new users who frequently broadcast pleas for help and almost as often find some experienced user willing to explain some facet of the system.

3 Introduction to the Pc-Nqthm System

Pc-Nqthm, written by Matt Kaufmann, is essentially an extension of Nqthm. In particular, it allows all the interaction that Nqthm does. But it also provides for lower-level interaction through an interactive loop. It has been in existence since approximately 1987, and like Nqthm, is currently available by ftp from Internet host `ftp.cli.com`.

Pc-Nqthm also provides full first-order quantification through a technique generally called *Skolemization*. This aspect of Pc-Nqthm has been thoroughly documented in [Kau92b], and we'll say no more about it here, focusing instead on the system's interactive features.

As with a variety of proof-checking systems, Pc-Nqthm is goal-directed in the following sense. One enters the system by presenting it with a theorem to be proved. As one proceeds, one typically simplifies and proves goals, but generates additional goals in the process. The proof is complete when the original goal, as well as all subgoals generated during the proof, have been proved. Upon completion of an interactive proof, the lemma with its proof may be stored as a `PROVE-LEMMA` event that can be added to the user's current database of definitions and lemmas. This event can later be replayed in "batch mode," i.e., without user interaction. Partial proofs can also be stored.

Some features provided by the interactive loop are as follows.

- a two-layer help facility
- separation from the Nqthm (Lisp) command level, to avoid confusion of environments
- the ability to focus on a subterm, where the context is used for simplification, equality substitution, function expansion, and especially manually-invoked rewriting, where unproved hypotheses from the rewrite rule are used to create new subgoals
- the ability to call on the full theorem prover to simplify or prove the current goal
- low-level commands at the goal level, which provide the ability to drop and (with a proof obligation) add hypotheses, to do case splitting, to generalize subterms, to change goals, or to start a proof by induction

- ability to choose the order in one which works on goals
- a capability for enabling and disabling events and sets of events
- abbreviation and comment mechanisms
- commands for undoing and *restoring* (undoing the undoing)
- support for saving multiple proof contexts
- support for instantiation of variables that it is sound to instantiate
- numerous commands for displaying relevant information, such as the current term
- a tactic-like feature called *macro commands*, allowing user-extensibility in a sound way

Technically, one can view Pc-Nqthm as an extension of Nqthm that allows a single new hint called `INSTRUCTIONS`. Consider for example the following event.

```
(PROVE-LEMMA ASSOC-OF-PLUS (REWRITE)
  (EQUAL (PLUS (PLUS I J) K)
         (PLUS I (PLUS J K)))
  ((INSTRUCTIONS INDUCT PROVE PROVE)))
```

The hint says: "Use `INDUCT`ion to replace the main goal by subgoals (the base and inductive steps), then use the full `PROVE`r capability of Nqthm to complete one subgoal, then similarly `PROVE` the other subgoal." Now actually, users rarely type in such hints. Instead, Pc-Nqthm provides an interactive loop for completing the proof, and upon completion, the event form displayed above is printed by the system so that the user may insert it in the events file.

In practice, many Nqthm users employ the interactive capability of Pc-Nqthm to discover additional useful lemmas to be proved. The example provided in the next section will serve to illustrate this kind of approach, as well as other basics of Pc-Nqthm use. Upon completion of an interactive proof, it is often the case that one has proved enough additional rewrite rules at the top (Nqthm) level that a proof may now succeed without interaction.

Thus, one often finds no `INSTRUCTIONS` hints in the final events file, which therefore can be run through Nqthm (without Pc-Nqthm).

In fact, it is useful to attempt to avoid `INSTRUCTIONS` hints in the final events file, because this approach encourages the user to think at a reasonably high level. A danger of Pc-Nqthm is that it makes interaction so easy that people sometimes do proofs at too low a level, which makes the proof scripts difficult to modify when one changes the supporting definitions. The low-level approach is also undesirable because it doesn't encourage the discovery of valuable rewrite rules that can be useful in subsequent proofs.

A user's manual [Kau88] provides other examples and more detail about the logical foundations of the system's backward-directed proof method. It also covers advanced topics such as the writing of macro commands. Little of substance has changed in the system since that manual was written; an update [Kau89a] describes the changes, notably variable instantiation, since the original manual.

4 An Example

In this section we show a proof of correctness of a simple sorting function using Pc-Nqthm, and hence Nqthm (since Pc-Nqthm is built on top of Nqthm). In fact, we use (Pc-)Nqthm-1992 below. An informal specification for a sorting function is that the result list should be sorted, and should be a permutation of the input list. We'll use a standard "mergesort" algorithm: sort a list by splitting it into two sublists, sorting each of those, and then merging the results together.

We will display user input in lower-case, always preceded by a prompt '`>`'. Comments by us not intended for Pc-Nqthm are set in italics. For example, we start by initializing the prover.

`>(boot-strap nqthm)`   *initialize the data base*

The 'global variable' `CHRONOLOGY` *tells us where we are. For now, we are in a state where the "boot-strap" has been done to create the "ground-zero" theory, but that's all.*

```
>chronology
(GROUND-ZERO)
```

Let us begin with a trivial exercise, just to get warmed up: define a function that adds 3 to its input.

```
>(defn plus3 (x)
   (plus x 3))
```

The system responds quickly as follows.

```
     Observe that (NUMBERP (PLUS3 X)) is a theorem.
```

If we take another look at the `CHRONOLOGY`*, we'll see that the definition event* `PLUS3` *has been pushed onto the list.*

```
>chronology
(PLUS3 GROUND-ZERO)
```

Let's undo (Undo Back Through) the last event; after all, it was just a warm-up. Serious users of Nqthm often undo in this manner many times during a proof exercise; in particular, it's easy to make mistakes in complex definitions.

```
>(ubt)
```

And the chronology is updated accordingly, as the system shows it to be `(GROUND-ZERO)` *once again.*

Let us try a recursive definition now, one which we'll find useful later. We also use this opportunity to introduce list processing. The function `CONS` *of two arguments constructs an ordered pair. The function* `CAR` *returns the first component of such a pair. Thus, an axiom is that* `(CAR (CONS X Y))` *is* `X`*. The function* `CDR` *returns the second component, i.e.,* `(CDR (CONS X Y))` *is* `Y`*. The function* `LISTP` *recognizes ordered pairs constructed by* `CONS`*, i.e.,* `LISTP` *returns* `T` *or* `F` *according to whether its argument is an ordered pair constructed by* `CONS`*. Ordered pairs are printed as lists. For example,* `(CONS 2 (CONS 4 (CONS 6 NIL)))` *is printed as* `'(2 4 6)`*. The symbol* `NIL` *is a constant (of type* `LITATOM`*) not constructed by* `CONS` *and is conventionally used to terminate lists.*

Now let us define the function that computes the length of a list.

```
>(defn length (x)
  (if (listp x)
      (add1 (length (cdr x)))
    0))
```

This definition may be read as follows: The `LENGTH` *of an object constructed by* `CONS` *is one plus the* `LENGTH` *of its* `CDR`*; the* `LENGTH` *of all other objects is* `0`*.*

The system responds to this `DEFN` *command as follows, explaining that the logic's definitional principle has been satisfied using the well-founded relation* `LESSP` *(i.e., $<$).*

```
     Linear arithmetic and the lemma CDR-LESSP inform us
that the measure (COUNT X) decreases according to the
well-founded relation LESSP in each recursive call.  Hence,
LENGTH is accepted under the principle of definition.  From
the definition we can conclude that (NUMBERP (LENGTH X)) is
a theorem.
```

Before we go on, let us "execute" some forms. The Nqthm system provides a capability for evaluating terms that do not contain variables. This "reduce loop" is valuable for debugging specification functions. Note that the prompt *is '*' and all input in the following display follows that prompt, except for the invocation of* `R-LOOP` *at the outset; everything else is printed by the system.*

```
>(r-loop)

Trace Mode: Off   Abbreviated Output Mode:  On
Type ? for help.
*(plus 2 3)
 5
*(cons 2 (cons 4 (cons 7 nil)))
 '(2 4 7)
*(length '(2 4 7))
 3
*ok
Exiting R-LOOP.
```

Let us return to the problem of writing a sorting function and proving it correct. The main subfunction for our `MERGESORT` *function is a* `MERGE` *function that "zippers" together two ordered lists so that the result is ordered. Recall that the function* `CAR` *returns the first element of a list and the function* `CDR` *returns what is left of the list after removing the very first element.*

```
>(defn merge (l m)
  (if (not (listp l))
      m
    (if (not (listp m))
        l
      (if (lessp (car l) (car m))
          (cons (car l) (merge (cdr l) m))
        (cons (car m) (merge l (cdr m)))))))
```

The system responds as follows.

```
ERROR:  The admissibility of this definition has not been
established.  The theorem prover's heuristics found no
plausible measure to justify the recursion.  In particular,
no single argument of the function is both tested in each
branch and changed in each recursive call.  The definition
is rejected.
```

Evidently the definition failed. Intuitively, the theorem prover could not verify that the arguments to MERGE *get "smaller" in each recursive call. Again, there is actually a formal "principle of definition" in the logic that applies here.*

However, we can see that in fact, the sum of the lengths of the two arguments to MERGE *is smaller in each recursive call than it is at the outset. That is, if* (LISTP L) *and* (LISTP M) *hold, then the sum of the lengths of* (CDR L) *and* M *is actually one less than the sum of the lengths of* L *and* M*; similarly for* L *and* (CDR M)*. The hint* (lessp (plus (length l) (length m))) *is intended to express such a claim, namely: for each recursive call* (MERGE x y)*, the sum of the lengths of* x *and* y *is* LESSP *the sum of the lengths of* L *and* M*, under the hypotheses that are known to hold from the* IF *structure of the body.*

```
>(defn merge (l m)
  (if (not (listp l))
      m
    (if (not (listp m))
        l
      (if (lessp (car l) (car m))
          (cons (car l) (merge (cdr l) m))
        (cons (car m) (merge l (cdr m))))))
  ((lessp (plus (length l) (length m)))))    ; hints
```

This time the system's response is more friendly.

```
     Linear arithmetic, the lemma SUB1-ADD1, and the
definitions of LESSP, PLUS, and LENGTH inform us that the
measure (PLUS (LENGTH L) (LENGTH M)) decreases according to
the well-founded relation LESSP in each recursive call.
Hence, MERGE is accepted under the principle of definition.
Note that (OR (LISTP (MERGE L M)) (EQUAL (MERGE L M) M)) is
a theorem.
```

Another function we need is one that splits a list into two sublists. That way, we can sort a list by sorting each "half" and then MERGE*-ing them together.*

```
>(defn odds (l)
  (if (not (listp l))
```

```
      nil
    (cons (car l) (odds (cddr l)))))
```

The system accepts the above definition without difficulty. Now let us attempt to define MERGESORT.

```
>(defn mergesort (l)
  (if (not (listp l))
      nil
    (if (not (listp (cdr l)))
        l
      (merge (mergesort (odds (cdr l)))
             (mergesort (odds l))))))
```

Unfortunately, the system cannot accept this definition.

```
ERROR:  The admissibility of this definition has not been
established.  The simplifier could not prove that the
measure(s) tried decrease in each recursive call.  The
definition is rejected.  Below are listed the relations and
measures tried and some of the unproved goals for each.

Relation:  LESSP
Measure:   (COUNT L)
Unproved goals:
      (IMPLIES (AND (LISTP L) (LISTP (CDR L)))
               (LESSP (COUNT (ODDS L)) (COUNT L)))
```

Let us try again using the function LENGTH *to measure* L *in place of the built-in function* COUNT*, since although we've said nothing about* COUNT*, we can see why if* L *and its tail* (CDR L) *are both non-emtpy, then a list formed from every other element of* L *has a smaller length than* L.

```
>(defn mergesort (l)
  (if (not (listp l))
      nil
    (if (not (listp (cdr l)))
        l
      (merge (mergesort (odds (cdr l)))
             (mergesort (odds l)))))
  ((lessp (length l))))    ; hint, saying that  (length x) <  (length l)
                           ; for each recursive call  (mergesort x)
```

Unfortunately, even with the hint to use LENGTH*, the admissibility proof fails.*

```
ERROR:  The admissibility of this definition has not been
established.  The simplifier could not prove that the
measure(s) tried decrease in each recursive call.  The
definition is rejected.  Below are listed the relations and
measures tried and the unproved goals for each.

Relation:  LESSP
Measure:   (LENGTH L)
Unproved goals:
      (IMPLIES (AND (LISTP L)
                    (LISTP (CDR L))
                    (NOT (EQUAL (LENGTH (ODDS L)) 0)))
               (LESSP (SUB1 (LENGTH (ODDS L)))
                      (LENGTH (CDR L))))
```

The definition failed to be accepted above, but we'll take a clue from what it says about "unproved goals" and see if we can prove a useful lemma. The hypothesis (NOT (EQUAL (LENGTH (ODDS L)) 0)) *there appears to be redundant, so we'll omit it. It may seem odd that we ask the theorem prover to prove something that it (essentially) just failed to prove. However, when the prover is trying to accept a definition it is trying for speed rather than completeness, so it does not use all of its power, but only uses simplification. In particular, it does not try proof by induction, which turns out to be crucial for the proof.*

```
>(prove-lemma mergesort-helper (rewrite)
  (implies (and (listp l)
                (listp (cdr l)))
           (equal (lessp (sub1 (length (odds l)))
                         (length (cdr l)))
                  t)))
```

The theorem prover proves this lemma successfully, so it seems quite reasonable to try to define MERGESORT *once again.*

```
>(defn mergesort (l)
  (if (not (listp l))
      nil
```

```
    (if (not (listp (cdr l)))
        l
      (merge (mergesort (odds (cdr l)))
             (mergesort (odds l)))))
  ((lessp (length l))))
```

Unfortunately, the admissibility proof still fails, this time with another goal, as shown.

```
ERROR:  The admissibility of this definition has not been
established.  The simplifier could not prove that the
measure(s) tried decrease in each recursive call.  The
definition is rejected.  Below are listed the relations and
measures tried and the unproved goals for each.

Relation:  LESSP
Measure:   (LENGTH L)
Unproved goals:
      (IMPLIES (AND (LISTP L)
                    (LISTP (CDR L))
                    (NOT (EQUAL (LENGTH (ODDS (CDR L))) 0)))
               (LESSP (SUB1 (LENGTH (ODDS (CDR L))))
                      (LENGTH (CDR L))))
```

So, let us prove the goal above as well. In fact, we'll roll back the previous lemma (using the (UBT) *feature described above), and prove the two "unproved goals" together.*

```
>(prove-lemma mergesort-helper (rewrite)
  (implies (and (listp l)
                (listp (cdr l)))
           (and (equal (lessp (sub1 (length (odds l)))
                              (length (cdr l)))
                       t)
                (equal (lessp (sub1 (length (odds (cdr l))))
                              (length (cdr l)))
                       t))))
```

The proof of this lemma also succeeds. And now, so does the admission of MERGESORT. *And, the function seems to behave properly on at least one example.*

```
>(r-loop)

Trace Mode: Off   Abbreviated Output Mode:  On
Type ? for help.
*(mergesort '(3 7 8 2 9 4 7))
 '(2 3 4 7 7 8 9)
*
```

One of the two standard properties to prove about a sorting function is that it returns a sorted list. So, let's define a predicate that says whether a list is sorted. Since there are no quantifiers in the Boyer-Moore logic (actually that's not entirely true, but what is there doesn't have nearly the mechanical support that recursion does), we use recursion to define this predicate (i.e., this Boolean-valued function).

```
>(defn sortedp (x)
  (if (listp x)
      (if (listp (cdr x))
          (and (not (lessp (car (cdr x)) (car x)))
               (sortedp (cdr x)))
        t)
    t))
```

Now we are ready for the theorem we have been wanting to prove about our sorting function. Notice in the proof transcript shown below that in fact the prover uses a generalization heuristic in order to generate the interesting, useful subgoal that says that the MERGE *of two sorted lists is sorted:*

```
(IMPLIES (AND (SORTEDP B) (SORTEDP U))
              (SORTEDP (MERGE U B)))
```

And now for the theorem...

```
>(prove-lemma sortedp-mergesort (rewrite)
  (sortedp (mergesort x)))


     Call the conjecture *1.
```

Let us appeal to the induction principle. There is only one suggested induction. We will induct according to the following scheme:

```
      (AND (IMPLIES (NOT (LISTP X)) (p X))
           (IMPLIES (AND (LISTP X) (NOT (LISTP (CDR X))))
                    (p X))
           (IMPLIES (AND (LISTP X)
                         (LISTP (CDR X))
                         (p (ODDS X))
                         (p (ODDS (CDR X))))
                    (p X))).
```

The lemmas SUB1-ADD1 and MERGESORT-HELPER and the definitions of LESSP and LENGTH inform us that the measure (LENGTH X) decreases according to the well-founded relation LESSP in each induction step of the scheme. The above induction scheme leads to three new goals:

```
Case 3. (IMPLIES (NOT (LISTP X))
                 (SORTEDP (MERGESORT X))),

  which simplifies, expanding the definitions of MERGESORT
  and SORTEDP, to:

        T.

Case 2. (IMPLIES (AND (LISTP X) (NOT (LISTP (CDR X))))
                 (SORTEDP (MERGESORT X))),

  which simplifies, expanding the definitions of MERGESORT
  and SORTEDP, to:

        T.

Case 1. (IMPLIES (AND (LISTP X)
                      (LISTP (CDR X))
                      (SORTEDP (MERGESORT (ODDS X)))
                      (SORTEDP (MERGESORT (ODDS (CDR X)))))
                 (SORTEDP (MERGESORT X))),
```

which simplifies, unfolding the definition of MERGESORT, to:

```
(IMPLIES (AND (LISTP X)
              (LISTP (CDR X))
              (SORTEDP (MERGESORT (ODDS X)))
              (SORTEDP (MERGESORT (ODDS (CDR X)))))
         (SORTEDP (MERGE (MERGESORT (ODDS (CDR X)))
                         (MERGESORT (ODDS X))))).
```

Appealing to the lemma CAR-CDR-ELIM, we now replace X by (CONS V Z) to eliminate (CDR X) and (CAR X). This generates the conjecture:

```
(IMPLIES
   (AND (LISTP Z)
        (SORTEDP (MERGESORT (ODDS (CONS V Z))))
        (SORTEDP (MERGESORT (ODDS Z))))
   (SORTEDP (MERGE (MERGESORT (ODDS Z))
                   (MERGESORT (ODDS (CONS V Z)))))).
```

We will try to prove the above formula by generalizing it, replacing (ODDS Z) by Y and (ODDS (CONS V Z)) by A. The result is the formula:

```
(IMPLIES
     (AND (LISTP Z)
          (SORTEDP (MERGESORT A))
          (SORTEDP (MERGESORT Y)))
     (SORTEDP (MERGE (MERGESORT Y) (MERGESORT A)))).
```

We will try to prove the above formula by generalizing it, replacing (MERGESORT Y) by U and (MERGESORT A) by B. We must thus prove:

```
(IMPLIES (AND (LISTP Z)
              (SORTEDP B)
              (SORTEDP U))
         (SORTEDP (MERGE U B))).
```

```
  Eliminate the irrelevant term.  We would thus like to
  prove:

        (IMPLIES (AND (SORTEDP B) (SORTEDP U))
                 (SORTEDP (MERGE U B))),

  which we will finally name *1.1.
```

```
     We will appeal to induction.  Three inductions are
suggested by terms in the conjecture.  However, they merge
into one likely candidate induction.  We will induct
according to the following scheme:
      (AND (IMPLIES (NOT (LISTP U)) (p U B))
           (IMPLIES (AND (LISTP U) (NOT (LISTP B)))
                    (p U B))
           (IMPLIES (AND (LISTP U)
                         (LISTP B)
                         (LESSP (CAR U) (CAR B))
                         (p (CDR U) B))
                    (p U B))
           (IMPLIES (AND (LISTP U)
                         (LISTP B)
                         (NOT (LESSP (CAR U) (CAR B)))
                         (p U (CDR B)))
                    (p U B))).
Linear arithmetic, the lemma SUB1-ADD1, and the definitions
of LESSP, PLUS, and LENGTH establish that the measure:
      (PLUS (LENGTH U) (LENGTH B))
decreases according to the well-founded relation LESSP in
each induction step of the scheme.  The above induction
scheme leads to the following six new conjectures:

Case 6. (IMPLIES (AND (NOT (LISTP U))
                      (SORTEDP B)
                      (SORTEDP U))
                 (SORTEDP (MERGE U B))).
```

```
  This simplifies, opening up MERGE and SORTEDP, to the
  following two new conjectures:

  Case 6.2.
```

The rest of this "proof output" isn't that interesting, so we have edited it out.

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

Now we work towards the second of two properties to prove about a sorting function, namely, that the result is a permutation of the input. It is convenient to specify this property by saying that every object occurs the same number of times in each list. One can prove equivalence of various notions of permutation, but we won't do that here. So, we start by writing a function that counts the number of occurrences of an object in a list.

```
>(defn occurrences (a x)
  (if (listp x)
      (if (equal a (car x))
          (add1 (occurrences a (cdr x)))
        (occurrences a (cdr x)))
    0))

     Linear arithmetic and the lemma CDR-LESSP establish
that the measure (COUNT X) decreases according to the
well-founded relation LESSP in each recursive call.  Hence,
OCCURRENCES is accepted under the principle of definition.
From the definition we can conclude that:
      (NUMBERP (OCCURRENCES A X))
is a theorem.
```

A proof of the second theorem can now be attempted. Unfortunately, it seems to fail, so we interrupt the proof.

```
>(prove-lemma occurrences-mergesort (rewrite)
  (equal (occurrences a (mergesort x))
         (occurrences a x)))
```

```
    Name the conjecture *1.
```

... most output omitted here ...

```
      (IMPLIES
       (AND
        (LISTP Z)
        (EQUAL
             (OCCURRENCES A
                          (MERGESORT (ODDS (CONS V Z))))
             (OCCURRENCES A (ODDS (CONS V Z))))
        (EQUAL (OCCURRENCES A (MERGESORT (ODDS Z)))
               (OCCURRENCES A (ODDS Z)))
        (NOT (EQUAL A V)))
       (EQUAL
        (OCCURRENCES A
                  (MERGE (MERGESORT (ODDS Z))
                         (MERGESORT (ODDS (CONS V Z)))))
        (OCCURRENCES A Z))).

   We will try to prove the above formula by generalizing
   it, replacing (ODDS Z) by Y and (ODDS (CONS V Z)) by U.
   We must thus prove the formula:

      (IMPLIES
       (AND (LISTP Z)
            (EQUAL (OCCURRENCES A (MERGESORT U))
                   (OCCURRENCES A U))
            (EQUAL (OCCURRENCES A (MERGESORT Y))
                   (OCCURRENCES A Y))
            (NOT (EQUAL A V)))
       (EQUAL
        (OCCURRENCES A
                    (MERGE (MERGESORT Y) (MERGESORT U)))
        (OCCURRENCES A Z)))
```

We may as well abort the proof at this point, since the prover has made a false generalization. The relation between `Z` *and* `(ODDS Z)` *(among others)* *has been lost. Note that soundness is not imperiled by such a generalization: the theorem prover merely adopts a goal that is impossible to achieve.*

One can actually look at the output above and guess a useful rewrite rule to prove as a lemma. Below, we'll demonstrate how to use the interactive capability of Pc-Nqthm in order to accomplish the same thing. We'll annotate an interactive session with comments. All input follows the prompt, "`->:` *"; the rest is output, except for the comments, which are all in italics.*

```
>(verify (equal (occurrences a (mergesort x))
                (occurrences a x)))
```

Now we're in the interactive "proof-checker" loop.

`->: p` *print the current term*

```
(EQUAL (OCCURRENCES A (MERGESORT X))
       (OCCURRENCES A X))
```

`->: induct` *call on the prover's heuristics to create subgoals corresponding to a proof by induction*

```
Inducting according to the scheme:

      (AND (IMPLIES (NOT (LISTP X)) (p A X))
           (IMPLIES (AND (LISTP X) (NOT (LISTP (CDR X))))
                    (p A X))
           (IMPLIES (AND (LISTP X)
                         (LISTP (CDR X))
                         (p A (ODDS X))
                         (p A (ODDS (CDR X))))
                    (p A X)))

Creating 3 new subgoals, (MAIN . 1), (MAIN . 2), and
(MAIN . 3).

The proof of the current goal, MAIN, has been completed.
However, the following subgoals of MAIN remain to be proved:
(MAIN . 1), (MAIN . 2), and (MAIN . 3).
Now proving (MAIN . 1).
```

`->: goals` *print the names of the goals that remain to be proved*

```
(MAIN . 1)
(MAIN . 2)
(MAIN . 3)
->: p

(IMPLIES (NOT (LISTP X))
         (EQUAL (OCCURRENCES A (MERGESORT X))
                (OCCURRENCES A X)))
```

`->: prove` *this is a “base case” which looks easy (* `X` *is the empty list), so we call on the prover to prove it automatically*

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

This simplifies, opening up the functions MERGESORT, LISTP,
OCCURRENCES, and EQUAL, to:

      T.

Q.E.D.


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

(IMPLIES (AND (LISTP X) (NOT (LISTP (CDR X))))
         (EQUAL (OCCURRENCES A (MERGESORT X))
                (OCCURRENCES A X)))
```

`->: prove` *another easy “base case”*

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

This formula simplifies, opening up the definitions of
MERGESORT, OCCURRENCES, and ADD1, to:
```

```
      T.

Q.E.D.


The current goal, (MAIN . 2), has been proved, and has no
dependents.
Now proving (MAIN . 3).
```

`->: goals` *notice that only one unproved goal remains*

```
(MAIN . 3)
```

`->: th` *same as* `p` *except that we also see the "top-level hypotheses" (there aren't any yet) and the "governors" (which we won't discuss here)*

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

`*** Active governors:` *As we said above, please ignore this*

```
There are no governors to display.

The current subterm is:
(IMPLIES
        (AND (LISTP X)
             (LISTP (CDR X))
             (EQUAL (OCCURRENCES A (MERGESORT (ODDS X)))
                    (OCCURRENCES A (ODDS X)))
             (EQUAL (OCCURRENCES A
                                 (MERGESORT (ODDS (CDR X))))
                    (OCCURRENCES A (ODDS (CDR X)))))
        (EQUAL (OCCURRENCES A (MERGESORT X))
               (OCCURRENCES A X)))
```

`->: promote` *set out to prove the consequent of the implication under the assumption that its antecedents hold*

```
->: th
```

```
*** Active top-level hypotheses:
H1.  (LISTP X)
H2.  (LISTP (CDR X))
H3.  (EQUAL (OCCURRENCES A (MERGESORT (ODDS X)))
            (OCCURRENCES A (ODDS X)))
H4.  (EQUAL (OCCURRENCES A
                         (MERGESORT (ODDS (CDR X))))
            (OCCURRENCES A (ODDS (CDR X))))

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

The current subterm is:
(EQUAL (OCCURRENCES A (MERGESORT X))
       (OCCURRENCES A X))
->: commands
```

let's see what we've done so far

```
The commands thus far (in reverse order, i.e. last one
first) have been:
1. PROMOTE
2. PROVE
3. PROVE
4. INDUCT
5. START
->: (comment Now open up (eXpand) the term (MERGESORT X))

->: commands

The commands thus far (in reverse order, i.e. last one
first) have been:
1. (COMMENT NOW OPEN UP (EXPAND) THE TERM (MERGESORT X))
2. PROMOTE
3. PROVE
4. PROVE
5. INDUCT
6. START
->: pp-top
```

prettyprint the conclusion, focusing on the "current subterm"

```
(*** (EQUAL (OCCURRENCES A (MERGESORT X))
            (OCCURRENCES A X))
     ***)
->: (dive 1 2)  focus on the first argument of  EQUAL, then
                on the second argument of that term

->: pp-top

(EQUAL (OCCURRENCES A
                    (*** (MERGESORT X) ***))
       (OCCURRENCES A X))
->: x  expand the definition of the current subterm's function symbol,
       i.e.  MERGESORT, on its actual arguments; then simplify

->: pp-top

(EQUAL (OCCURRENCES A
                    (*** (MERGE (MERGESORT (ODDS (CDR X)))
                                (MERGESORT (ODDS X)))
                         ***))
       (OCCURRENCES A X))
->: up

->: th

*** Active top-level hypotheses:
H1.  (LISTP X)
H2.  (LISTP (CDR X))
H3.  (EQUAL (OCCURRENCES A (MERGESORT (ODDS X)))
            (OCCURRENCES A (ODDS X)))
H4.  (EQUAL (OCCURRENCES A
                         (MERGESORT (ODDS (CDR X))))
            (OCCURRENCES A (ODDS (CDR X))))

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

The current subterm is:
(OCCURRENCES A
```

```
             (MERGE (MERGESORT (ODDS (CDR X)))
                    (MERGESORT (ODDS X))))
->: show-rewrites

No rewrite rules apply to the current term.
```

At this point we seem to be stuck. However, if we stare at the conclusion we may imagine a possible simplification: we may eliminate the call of `MERGE` *by noting that the number of occurrences of an object* `A` *in the* `MERGE` *of two lists is the sum of its occurrences in each list. Let us note a comment to that effect, then exit the interactive loop and prove such a theorem at the top level.*

```
->: (comment Now exit and prove the lemma OCCURRENCES-MERGE)

->: exit


Quitting the interactive proof checker.  Submit (VERIFY) to
get back in at this state.  **NOTE** -- No event has been
stored.
NIL

>(prove-lemma occurrences-merge (rewrite)
  (equal (occurrences a (merge x y))
         (plus (occurrences a x) (occurrences a y))))
```

The proof by induction is successful; we omit it here.

`>(verify)` *re-enter the interactive loop where we left off*

```
->: th

*** Active top-level hypotheses:
H1.  (LISTP X)
H2.  (LISTP (CDR X))
H3.  (EQUAL (OCCURRENCES A (MERGESORT (ODDS X)))
            (OCCURRENCES A (ODDS X)))
H4.  (EQUAL (OCCURRENCES A
                         (MERGESORT (ODDS (CDR X))))
```

```
            (OCCURRENCES A (ODDS (CDR X))))

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

The current subterm is:
(OCCURRENCES A
             (MERGE (MERGESORT (ODDS (CDR X)))
                    (MERGESORT (ODDS X))))
->: show-rewrites

1. OCCURRENCES-MERGE
     New term:
      (PLUS (OCCURRENCES A
                         (MERGESORT (ODDS (CDR X))))
            (OCCURRENCES A (MERGESORT (ODDS X))))

     Hypotheses:  <none>
->: rewrite  apply the rewrite rule manually, in order
             to keep control of the proof.

Rewriting with OCCURRENCES-MERGE.
->: p

(PLUS (OCCURRENCES A
                   (MERGESORT (ODDS (CDR X))))
      (OCCURRENCES A (MERGESORT (ODDS X))))
->: top  move to the top of the conclusion, so that the
         current subterm is the entire conclusion

->: th

*** Active top-level hypotheses:
H1.  (LISTP X)
H2.  (LISTP (CDR X))
H3.  (EQUAL (OCCURRENCES A (MERGESORT (ODDS X)))
            (OCCURRENCES A (ODDS X)))
H4.  (EQUAL (OCCURRENCES A
                         (MERGESORT (ODDS (CDR X))))
```

```
            (OCCURRENCES A (ODDS (CDR X))))

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

The current subterm is:
(EQUAL (PLUS (OCCURRENCES A
                          (MERGESORT (ODDS (CDR X))))
             (OCCURRENCES A (MERGESORT (ODDS X))))
       (OCCURRENCES A X))
```

Oh—now we see that we want to use the equality hypotheses.

```
->: undo

Undoing: TOP
->: p

(PLUS (OCCURRENCES A
                   (MERGESORT (ODDS (CDR X))))
      (OCCURRENCES A (MERGESORT (ODDS X))))
->: 1

->: p

(OCCURRENCES A
             (MERGESORT (ODDS (CDR X))))
```

use an equality hypothesis that equates this with another term

```
->: =

->: nx
```

move to the right sibling (the NeXt argument)

```
->: p

(OCCURRENCES A (MERGESORT (ODDS X)))
```

use an equality hypothesis that equates this with another term

```
->: =

->: p
```

```
(OCCURRENCES A (ODDS X))
->: top

->: th

*** Active top-level hypotheses:
H1.  (LISTP X)
H2.  (LISTP (CDR X))
H3.  (EQUAL (OCCURRENCES A (MERGESORT (ODDS X)))
            (OCCURRENCES A (ODDS X)))
H4.  (EQUAL (OCCURRENCES A
                         (MERGESORT (ODDS (CDR X))))
            (OCCURRENCES A (ODDS (CDR X))))

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

The current subterm is:
(EQUAL (PLUS (OCCURRENCES A (ODDS (CDR X)))
             (OCCURRENCES A (ODDS X)))
       (OCCURRENCES A X))
->: (drop 3 4)  now drop the last two hypotheses; after all,
                we have already used them

Dropping hypotheses 3 and 4.
->: th

*** Active top-level hypotheses:
H1.  (LISTP X)
H2.  (LISTP (CDR X))

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

The current subterm is:
(EQUAL (PLUS (OCCURRENCES A (ODDS (CDR X)))
             (OCCURRENCES A (ODDS X)))
       (OCCURRENCES A X))
->: prove  can we finish this by calling the prover?
```

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

This conjecture simplifies, expanding OCCURRENCES, to the
following two new conjectures:
```

The proof succeeds! We'll omit the output here.

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


The current goal, (MAIN . 3), has been proved, and has no
dependents.

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

make an "event" to add to the CHRONOLOGY

```
->: (exit occurrences-mergesort (rewrite))


The indicated goal has been proved.  Here is the desired event:
(PROVE-LEMMA OCCURRENCES-MERGESORT
             (REWRITE)
             (EQUAL (OCCURRENCES A (MERGESORT X))
                    (OCCURRENCES A X))
             ((INSTRUCTIONS INDUCT PROVE PROVE PROMOTE
                            (COMMENT NOW OPEN UP
                                     (EXPAND)
                                     THE TERM
                                     (MERGESORT X))
                            (DIVE 1 2)
                            X UP
                            (COMMENT NOW EXIT AND PROVE THE LEMMA
                                     OCCURRENCES-MERGE)
                            (REWRITE OCCURRENCES-MERGE)
                            (DIVE 1)
                            = NX = TOP
```

```
                              (DROP 3 4)
                              PROVE)))
Do you want to submit this event?
Y (Yes), R (Yes and replay commands), or N (No) ? y
```

This ends our proof. However, often such interactive proofs go through automatically, once one has proved appropriate rewrite rules along the way. Having proved such a rule (`OCCURRENCES-MERGE`), it is tempting to give it a try (after executing `(UBT)` in order to undo the event just proved):

```
>(prove-lemma occurrences-mergesort (rewrite)
  (equal (occurrences a (mergesort x))
         (occurrences a x)))
```

However, the proof fails. But with the lemma below, it succeeds. This lemma is exactly the goal that was left before the final call of prove in the interactive proof.

```
>(prove-lemma plus-occurrences-odds (rewrite)
  (implies (and (listp x) (listp (cdr x)))
           (equal (plus (occurrences a (odds (cdr x)))
                        (occurrences a (odds x)))
                  (occurrences a x))))
```

Since the interactive proof encountered this exact same lemma, maybe we believe that somehow the proof of our main theorem `OCCURRENCES-MERGESORT` *should* have gone through automatically at this point. *However,* recall that we explicitly dropped two equality hypotheses in our interactive proof before calling on the theorem prover to finish it. By isolating the rewrite rule shown immediately above, we allow this fact about `OCCURRENCES` to be applied even while those equality hypotheses are still present, without the danger that a proof of that final goal by induction will get "confused."

Here is the final non-interactive proof, using the lemma just proved above.

```
>(prove-lemma occurrences-mergesort (rewrite)
  (equal (occurrences a (mergesort x))
         (occurrences a x)))


     Name the conjecture *1.
```

Perhaps we can prove it by induction. There are two plausible inductions, both of which are unflawed. So we will choose the one suggested by the largest number of nonprimitive recursive functions. We will induct according to the following scheme:

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

The lemmas SUB1-ADD1 and MERGESORT-HELPER and the definitions of LESSP and LENGTH establish that the measure (LENGTH X) decreases according to the well-founded relation LESSP in each induction step of the scheme. The above induction scheme produces the following three new goals:

```
Case 3. (IMPLIES (NOT (LISTP X))
                 (EQUAL (OCCURRENCES A (MERGESORT X))
                        (OCCURRENCES A X))).

  This simplifies, expanding MERGESORT, LISTP, OCCURRENCES,
  and EQUAL, to:

        T.

Case 2. (IMPLIES (AND (LISTP X) (NOT (LISTP (CDR X))))
                 (EQUAL (OCCURRENCES A (MERGESORT X))
                        (OCCURRENCES A X))).

  This simplifies, unfolding MERGESORT, OCCURRENCES, and
  ADD1, to:

        T.
```

```
Case 1. (IMPLIES
         (AND
             (LISTP X)
             (LISTP (CDR X))
             (EQUAL (OCCURRENCES A (MERGESORT (ODDS X)))
                    (OCCURRENCES A (ODDS X)))
             (EQUAL (OCCURRENCES A
                                 (MERGESORT (ODDS (CDR X))))
                    (OCCURRENCES A (ODDS (CDR X)))))
         (EQUAL (OCCURRENCES A (MERGESORT X))
                (OCCURRENCES A X))).

  This simplifies, rewriting with the lemmas
  PLUS-OCCURRENCES-ODDS and OCCURRENCES-MERGE, and
  expanding the definitions of MERGESORT and OCCURRENCES,
  to:

        T.


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


[ 0.0 0.2 0.1 ]

OCCURRENCES-MERGESORT
T

>
```

5 Example Event Files

We now summarize a number of applications of Nqthm and Pc-Nqthm. Each entry below is of the following form:

```
(Author, citation, filename)
Description
```

The citation may be omitted; in that case, no published description of the work is available and the interested reader should look at the indicted file, which is included with the system. Many of the files have explanatory comments. Each file has been successfully processed by `PROVE-FILE`. The files are listed in alphabetical order. Much of this text appears essentially in [Kau92a].

5.1 Nqthm example event files

First we list the example files for Nqthm.

(Boyer, `"basic/alternating.events"`)
a formalization and correctness proof of the "Gilbreath Trick" [Gar60, Gil58], a card trick having to do with the outcome of shuffling a deck of cards that has been previously arranged into alternating colors; the Nqthm attack on this problem was inspired by Gerard Huet's use of the COQ theorem prover to do the proof[Hue91]

(Moore, [Moo91], `"basic/async18.events"`)
a model of asynchronous communication and a proof of the reliability of the biphase mark communications protocol

(Boyer and Moore, [BM88a], `"basic/binomial.events"`)
the binomial theorem expressed with `FOR` and a proof thereof

(Bronstein and Talcott, [BT88, Bro89, BT89b, BT89a], `"bronstein/*.events"`)
a collection of twenty six event files that are described in the four papers cited above; the work includes a formalization of "string-functional semantics" for circuit descriptions and its use to verify the correctness properties of many circuits, including the Saxe-Leiserson retimed correlator, a pipelined ripple adder, and an abstract pipelined cpu

(Boyer, Moore, and Green, [BGM90],
`"basic/controller.events"`)
a model of the problem of controlling a vehicle's course and a proof that under certain conditions a particular program keeps the vehicle within a certain corridor of the desired course and, under more ideal conditions, homes to the course

(Cowles, `"basic/fibsums.events"`)
proofs of several interesting theorems about the sums of Fibonacci numbers

(Boyer and Moore, [BM81], `"basic/fortran.events"`)
supporting definitions for a Fortran verification condition generator

(Boyer, Goldschlag, Kaufmann, and Moore, [BGKM],
`"basic/fs-examples.events"`)
illustrations of the use of constrained functions and functional instantiation

(Russinoff, [Rus92], `"basic/gauss.events"`)
the original Nqthm proof of Gauss' law of quadratic reciprocity

(Russinoff, [Rus92], `"basic/new-gauss.events"`)
an improved proof of Gauss' law of quadratic reciprocity (after all, Gauss proved it eight times!)

(Boyer and Moore, `"basic/parser.events"`)
a formalization of the syntax and abbreviation conventions of the Nqthm extended logic, expressed as a function from lists of ASCII character codes to the quotations of formal terms

(Boyer, `"basic/peter.events"`)
a sequence of lemmas describing the relationship between Ackermann's original function and R. Peter's version of it

(Boyer, `"basic/pr.events"`)
a proof of the existence of nonprimitive recursive functions

(Boyer and Moore, approximately Appendix A of [ACL],
`"basic/proveall.events"`)

elementary list processing, number theory through Euclid's theorem and prime factorization, soundness and completeness of a tautology checker, correctness of the `CANCEL` metafunction, correctness of a simple assembly language program, correctness of a simple optimizing expression compiler

(Boyer and Moore, [BM88a], `"basic/quant.events"`)
illustrations of the use of `V&C$` and `FOR`, including a study of several partial functions and functions, such as the "91 function," that recurse on the value of their own recursive calls

(Boyer and Moore, [BM84c], `"basic/rsa.events"`)
proof of the invertibility of the public key encryption algorithm of Rivest, Shamir, and Adleman

(Moore, `"basic/small-machine.events"`)
a simple operational semantics and its use to prove program properties directly and via the so-called "functional" and "inductive assertion" methods

(Moore, `"basic/tic-tac-toe.events"`)
a formalization of what it means for a program to play non-losing tic-tac-toe, the proof that a certain algorithm does so, and the successive refinement of the algorithm into the functional expression of an iterative number-crunching program

(Boyer and Moore, [BM84a], `"basic/tmi.events"`)
proof of the Turing completeness of Pure Lisp

(Boyer and Moore, [BM84b], `"basic/unsolv.events"`)
proof of the unsolvability of the halting problem for Pure Lisp

(Russinoff, [Rus85], `"basic/wilson.events"`)
proof of Wilson's theorem

(Moore, [Moo79], `"basic/ztak.events"`)
proof of the termination of Takeuchi's function

(Bevier, [Bev87], `"bevier/kit.events"`)
the formalization, implementation and proof that a simple separation

kernel (implementing multi-processing on a uniprocessor) provides process scheduling, error handling, message passing, and interfaces to asynchronous devices

(Cowles, `"cowles/intro-eg.events"`)
: a brief introduction to Nqthm intended for mathematicians and a proof of a theorem about factorial

(Cowles, `"cowles/shell.events"`)
: alternative ways to decompose sequences and a study of Nqthm's shell principle

(Flatau, [Fla92], `"flatau/app-c-d-e.events"`)
: the development and proof of correctness of a compiler and runtime system for a subset of the Nqthm language (including `IF`, `CONS`, and subroutine call) requiring dynamic storage allocation; this event list corresponds to Appendices C, D, and E of [Fla92] and deals with a runtime system that does not provide a garbage collector.

(Flatau, [Fla92], `"flatau/app-f.events"`)
: this event file is analogous to the immediately preceding one, but corresponds to Appendix F of [Fla92] and deals with a runtime system including a reference counting garbage collector.

(Moore, [Moo88], `"fm9001-piton/big-add.events"`)
: a proof of the correctness of a Piton program for adding arbitrarily long numbers in base 2^{32}

(Brock and Hunt, [HB92], `"fm9001-piton/fm9001.events"`)
: formalizations of a netlist description language, the machine code for the 32-bit FM9001 microprocessor, the design of an implementation of the processor, and a proof of the correspondence of the design and the machine code specification

(Wilding, [Wil92], `"fm9001-piton/nim-piton.events"`)
: a proof that a given 300-line Piton program plays the game of Nim optimally; the program is also shown to be loadable onto the FM9001 (satisfying the requirements of the correctness theorem for Piton); bounds on the program's execution time have been proved using Pc-Nqthm.

(Moore, [Moo88], `"fm9001-piton/piton.events"`)
the definition of the Piton assembly language, its implementation on the FM9001 via a compiler, assembler and linker, and a proof of the correctness of the FM9001 implementation

(Boyer and Moore, [BM81], `"fortran-vcg/fortran.events"`)
the same file as `basic/fortran`, above, which is duplicated on this subdirectory for technical reasons

(Boyer and Moore, [BM81], `"fortran-vcg/fsrch.events"`)
proofs of the verification conditions for a Fortran implementation of the Boyer-Moore fast string searching algorithm

(Boyer and Moore, [BGM90], `"fortran-vcg/isqrt.events"`)
proofs of the verification conditions for a Fortran implementation of the integer version of Newton's square root algorithm

(Boyer and Moore, [BM91], `"fortran-vcg/mjrty.events"`)
proofs of the verification conditions for a Fortran implementation of a linear-time majority vote algorithm

(Hunt, [Hun85], `"hunt/fm8501.events"`)
formalizations of the machine code for the 16-bit FM8501 microprocessor, a register transfer model of a microcoded implementation of the machine, and a proof of their correspondence

(Kaufmann, see Young [You90],
`"kaufmann/expr-compiler.events"`)
the proof of correctness of a simple expression compiler, designed as an exercise for beginners

(Kaufmann, `"kaufmann/foldr.events"`)
an illustration of a method of proving permutation-independence of list processing functions

(Kaufmann, [Kau91b], `"kaufmann/generalize-all.events"`)
the correctness of a generalization algorithm that operates in the presence of free variables

(Kaufmann, [Kau92b], "kaufmann/koenig.events")
a proof of Koenig's tree lemma

(Kaufmann, [Kau91a], "kaufmann/locking.events")
a model of a simple data base against which read and write transactions can occur

(Kaufmann, "kaufmann/mergesort-demo.events")
the correctness of a merge sort function, similar to the one in Section 4 in this paper

(Kaufmann, [Kau88b], "kaufmann/note-100.events")
the proof of Ramsey's theorem for exponent 2, finite case, described in a style intended to assist those wishing to improve their effectiveness with Nqthm

(Kaufmann, "kaufmann/partial.events")
an approach to handling partial functions with Nqthm

(Kaufmann, "kaufmann/permutationp-subbagp.events")
a formalization of the notion of permutation via bags

(Kaufmann, [Kau92b], "kaufmann/ramsey.events")
a proof of Ramsey's theorem for the infinite case

(Kaufmann, [Kau90b], "kaufmann/rotate.events")
a proof about rotations of lists, intended as an introduction to Nqthm

(Kaufmann and Jamsek, "kaufmann/rpn.events")
an exercise in reverse Polish notation evaluation

(Kaufmann, "kaufmann/shuffle.events")
another solution to the Gilbreath card trick challenge (see example file "basic/alternating.events")

(Kunen, "kunen/ack.events")
an illustrative definition of Ackermann's function

(Kunen, "kunen/new-prime.events")
an alternative proof of the fundamental theorem of arithmetic that –

unlike the one presented in [ACL] – does not use concepts not involved in the statement of the theorem

(Bevier, `"numbers/bags.events"`)
a library of useful definitions and lemmas about bags

(Wilding, `"numbers/extras.events"`)
a trivial extension of the `integers` library used in `fib2` below

(Wilding, [Wil91], `"numbers/fib2.events"`)
a proof of Matijasevich's lemma about Fibonacci numbers

(Bevier, Kaufmann, and Wilding, [Kau90a], `"numbers/integers.events"`)
a library of useful definitions and lemmas about the integers

(Bevier, Kaufmann, and Wilding, [Bev88], `"numbers/naturals.events"`)
a library of useful definitions and lemmas about the natural numbers

(Wilding, `"numbers/nim.events"`)
a formalization of the game of Nim and a proof that a certain algorithm implements a winning strategy

(Shankar, [Sha88], `"shankar/church-rosser.events"`)
a proof of the Church-Rosser theorem for lambda-calculus

(Shankar, [Sha86], `"shankar/goedel.events"`)
a proof of Gödel's incompleteness theorem for Shoenfield's first order logic extended with Cohen's axioms for hereditarily finite set theory, Z2

(Shankar, [Sha85], `"shankar/tautology.events"`)
a proof that every tautology has a proof in Shoenfield's propositional logic

(Nagayama and Talcott, [NT91], `"talcott/mutex-atomic.events"`)
a proof of the local correctness of a mutual exclusion algorithm under a certain "atomicity assumption"

(Nagayama and Talcott, [NT91],
"`talcott/mutex-molecular.events`")
a proof of the local correctness of a mutual exclusion algorithm without the "atomicity assumption" mentioned above

(Yu, "`yu/amax.events`")
the correctness proof for the MC68020 machine code produced by the Gnu C compiler for a C program that finds the maximum value in an integer array

(Yu, [Yu92], "`yu/asm.events`")
the correctness proof for the MC68020 machine code produced by the Gnu C compiler for a trivial C program that uses embedded assembly code (the object being to demonstrate that embedded assembly code can be handled)

(Yu, [Yu92], "`yu/bsearch.events`")
the correctness proof for the MC68020 machine code produced by the Gnu C compiler for a binary search program written in C

(Yu, [Yu92], "`yu/cstring.events`")
the correctness proofs for the MC68020 machine code produced by the Gnu C compiler for 21 of the 22 C String Library functions from the Berkeley Unix C string library; the proof for each function is broken into two "phases;" the first phase establishes the correspondence of the machine code and a suitable recursive function and the second phase establishes that the recursive function has the specified properties; the file `yu/cstring.events` actually contains the second phase proofs for all of the string functions handled; the first phase proof for each string function is contained in a separate `events` file named for the string function, e.g., `yu/memchr.events`, `yu/memcmp.events`, `yu/strnspn.events`, etc.

(Yu, "`yu/fixnum-gcd.events`")
the correctness proof for the MC68020 machine code produced by the AKCL compiler for a Common Lisp program that computes the greatest common divisor of two `FIXNUM`s

(Yu, [Yu92], `"yu/fmax.events"`)
the correctness proof for the MC68020 machine code produced by the Gnu C compiler for a trivial C program to compute the maximum of two integers according to a supplied comparison function (the object being to demonstrate that C "function pointers" are handled)

(Yu, [Yu92] (also [Hea08]), `"yu/gcd.events"`)
the correctness proof for the MC68020 machine code produced by the Gnu C compiler for Euclid's greatest common divisor algorithm written in C

(Yu, [Yu92], `"yu/gcd3.events"`)
the correctness proof for the MC68020 machine code produced by the Gnu C compiler for a C program consisting of two nested calls of the GCD program (the object being to demonstrate that procedure call is handled in a way that allows hierarchical verification)

(Yu, [Yu90], `"yu/group.events"`)
proofs of two theorems in finite group theory, the first is about kernels of homomorphisms and the second is the Lagrange theorem

(Yu, `"yu/isqrt.events"`)
the correctness proof for the MC68020 machine code produced by the GNU C compiler for C program for computing integer square roots via Newton's method

(Yu, `"yu/isqrt-ada.events"`)
the correctness proof for the MC68020 machine code produced by the Verdix Ada compiler from an Ada program for computing integer square roots via Newton's method

(Yu, `"yu/log2.events"`)
the correctness proof for the MC68020 machine code produced by the Gnu C compiler for a C program for computing integer logarithms (base 2) e.g., repeated division by 2

(Yu, [Yu92], `"yu/mc20-0.events"`)
some utilities involved in the formal specification of the MC68020

(Yu, [BY92], `"yu/mc20-1.events"`)
: the formal specification of about 80% of the user available instructions for the Motorola MC68020 microprocessor

(Yu, [Yu92], `"yu/mc20-2.events"`)
: a library of useful definitions and lemmas about the formalization of the MC68020

(Yu, `"yu/mjrty.events"`)
: the correctness proof for the MC68020 machine code produced by the Gnu C compiler for a linear time majority vote algorithm written in C

(Yu, `"yu/qsort.events"`)
: the correctness proof for the MC68020 machine code produced by the Gnu C compiler for Hoare's *in situ* quick sort program written in C; the source code is that on page 87 of [KR88] except that inline code is used rather than the subroutine `swap`; Yu reports (private communication) that this change was made only because he was, at the time, investigating methods of proving recursive programs correct and did not want to be distracted by other subroutine calls

(Yu, [Yu92], `"yu/switch.events"`)
: the correctness proof for the MC68020 machine code produced by the GNU C compiler for a trivial C program that employs the `switch` statement (the object being to demonstrate the technique used to compile `switch` can be handled)

(Yu, `"yu/zero.events"`)
: the correctness proof for the MC68020 machine code produced by the GNU C compiler for a C program that zeros an integer array (the object being to demonstrate that writes to memory can be handled)

5.2 Pc-Nqthm example event files

Next we list the example files for Pc-Nqthm.

(Kaufmann, `"basic/arith.events"`)
: some supporting arithmetic events for other event files in this directory

(Kaufmann, `"basic/hanoi.events"`)
proof of correctness of a Towers of Hanoi program

(Kaufmann, `"basic/pigeon-hole.events"`)
proof of a version of the pigeon-hole principle

(Kaufmann, `"basic/ramsey1.events"`)
proof of correctness of Ramsey's Theorem for exponent 2

(Kaufmann, `"basic/ramsey2.events"`)
proof that a certain binomial coefficient serves as a bound on the Ramsey number

(Kaufmann, `"basic/square.events"`)
ugly proof of an ugly formalization of the irrationality of the square root of 2

(Kaufmann, `"basic/subset.events"`)
some supporting events about lists and their use as an implementation of sets

(Kaufmann, `"basic/symmetric-difference.events"`)
commutativity and associativity of symmetric difference as a set operation

(Kaufmann, `"basic/transitive-closure.events"`)
proof of correctness of a transitive closure algorithm

(Kaufmann, [Kau86], `"basic/tsquare.events"`)
proof of correctness of a "true square" algorithm of Gries

(Kaufmann, [Kau92b], `"defn-sk/csb.events"`)
proof of a formalization of the Schroeder-Bernstein theorem of set theory

(Kaufmann, `"defn-sk/finite-state-machine-example.events"`)
a little finite state machine example

(Kaufmann, [Kau92b], `"defn-sk/koenig.events"`)
a formalization of Koenig's Tree Lemma, which says that any finitely branching tree which is infinite has an infinite branch

(Kaufmann, [Kau92b], `"defn-sk/ramsey.events"`)
proof of a formalization of the infinite Ramsey Theorem for exponent 2

(Bevier, [Bev88], `"dmg/bags.events"`)
some supporting events about bags

(Goldschlag, [Gol91], `"dmg/dining.events"`)
the verification of a dining philosopher's program, under the assumptions of deadlock freedom and strong fairness, using a mechanized implementation of Unity on the Boyer-Moore prover

(Goldschlag, [Gol90d], `"dmg/fifo.events"`)
the verification of both the safety and liveness properties of an n-node delay insensitive fifo circuit, using a mechanized implementation of Unity on the Boyer-Moore prover

(Goldschlag [Gol90c], `"dmg/interpreter.events"`)
a mechanized implementation of Unity on the Boyer-Moore prover

(Goldschlag [GolThesis], `"dmg/me.events"`)
verification of an n-processor program satisfying both mutual exclusion and absence of starvation, using a mechanized implementation of Unity on the Boyer-Moore prover

(Goldschlag [GolThesis], `"dmg/min.events"`)
the correctness of a distributed algorithm that computes the minimum node value in a tree, using a mechanized implementation of Unity on the Boyer-Moore prover

(Bevier and Wilding, [Bev88], `"dmg/naturals.events"`)
some supporting events about natural numbers

(Kaufmann, [Kau91b], `"generalize/*.events"`)
the correctness of a generalization algorithm that operates in the presence of free variables; same as the corresponding events from the Nqthm example suite, except that the quantifier (`DEFN-SK`, [Kau92b]) events have been replaced by `DCL` and `ADD-AXIOM` events in that version

(Young, [You89], `"mg/*.events"`)
a mechanically-verified code-generator for micro-Gypsy, which is a Pascal-like language

(Good, Siebert, Young, [GSY], `"middle-gypsy/*.events"`)
a mathematical definition of a subset of the Gypsy 2.05 language, including a preliminary rationals library created by Matt Wilding

(Wilding, `"wilding/ground-resolution.events"`)
a proof of the completeness of ground resolution using Bledsoe's excess literal technique

References

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

[ACLH] R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, New York, 1988.

[BBH72] W. Bledsoe, R. S. Boyer, and W. Henneman. Computer proofs of limit theorems. *Artificial Intelligence*, 3:27–60, 1972.

[BGKM] R.S. Boyer, D.M. Goldschlag, M. Kaufmann, and J S. Moore. Functional instantiation in first order logic. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 7–26. Academic Press, 1991.

[BGM90] R. S. Boyer, M. W. Green, and J S. Moore. The Use of a Formal Simulator to Verify a Simple Real Time Control Program. In W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J. Misra, editor, *Beauty is Our Business: A Birthday Salute to Edsger W. Dijkstra*, pages 54–66. Springer-Verlag Texts and Monographs in Computer Science, 1990.

[BKY90] W. Bevier, M. Kaufmann, and W. Young. Translation of a Gypsy compiler example into the Boyer-Moore logic. Internal Note 169, Computational Logic, Inc., January 1990.

[BM75] R. S. Boyer and J S. Moore. Proving theorems about Lisp functions. *JACM*, 22(1):129–144, 1975.

[BM77] R. S. Boyer and J S. Moore. A lemma driven automatic theorem prover for recursive function theory. In *Proceedings of the 5th Joint Conference on Artificial Intelligence*, pages 511–519. International Joint Conferences on Artifical Intelligence, 1977.

[BM81] R. S. Boyer and J S. Moore. A verification condition generator for Fortran. In *The Correctness Problem in Computer Science*. Academic Press, London, 1981.

[BM81a] R. S. Boyer and J S. Moore. The mechanical verification of a Fortran square root program. Csl report, SRI International, 1981.

[BM81b] R. S. Boyer and J S. Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. In *The Correctness Problem in Computer Science*. Academic Press, London, 1981.

[BM84a] R. S. Boyer and J S. Moore. A mechanical proof of the Turing completeness of Pure Lisp. In *Automated Theorem Proving: After 25 Years*, pages 133–167. American Mathematical Society, Providence, R.I., 1984.

[BM84b] R. S. Boyer and J S. Moore. A mechanical proof of the unsolvability of the halting problem. *JACM*, 31(3):441–458, 1984.

[BM84c] R. S. Boyer and J S. Moore. Proof checking the RSA public key encryption algorithm. *American Mathematical Monthly*, 91(3):181–189, 1984.

[BM85] R. S. Boyer and J S. Moore. Program verification. *Journal of Automated Reasoning*, 1(1):17–23, 1985.

[BM88a] R. S. Boyer and J S. Moore. The addition of bounded quantification and partial functions to a computational logic and its theorem prover. *Journal of Automated Reasoning*, 4(2):117–172, 1988.

[BM88b] R. S. Boyer and J S. Moore. Integrating decision procedures into heuristic theorem provers: A case study with linear arithmetic. In *Machine Intelligence 11*. Oxford University Press, 1988.

[BM91] R. S. Boyer and J S. Moore. *MJRTY - A Fast Majority Vote Algorithm*, pages 105–117. Automated Reasoning Series, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1991.

[BSvHIS] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A Heuristic for Guiding Inductive Proofs. Department of Artificial Intelligence Research Paper No. 567, Edinburgh University 1991. To appear in *The Journal of Artificial Intelligence.*

[BT88] A. Bronstein and C. Talcott. String-functional semantics for formal verification of synchronous circuits, Report No. STAN-CS-88-1210. Technical report, Computer Science Department, Stanford University, 1988.

[BT89a] A. Bronstein and C. Talcott. Formal verification of pipelines based on string-functional semantics. In *IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, Leuven, Belgium*, 1989.

[BT89b] A. Bronstein and C. Talcott. Formal verification of synchronous circuits based on string-functional semantics: The 7 Paillet circuits in Boyer-Moore. In *C-Cube 1989 Workshop on Automatic Verification Methods for Finite State Systems. LNCS 407*, pages 317–333, 1989.

[BY92] R.S. Boyer and Y. Yu. A formal specification of some user mode instructions for the motorola 68020. Technical Report TR-92-04, Computer Sciences Department, University of Texas, Austin, February 1992.

[Bev87] W. Bevier. *A Verified Operating System Kernel.* PhD thesis, University of Texas at Austin, 1987.

[Bev88] W. Bevier. A library for hardware verification. Internal Note 57, Computational Logic, Inc., 1717 W. Sixth Street, Suite 290, Austin, TX 78703, August 1988.

[Bev89] W. Bevier. Kit and the short stack. *Journal of Automated Reasoning*, 5(4):519–530, 1989.

[Ble71] W. W. Bledsoe. Splitting and reduction heuristics in automatic theorem proving. *Artificial Intelligence*, 2:55–77, 1971.

[Bou68] N. Bourbaki. *Elements of Mathematics*. Addison Wesley, Reading, Massachusetts, 1968.

[Bro89] A. Bronstein. *MLP: String-functional semantics and Boyer-Moore mechanization for the formal verification of synchronous circuits*. PhD thesis, Stanford University, 1989.

[Bur69] R. Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41–48, 1969.

[Cho88] S. Chou. *Mechancial Geometry Theorem Proving*. Reidel, 1988.

[Fla92] A. Flatau. *A Verified Implementation of an Applicative Language with Dynamic Storage Allocation*. PhD thesis, University of Texas, 1992. Also available through Computational Logic, Inc., Suite 290, 1717 West Sixth Street, Austin, TX 78703.

[GSY] D. I. Good, Ann E. Siebert, and W. D. Young. Middle Gypsy 2.05 Definition. Technical Report 59, Computational Logic, Inc., May 1990.

[Gar60] M. Gardner. Mathematical recreation column. *Scientific American*, 203(2):149–154, August, 1960.

[Gil58] N. Gilbreath. Magnetic colors. *The Linking Ring*, 38(5):60, 1958.

[Gol90a] D. M. Goldschlag. Proving proof rules: A proof system for concurrent programs. *Compass '90*, June 1990.

[Gol90b] D. M. Goldschlag. Mechanically verifying concurrent programs with the Boyer-Moore prover. *IEEE Transactions on Software Engineering*, September 1990. To appear.

[Gol90c] D. M. Goldschlag. Mechanizing Unity. In *Proceedings of the IFIP TC2/WG2.3 Working Conference on Programming Concepts and Methods*. Elsevier, Amsterdam, 1990.

[Gol90d] D. M. Goldschlag. Mechanically Verifying Safety and Liveness Properties of Delay Insensitive Circuits. *Computer Aided Verification 1991*, July, 1991.

[Gol91] D. M. Goldschlag. A Mechanical Formalization of Several Fairness Notions. In S. Prehn and W.J. Toetenel, editors, *VDM '91: Formal Software Development Methods*, pages 133–167. Springer-Verlag Lecture Notes in Computer Science 551, 1991.

[GolThesis] D. M. Goldschlag. *Mechanically Verifying Concurrent Programs*. PhD thesis, University of Texas at Austin, 1992.

[Goo64] R. L. Goodstein. *Recursive Number Theory*. North-Holland Publishing Company, Amsterdam, 1964.

[HB92] W.A. Hunt Jr. and B. Brock. A formal HDL and its use in the FM9001 verification. *Proceedings of the Royal Society*, (to appear, April 1992), 1992.

[Hea08] T. L. Heath (translation and commentary). *The Thirteen Books of Euclid's Elements*. Dover, New York, 1908. p. 298, Vol 2., i.e. Proposition 2, Book VII.

[Hue91] G. Huet. The Gilbreath trick: A case study in axiomatization and proof development in the COQ proof assistant. Technical Report 1511, INRIA, Domaine de Voluceau, Rocquencourt B.P. 105, 78153 Le Chesnay Cedex, France, 1991.

[Hun85] W.A. Hunt, Jr. *FM8501: A Verified Microprocessor*. PhD thesis, University of Texas at Austin, 1985.

[Hun89] W. A. Hunt, Jr.. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, 1989.

[KR88] B.W. Kernighan and D.M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, 1988.

[KW89] M. Kaufmann and M. Wilding. A parallel version of the Boyer-Moore prover. Technical Report 39, Computational Logic, Inc., February 1989.

[KY87] M. Kaufmann and W. D. Young. Comparing Gypsy and the Boyer-Moore logic for specifying secure systems. Technical report, Institute for Computing Science, University of Texas at Austin, May 1987. ICSCA-CMP-59.

[Kau86] M. Kaufmann. A mechanically-checked semi-interactive proof of correctness of Gries's algorithm for finding the largest size of a square true submatrix. Internal Note 236, Institute for Computing Science, University of Texas at Austin, October 1986.

[Kau87] M. Kaufmann. A formal semantics and proof of soundness for the logic of the Nqthm version of the Boyer-Moore theorem prover. Internal Note 229, Institute for Computing Science, University of Texas at Austin, February 1987.

[Kau88] M. Kaufmann. A user's manual for an interactive enhancement to the Boyer-Moore theorem prover. Technical Report 19, Computational Logic, Inc., May 1988.

[Kau88a] M. Kaufmann. Boyer-Moore-ish Micro Gypsy and a prototype hardware expander. Internal Note 73, Computational Logic, Inc., August 1988.

[Kau88b] M. Kaufmann. An example in Nqthm: Ramsey's theorem. Internal Note 100, Computational Logic, Inc., November 1988.

[Kau88c] M. Kaufmann. A mutual recursion and dependency analysis tool for Nqthm. Internal Note 99, Computational Logic, Inc., 1988.

[Kau89a] M. Kaufmann. Addition of free variables to an interactive enhancement of the Boyer-Moore theorem prover. Technical Report 42, Computational Logic, Inc., Austin, Texas, May 1989.

[Kau89b] M. Kaufmann. A user's manual for RCL. Internal Note 157, Computational Logic, Inc., October 1989.

[Kau90a] M. Kaufmann. An integer library for Nqthm. Internal Note 182, Computational Logic, Inc., March 1990.

[Kau90b] M. Kaufmann. An instructive example for beginning users of the Boyer-Moore theorem prover. Internal Note 185, Computational Logic, Inc., 1717 W. Sixth Street, Suite 290, Austin, TX 78703, April 1990.

[Kau91a] M. Kaufmann. A simple example for Nqthm: Modeling locking. Internal Note 216, Computational Logic, Inc., 1717 W. Sixth Street, Suite 290, Austin, TX 78703, February 1991.

[Kau91b] M. Kaufmann. Generalization in the presence of free variables: A mechanically-checked proof for one algorithm. *Journal of Automated Reasoning*, 7(1), March 1991.

[Kau92a] M. Kaufmann. Response to FM91 Survey of Formal Methods: Nqthm and Pc-Nqthm. Technical Report 75, Computational Logic, Inc., Austin, Texas, March 1992.

[Kau92b] M. Kaufmann. An extension of the Boyer-Moore theorem prover to support first-order quantification. *J. Automated Reasoning*, 9(3):355–372, December 1992.

[McC60a] J. McCarthy. *The Lisp Programmer's Manual.* M.I.T. Computation Center, 1960.

[McC60b] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the Association for Computing Machinery*, 3(4):184–195, 1960.

[McC62a] J. McCarthy. Computer programs for checking mathematical proofs. In *Recursive Function Theory, Proceedings of a Symposium in Pure Mathematics*, volume V, pages 219–227, Providence, Rhode Island, 1962. American Mathematical Society.

[McC62b] J. McCarthy. Towards a mathematical science of computation. In *Proceedings of IFIP Congress*, pages 21–28. North-Holland, 1962.

[McC63] J. McCarthy. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*. North-Holland Publishing Company, Amsterdam, The Netherlands, 1963.

[McC65] J. McCarthy et al. *LISP 1.5 Programmer's Manual*. The MIT Press, Cambridge, Massachusetts, 1965.

[Moo79] J S. Moore. A mechanical proof of the termination of Takeuchi's function. *Information Processing Letters*, 9(4):176–181, 1979.

[Moo88] J S. Moore. Piton: A verified assembly-level language. Technical Report CLI-22, Computational Logic, Inc., Austin, Tx, June 1988.

[Moo89] J S. Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4):461–492, 1989.

[Moo89a] J S. Moore et al. Special issue on system verification. *Journal of Automated Reasoning*, 5(4):409–530, 1989.

[Moo91] J S. Moore. A formal model of asynchronous communication and its use in mechanically verifying a biphase mark protocol. Technical Report 68, Computational Logic, Inc., 1717 W. Sixth Street, Suite 290, Austin, TX 78703, August 1991. To appear in *Formal Aspects of Computing*, 1993.

[NT91] M. Nagayama and C. Talcott. An Nqthm mechanization of "An exercise in the verification of multi-process programs". Technical Report STAN-CS-91-1370, Computer Science Department, Stanford University, 1991.

[Rus85] D. M. Russinoff. An experiment with the Boyer-Moore theorem prover: A proof of Wilson's theorem. *Journal of Automated Reasoning*, 1(2):121–139, 1985.

[Rus92] D.M. Russinoff. A mechanical proof of quadratic reciprocity. *Journal of Automated Reasoning*, 8(1):3–21, 1992.

[SLP82] R. Shostak, L. Lamport, and M. Pease. The Byzantine generals problem. *ACM TOPLAS*, 4(3):382–401, July 1982.

[Sha85] N. Shankar. Towards mechanical metamathematics. *Journal of Automated Reasoning*, 1(4):407–434, 1985.

[Sha86] N. Shankar. *Proof Checking Metamathematics*. PhD thesis, University of Texas at Austin, 1986.

[Sha88] N. Shankar. A mechanical proof of the Church-Rosser theorem. *JACM*, 35(3):475–522, 1988.

[Sho67] J. R. Shoenfield. *Mathematical Logic*. Addison-Wesley, Reading, Ma., 1967.

[Sko67] T. Skolem. The foundations of elementary arithmetic established by means of the recursive mode of thought, without the use of apparent variables ranging over infinite domains. In J. van Heijenoort, editor, *From Frege to Godel*. Harvard University Press, Cambridge, Massachusetts, 1967.

[Sta87] R. M. Stallman. *GNU Emacs Manual*. Free Software Foundation, 1000 Massachusetts Avenue, Cambridge, MA 02138, 1987.

[Ste84] G. L. Steele Jr. *Common Lisp the Language*. Digital Press, 30 North Avenue, Burlington, MA 01803, 1984.

[Vit82] B. L. Di Vito. *Verification of Communications Protocols and Abstract Process Models*. PhD thesis, University of Texas at Austin, 1982.

[Wil91] M. Wilding. Proving Matijasevich's lemma with a default arithmetic strategy. *Journal of Automated Reasoning*, 7(3):439–446, 1991.

[Wil92] M. Wilding. A proved application with simple real-time properties. Technical Report 78, Computational Logic, Inc., 1717 W. Sixth Street, Suite 290, Austin, TX 78703, 1992.

[Wos67] L. Wos et al. The concept of demodulation in theorem proving. *Journal of the ACM*, 14:698–709, 1967.

[You89] W. D. Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5(4):493–518, 1989.

[You90] W. D. Young. A simple expression compiler: A programming and proof example for an Nqthm course. Internal Note 210, Computational Logic, Inc., 1717 W. Sixth Street, Suite 290, Austin, TX 78703, November 1990.

[Yu90] Y. Yu. Computer proofs in group theory. *Journal of Automated Reasoning*, 6(3), 1990.

[Yu92] Y. Yu. *Automated Proofs of Object Code for a Widely Used Microprocessor*. PhD thesis, University of Texas, 1992.