Interaction with the Boyer-Moore Theorem Prover: A Tutorial Study Using the Arithmetic-Geometric Mean Theorem

Matt Kaufmann (CLI) and Paolo Pecchiari (IRST, DIST)

Technical Report 100 August, 1994

Computational Logic, Inc.
1717 West Sixth Street, Suite 290
Austin, Texas 78703-4776

TEL: +1 512 322 9951
FAX: +1 512 322 0656

EMAIL: kaufmann@cli.com and peck@frege.mrg.dist.unige.it

Interaction with the Boyer-Moore Theorem Prover: A Tutorial Study Using the Arithmetic-Geometric Mean Theorem.

Matt Kaufmann
Computational Logic, Inc.
Austin, TX 78703 USA

Paolo Pecchiari
Mechanized Reasoning Group,
IRST - Povo, 38100 Trento, Italy and
DIST - University of Genoa - Genoa, Italy

Keywords: Boyer-Moore theorem prover, Nqthm, automated reasoning, interaction

Abstract

There are many papers describing problems solved using the Boyer-Moore theorem prover, as well as papers describing new tools and functionalities added to it. Unfortunately, so far, there has been no tutorial paper describing typical interactions that a user has with this system when trying to solve a nontrivial problem, including a discussion of issues that arise in these situations. In this paper we aim to fill this gap by illustrating how we have proved an interesting theorem with the Boyer-Moore theorem prover: a formalization of the assertion that the arithmetic mean of a sequence of natural numbers is greater than or equal to their geometric mean. We hope that this report will be of value not only for (non-expert) users of this system, who can learn some approaches (and tricks) to use when proving theorems with it, but also for implementors of automated deduction systems. Perhaps our main point is that, at least in the case of Nqthm, the user can interact with the system without knowing much about how it works inside. This perspective suggests the development of theorem provers that allow interaction that is user oriented and *not* system developer oriented.

1 Introduction

Numerous papers have been written to describe challenge problems solved using the Boyer-Moore theorem prover, Nqthm[1] [3, 5].[2] There are also papers on new tools and functionalities

[1] named after a directory on which the system was originally developed

[2] This system, and its Pc-Nqthm interactive enhancement (described below), are currently available from Internet host ftp.cli.com: directory /pub/nqthm/nqthm-1992/ for the Nqthm-1992 release, and directory /pub/pc-nqthm/pc-nqthm-1992/ for the Pc-Nqthm-1992 release.

added to Nqthm (for example first-order quantification, [17]), and on applications, including an entire issue of the Journal of Automated Reasoning [2]. However, the literature about this system lacks a tutorial paper that focuses on how to use it to solve hard problems and what kind of issues arise in these situations.[3]

In this paper we give a tutorial introduction to Nqthm by illustrating how we have used this system on an example suggested at an inductive theorem-proving workshop [7], the Arithmetic-Geometric Mean Theorem. Informally, the theorem says that the arithmetic mean is greater than equal to the geometric mean for non-empty sequences of nonnegative real numbers:

$$(\sum_{i=1}^{n} x_i)/n \geq (\prod_{i=1}^{n} x_i)^{(1/n)}$$

However, the formalization we have chosen to take is the following, which makes sense over the natural numbers as well:

$$(\sum_{i=1}^{n} x_i)^n \geq n^n * \prod_{i=1}^{n} x_i,$$

where $x_i \in \mathcal{N}$ for $0 < i \leq n$. Notice that this is a fairly interesting theorem, not completely obvious in our view, but we stress that our purpose here is not to provide the simplest proof of it. In fact, there are simpler proofs over the nonnegative real (or rational) numbers, and perhaps over the natural numbers as well (although we are not familiar with any). The purpose of this paper is not presentation of the mathematics *per se*, but rather a description of the process of *mechanization* of that mathematics through Nqthm.

We would like to think that the paper is accessible to anyone who has some experience with mathematics or with any automated reasoning system. For this purpose we have added enough background material to make the document self-contained (there are only few "advanced" details in the presentation that require pointers to other references). Moreover, we have organized the tutorial in a "layered" fashion so that the reader can obtain a good overview without reading the appendices, or a very thorough instruction by reading the paper in its entirety.

We believe that it is important to provide a *user* view of interaction with theorem provers such as Nqthm, for two reasons.

- A user view of a theorem prover is useful, of course, for prospective *users* of this kind of technology.
- A user view of a theorem prover is useful as a guide to *implementors* of theorem provers who are looking for ways to make their systems more practical to use.

In order to present a user's perspective on this system we have tried to give an honest account of what transpired in the sessions we had with Nqthm, including some of our missteps. We do

[3] See however any of [14, 15, 1, 16] for introductions to the use of Nqthm. In fact [16] gives a detailed account of the use of Nqthm and its interactive enhancement to do a non-trivial proof. However, it has a different emphasis than the present paper's tutorial direction; for example, it omits the hints given to Nqthm, which are an important part of the process of interacting with the system. There are other introduction to Nqthm. For example, John Cowles has written a brief introduction in the file examples/cowles/intro-eg.events contained in the Nqthm-1992 release. The paper [9] provides an introduction to Nqthm as part of a verification system.

clean things up just a bit when that significantly improves the exposition, such as introducing the `LET` construct for readability (though we *could* have done so in the first place!). To support our focus on user-level perspective, we do not always give an analysis of what the prover did, at least not to the extent that a researcher in automated reasoning might be tempted to give. (Readers who want more of an implementation-level perspective on Nqthm are referred to [3].) Notice that we *can* get away from the implementor's perspective – that is, we can avoid looking too much inside what the prover does – because Nqthm gives feedback that is meaningful to *users,* **not just to implementors**. The implementor's perspective may be much more common in the automated reasoning literature.

As suggested above, we hope that this work will be valuable both for Nqthm users and implementors of automated deduction systems. Nqthm users will find in this document an extensive panorama, by means of examples, of the approaches and the tricks that can be used to prove theorems with Nqthm. Implementors could find interesting the view of interaction with Nqthm imparted by this "empirical" work, especially as highlighted in Section 3.

1.1 Structure of the paper

In Section 2 we give some background material about Nqthm, its "interactive enhancement" Pc-Nqthm,[4] and the "checkpoint" tool. In Section 3 we make some observations about the kind of interaction that an (expert) user has with Nqthm. Section 4 contains the hand proof that guided our mechanized proof. The rest of the paper is devoted to the description of our mechanized proof, in the spirit (though not quite the letter) of an annotated transcript. As mentioned above, we have structured it in layered fashion. There are three different levels, where each topic in level 1 can have pointers to sections in level 2, and each topic in level 2 can have pointers to sections in level 3. (The result of "flattening" these levels would produce something close to a chronological record of the original session.) The first level, presented in Section 5, contains the most important events of our final script, some motivating organizational comments and enough explanation to give the reader an example of almost every kind of thing we do with Nqthm. The goal of this view of the proof is to give a good introduction to the general idea of how to use Nqthm and Pc-Nqthm. The second level of detail, presented in Appendix A, includes the rest of the events of the final script, and describes in greater detail the proof techniques that are commonly utilized in Nqthm. Anyone who reads this level and really understands it will be quite knowledgeable in the ways of Nqthm! The last level, presented in Appendix B, contains the rest of the mechanized proof, and illustrates some advanced techniques in the use of Nqthm. Anyone who reads this really will be on the way to becoming an expert in Nqthm usage. Appendix C contains a clean list of events, without comments. This list of events has been successfully processed by Nqthm. Finally, Appendix D contains two useful glossaries: one for the Nqthm commands, and one for the built-in Nqthm functions and constants that we have used in the mechanized proof.

[4] The initials "Pc" stand for "Proof-checker", an odd reference to the lower-level nature of the interaction made possible in this system

1.2 Notational conventions

In order to guarantee the tutorial flavor of this paper we have decided to present Nqthm expressions maintaining their Lisp (s-expression) form, so that the user need not guess what we actually did with the system. Text appearing on a line after a semicolon is a comment.

In our presentation, the prover input commands that end up in our final script are displayed in UPPER CASE; they are in lower case when they fail or when they are later undone.

Each time we first introduce an Nqthm command in the main account of the proof (Section 5), we explain the syntax and meaning of that command. As mentioned above, a summary of the commands may be found in Appendix D.

1.3 Acknowledgements

We thank Fausto Giunchiglia and Rich Cohen for comments on drafts of this paper. We also thank Alan Bundy and Deepak Kapur for suggesting that we take on this example.

Mahadevan Subramaniam previously proved a version of the arithmetic-geometric mean theorem using the RRL theorem prover. His success encouraged us to make an attempt.

2 Background on Nqthm, Pc-Nqthm, and Checkpoints

The first and the third subsections below describe the Nqthm logic and theorem prover, respectively. They are taken with permission, with some small changes and additions, from [8], which in turn probably took some of its words from the Nqthm "folklore". For further details see [5]. The words in the Subsection 2.5 are adapted from [12]. See also [6] for more about Nqthm and Pc-Nqthm and about applications of these systems, and for a somewhat simpler example of their use than the one presented here.

2.1 The Nqthm Logic

The logic of Nqthm (sometimes called the "Boyer-Moore Logic") is a simple quantifier-free fragment of first-order logic resembling in syntax and semantics the Lisp programming language. Terms in the logic are written using a prefix syntax, which is case-insensitive –we write `(PLUS I J)` where others might write `PLUS(I,J)` or `i+j`. The logic is formally defined as an extension of propositional calculus with variables, function symbols, and the equality relation. Axioms are added defining the following:

- the Boolean (logical) constants `(TRUE)` and `(FALSE)`, abbreviated `T` and `F`;
- the if-then-else function, `IF`, with the property that `(IF x y z)` is `z` if `x` is `F` and `y` otherwise;
- the Boolean connectives `AND`, `OR`, `NOT`, and `IMPLIES`;
- the equality function `EQUAL`, with the property that `(EQUAL x y)` is `T` or `F` according to whether `x` is `y`;

- and inductively constructed objects including natural numbers and ordered pairs (which also represent lists):
  - Natural numbers, generated by successor function `ADD1`, with corresponding predecessor function `SUB1`, and *base object* `(ZERO)`;
  - Lists and trees, generated by ordered-pair constructor `CONS` and accessors `CAR` and `CDR`, and recognized by the predicate `LISTP`.[5] When such objects are viewed as lists, `CAR` returns the first element of any non-empty list, `CDR` returns the tail of such a list, and `NIL` is an atom typically used to represent the empty list. (These names are borrowed from the family of Lisp languages.)

In addition, there is the ability within the logic to add user-defined inductive data structures, though we do not take advantage of this capability in the arithmetic-geometric mean example.

Nqthm offers a construct, `LET`, for introducing "local variables". The syntax is probably best illustrated by an example, such as the following:

```
(LET ((X (FOO X))
      (Y (G U V)))
     (H X Y U))
```

is an abbreviation for the expression

```
(H (FOO X) (G U V) U)
```

The logic also provides a principle of recursive definition under which new function symbols may be introduced. The following, for example, is a definition within the logic of a list concatenation function `APPEND`.

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

This equation submitted as a definition is accepted as a new axiom under certain conditions that guarantee that one and only one function satisfies the equation. One of these conditions is that certain derived formulas be theorems. Intuitively, these formulas insure that the recursion terminates by exhibiting a "measure" of the arguments that decreases, in a well-founded sense, in each recursive call of the function. Several functions are defined in the initial "ground-zero" theory, including standard recursive definitions of the arithmetic functions `PLUS` and `TIMES` over

[5] Unlike some other Lisp dialects, the predicate `LISTP` in the Nqthm object recognizes ordered pairs and *non-empty* lists, *not* the empty list.

the natural numbers, and the "less-than" relation on the natural numbers, LESSP with its cousins LEQ (less than or equal) and GEQ (greater than or equal).

The rules of inference of the logic, in addition to those of propositional calculus and equality, include mathematical induction. The formulation of the induction principle is similar to that of the definitional principle. To justify an induction scheme it is necessary to prove certain theorems that establish that, under a given measure, the inductive hypotheses are about "smaller" objects than the conclusion. Using induction it is possible to prove such theorems as the associativity of the APPEND function defined above; this can be stated as a theorem in the logic.

```
Theorem ASSOCIATIVITY-OF-APPEND
(EQUAL (APPEND (APPEND A B) C)
       (APPEND A (APPEND B C)))
```

Notice that this theorem provides a partial *specification* of the APPEND function. It is one of myriad properties of this function and its relation to others that can be defined and proved within the logic.

2.2 Brief introduction to conditional rewriting

The theorem above, once proven, may be used in the proofs of subsequent theorems if it is labeled as a *rewrite rule* (syntax presented in the next subsection). That is, we can interpret this theorem as sanctioning the replacement of any expression that matches the left hand side of the equality by the corresponding instance of the right hand side. Thus, for example, the expression

```
(*)        (APPEND (APPEND V (APPEND W X)) (APPEND Y Z))
```

can be readily shown to be equal to

```
(**)       (APPEND V (APPEND W (APPEND X (APPEND Y Z))))
```

by two applications of ASSOCIATIVITY-OF-APPEND interpreted as a rewrite rule. By defining and proving rewrite rules, it is possible to build up a powerful theory for proving interesting facts about a specific domain. In fact, probably the most important activity of most users when interacting with Nqthm is the formulation of rewrite rules, which will be applied automatically during subsequent proof attempts. For example, if the prover sees the term (*) above during a proof, and if the theorem ASSOCIATIVITY-OF-APPEND has already been proved and stored as a rewrite rule, then the prover will replace (*) with (**).

Rewrite rules in Nqthm may be *conditional*; that is, they may have hypotheses. Consider for the sake of example the following somewhat weaker version of the rewrite rule above, which says that the variable A must be an ordered pair (a non-empty list):

```
Theorem ASSOCIATIVITY-OF-APPEND

(LISTP A)
 -->
(EQUAL (APPEND (APPEND A B) C)
       (APPEND A (APPEND B C)))
```

Then to apply this lemma to the same term as above,

```
(APPEND (APPEND V (APPEND W X)) (APPEND Y Z)),
```

the Nqthm rewriter would call itself recursively, attempting to prove

```
(LISTP (APPEND V (APPEND W X))).
```

If that recursive rewrite were to fail, then the rewriter would not apply this lemma to this term.

Next we take a look at the Nqthm theorem prover as a whole, keeping in mind that (conditional) rewriting tends to be the most important part of the activity of this system.

2.3 The Nqthm Theorem Prover

The Nqthm theorem prover is a computer program that takes as input a conjecture formalized as a term in the logic and attempts to prove it by repeatedly transforming and simplifying it. The theorem prover employs the following basic transformations:

- *simplification,* which includes:
  - decision procedures for propositional calculus, equality, and linear arithmetic,
  - conditional rewriting based on axioms, definitions and previously proved lemmas,
  - and automatic application of user-supplied simplification procedures that have been proven correct;
- elimination of calls to certain "destructor" functions in favor of others that are "better" from a proof perspective;
- heuristic use of equality hypotheses;
- generalization by the replacement of terms by variables;
- elimination of apparently irrelevant hypotheses; and
- mathematical induction.

The theorem prover contains many heuristics to control the orchestration of these basic techniques. However, as we have already suggested, it is rarely necessary or even useful for the user to have deep knowledge of such heuristics.

Definitions are submitted to the system using the `DEFN` command. The syntax of definitions in Nqthm is as follows:

```
(DEFN name arguments body)
```

For example, the syntax used to define the `APPEND` function, introduced in Subsection 2.1, is as follows.

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

Later we will see that an additional argument may be supplied, which can be used to "explain" why a given recursively defined function terminates.

The system displays a script of the proof attempt allowing the user to follow the progress of the proof and take steps to abort misdirected proof attempts. From the script it is often apparent to the skilled user how to improve the prover's knowledge base so that a subsequent proof attempt will succeed. The script printed by the prover in discovering the proof of the lemma named `ASSOCIATIVITY-OF-APPEND` is shown in Figure 1. That lemma follows the syntax of theorems in Nqthm:

```
(PROVE-LEMMA name rule-classes formula)
```

where `name` is a name to be associated with the theorem, `formula` is the theorem to be proved, and `rule-classes` indicates how the theorem is to be stored. Typically `rule-classes` is `(REWRITE)`, meaning that the theorem should be stored as a rewrite rule, in the sense described in Subsection 2.2. We will also see on occasion an extra, optional argument, which is a list of hints to the theorem prover.

In a shallow sense, the prover is fully automatic; the system accepts no advice or directives from the user once a proof attempt has started. The only way the user can alter the behavior of the system during a proof attempt is to abort the attempt. However, in a deeper sense, the theorem prover is interactive; the system's behavior is influenced by the database of lemmas that have already been formulated by the user and proved by the system. Each conjecture, once proved, is converted into one or more rules and stored in the prover's database to guide the theorem prover's actions in subsequent proof attempts. Often these are rewrite rules, but there

```
(PROVE-LEMMA ASSOCIATIVITY-OF-APPEND (REWRITE)
       (EQUAL (APPEND (APPEND X Y) Z)
              (APPEND X (APPEND Y Z))))

     Call the conjecture *1.

     Perhaps we can prove it by induction.  Three inductions are
suggested by terms in the conjecture.  They merge into two likely
candidate inductions.  However, only one is unflawed.  We will
induct according to the following scheme:

      (AND (IMPLIES (AND (LISTP X) (p (CDR X) Y Z))
                    (p X Y Z))
           (IMPLIES (NOT (LISTP X)) (p X Y Z))).

Linear arithmetic and the lemma CDR-LESSP can be used to prove
that the measure (COUNT X) decreases according to the well-founded
relation LESSP in each induction step of the scheme.  The above
induction scheme leads to two new goals:

Case 2. (IMPLIES (AND (LISTP X)
                      (EQUAL (APPEND (APPEND (CDR X) Y) Z)
                             (APPEND (CDR X) (APPEND Y Z))))
                 (EQUAL (APPEND (APPEND X Y) Z)
                        (APPEND X (APPEND Y Z)))),

  which simplifies, applying the lemmas CDR-CONS and CAR-CONS, and
  opening up the definition of APPEND, to:

        T.

Case 1. (IMPLIES (NOT (LISTP X))
                 (EQUAL (APPEND (APPEND X Y) Z)
                        (APPEND X (APPEND Y Z)))),

  which simplifies, unfolding the function APPEND, to:

        T.

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

Figure 1: Proof of the lemma ASSOCIATIVITY-OF-APPEND

are other classes of rules as well. A database is thus more than a logical theory; it is a set of rules for proving theorems in the given theory. We consider interaction in Section 3.

Using this approach the Boyer-Moore prover has been used to check the proofs of some quite deep theorems. For example, some theorems from traditional mathematics that have been mechanically checked using the system include: the existence and uniqueness of prime factorizations; Gauss' law of quadratic reciprocity; the Church-Rosser theorem for lambda calculus; and Goedel's incompleteness theorem. Somewhat outside the range of traditional mathematics, the theorem prover has been used to check: the recursive unsolvability of the halting problem for Pure Lisp; the proof of invertibility of a widely used public key encryption algorithm; the correctness of metatheoretic simplifiers for the logic; the correctness of a simple real-time control algorithm; the optimality of a transformation for introducing concurrency into sorting networks; the correctness of an implementation of an algorithm for achieving agreement among concurrently executing processes in the presence of faults; correctness of compilers; and correctness properties of a microprocessor that has been fabricated. When connected to a specialized front-end for Fortran, the system has also proved the correctness of Fortran implementations of a fast string searching algorithm and a linear time majority vote algorithm. Many other interesting theorems have been proven as well; see [5] for a rather lengthy list of applications of the system (with references), also see [12].

We conclude our description of the Nqthm prover by pointing out that at a coarse level, interaction with Nqthm proceeds at the level of *events*. An event is the introduction of a definition (such as `APPEND`), the statement of a theorem (such as the `PROVE-LEMMA` form above), or any of a few other *event forms*. The user's goal is to formulate appropriate definitions and theorems and create a file of events that includes not only those, but also any additional events necessary to support the proofs for the desired events. At all points during a session, the Nqthm database contains a stack of events that have already been processed; each time it successfully processes a new event, it pushes the new event on the stack, which is called a *history*. There are a few other commands that are useful besides events. For example, the command `(UBT)`, "undo-back-through", pops an event off the history stack; when given a name as an argument, it pops all events back through that name, inclusive. Other Nqthm commands that we use are explained below at the first time they are used; see also Appendix D.

2.4 Introduction to Induction Schemes in Nqthm

We turn now to a very informal introduction to the relation in Nqthm between recursive function definitions and induction schemes. By adopting a very informal style, we hope to make the presentation below self-contained for all but the most extreme logical purists (who probably would be able to work out details of the Nqthm logic from this informal description, anyhow!).

In traditional proofs by (strong) mathematical induction, or more precisely *Noetherian* induction, if one wants to prove a proposition of the form $\forall \vec{x} P(\vec{x})$, one may assume $P(\vec{y})$ for all $\vec{y}$ that are "smaller" than $\vec{x}$ in an appropriate sense. That is: it suffices to prove a statement of the form "(for all $\vec{y}$ smaller than $\vec{x}$, $P(\vec{y})$) **implies** (for all $\vec{x}$, $P(\vec{x})$)". However, the Nqthm logic does not have quantifiers; in particular, one can not express the preceding statement in that logic. Instead, that logic has an induction principle (stated precisely in Chapter 4 of [5]) that

serves much the same purpose as the Noetherian induction principle stated informally above. Here is an overview that should suffice for the purposes of this paper; for more details, see [5].

The Nqthm idea of inductive proof is based on the idea of breaking a proof into cases, and then assuming instances of the theorem as inductive hypotheses in a given case, as long as these instances replace variables with terms that are appropriately "smaller" under the assumption of that case. Significantly, this decomposition into cases, each with zero or more induction hypotheses, corresponds to the structure of recursive definitions in Nqthm. Let us illustrate this correspondence by way of a simple example: a definition of a function that returns the length of a list, which we will need later anyhow.

```
(DEFN LENGTH (X)
  (IF (LISTP X)
      (ADD1 (LENGTH (CDR X)))
    0))
```

Suppose now that we want to prove a proposition `p(X)`. Let us break our goal into two cases according to the `IF` structure in the definition above.

Case 1: `(LISTP X)`. Then (if we believe that the definition of `LENGTH` terminates), it "should" be the case that the `(CDR X)` is "smaller" than `X` in this case, in some appropriate sense. So when proving `p(X)`, we "may" assume the inductive hypothesis, which is `p((CDR X))` in this case, in addition to the case hypothesis `(LISTP X)`.

Case 2: otherwise. Then since there are no recursive calls of `LENGTH` in this "branch" of the `IF` structure of the body of the definition, we make no inductive hypothesis in this case. After all, this case is what we would normally call a "base case".

Now *suppose* that we want to prove a proposition by the induction argument above, where we break into cases `(LISTP X)` and `(NOT (LISTP X))`, and where we assume in the former case that the proposition holds with `(CDR X)` replacing `X`. *Then* we can imagine simply saying: "Use the induction scheme suggested by the definition of `LENGTH`." And, in fact, if we were to give to Nqthm the hint

```
(induct (length x))
```

to a `PROVE-LEMMA` event, we would be telling the theorem prover to prove the indicated theorem by just that induction argument. We can display the structure of that induction argument as follows; see also [3] for a more thorough presentation of this idea.

```
Case 1:  (LISTP X)
  For the inductive hypothesis:  replace X by (CDR X)

Case 2:  (NOT (LISTP X))
  No inductive hypothesis.
```

Note that in order to believe in the soundness of this induction scheme, we need to know that `(CDR X)` is "smaller" than `X` when `(LISTP X)` holds, where "smaller" is in the sense of a well-founded relation. In fact there is a function `COUNT` with the property that in the base Nqthm theory, it is a theorem that `(LISTP X)` implies that `(COUNT (CDR X))` is less than `(COUNT X)` in the usual ordering of the natural numbers, which is well-founded.

2.5 The Pc-Nqthm Interactive Enhancement

Pc-Nqthm ("Proof-checker Nqthm") is an interactive enhancement of Nqthm. The system is described in detail in [10], [11], and [17]; here is a brief introduction.

The user enters an interactive loop with the command `VERIFY`. Consider the example of the associativity of append from the preceding subsection:

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

The system replies with a prompt, "`->:`", indicating that one is now in an interactive loop. In this loop, the 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 Nqthm prover). For example, the `INDUCT` command replaces the current goal by subgoals corresponding to the base and induction steps. In the following transcript, everything except the prompt and "`induct`" was printed by the system.

```
->: induct

Inducting according to the scheme:

      (AND (IMPLIES (AND (LISTP X) (p (CDR X) Y Z))
                    (p X Y Z))
           (IMPLIES (NOT (LISTP X)) (p X Y Z)))

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

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

Commands also exist for displaying useful information (such as printing the current hypotheses and conclusion, displaying the currently applicable rewrite rules, or showing the current abbreviations) and for controlling the progress of the proof (such as undoing a specified number of

commands, changing goals, or disabling certain rewrite rules). A notion of "macro commands" lets the user create compound commands, roughly in the spirit of the tactics and tacticals of LCF and its descendents. An on-line help facility is provided.

As with a variety of proof-checking systems, this system is goal-directed: a proof is complete when the main goal and all subgoals have been proved. Upon completion of an interactive proof, the lemma may be added to the Nqthm history. This event can later be replayed in "batch mode". Partial proofs can also be stored.

Pc-Nqthm has been used to prove theorems stating the correctness of a transitive closure program, a Towers of Hanoi program, a ground resolution prover, a compiler, irrationality of the square root of 2, an algorithm of Gries for finding the largest "true square" submatrix of a Boolean matrix, the exponent two version of Ramsey's Theorem, the Shroeder-Bernstein theorem, Koenig's tree lemma, and others. It has also been used to check the correctness of several Unity programs and has been used for hardware verification.

2.6 Checkpoints

Intensive users of theorem provers often rely on feedback from the system they use when a proof attempt, or proof step, fails to achieve the desired effect. Nqthm users are no exception, and those who are successful tend to be accomplished at making sense out of the theorem prover's output. They seem to have a knack for looking at the output from a failed proof attempt and quickly deciding on an approach that seems promising for guiding the theorem prover to success. How do they do it?

The main trick is to focus on goals that are stable under simplification, i.e., are left unchanged by the Nqthm simplifier, which is described briefly near the beginning of Subsection 2.3. For each goal that Nqthm prints to the terminal, it first attempts to *simplify* that goal before trying any of the other tricks listed in Subsection 2.3. Thus, when the theorem prover applies any techniques outside the realm of simplification to a goal, it means that simplification failed to have any effect on that goal. It follows that the goals stable under simplification are very closely related to the "checkpoints" discussed in Chapter 9 of [5]. Intuitively, a *checkpoint* is a point in an Nqthm proof at which some "daring" transition happens, for example one that can turn a provable goal into one that is not provable (such as generalization). That is, a checkpoint is a point in the proof that may well bear some careful inspection by the user. At any rate, every checkpoint is a goal that is stable under simplification.

Why is it so useful to find goals that are stable under simplification? Very often it is the case that these goals are "obvious" to the user, and will become "obvious" to the theorem prover as well if only a useful rewrite rule can be proved. For example, a goal such as

```
(implies (and (p x) (p y))
         (equal (foo (append (append x y) (h x y)))
                (foo (append x (append y (h x y))))))
```

seems obvious because `APPEND` is associative. If this goal is stable under simplification, then apparently the theorem prover simply does not yet "know" that `APPEND` is associative. The

next step, then, would be to prove such a lemma as a *rewrite rule*, so that it can be used automatically by the simplifier. Such a rule is proved in Subsection 2.3. A new proof attempt could then proceed past the point of the goal displayed above.

During the present proof effort, we employed a tool that we refer to in this paper as the "checkpoint tool". This tool, which is documented in [13][6], allows the user who is running Nqthm under Gnu emacs [18] to get the cursor to move quickly to a checkpoint, which will be a goal that is stable under simplification. It actually allows one to visit additional checkpoints as well. Our main use of that additional capability took place when the Nqthm prover decided to throw away what it had done and start a proof over, using induction on the original goal. In such cases, the checkpoint tool will beep and offer the user the option of visiting a checkpoint that appears in the output after the proof has started over. It is often the case in such situations that the user already expected the proof to proceed by induction, and is therefore not interested in the part of the proof that was "thrown away"; so, the user types `<return>` and the checkpoint tool moves to the later checkpoint.

3 Interacting with Nqthm

In this section we give a high-level description of (our view of) Nqthm interaction. These points are exemplified in detail, many times over, in Section 5 and in the appendices.

In our view, there is no deep mystery to typical interaction with Nqthm. The user formulates theorems and supporting definitions, then invents a main strategy, and then proves theorems and modifies definitions by inspecting goals from failed proofs that are left unchanged by simplification. (See the description of checkpoints in Subsection 2.6.)

Somewhat more precisely, we can say that a "low" (non-strategic) level, the idea is to use Nqthm primarily as a simplifier, though allowing some use of induction and other proof techniques, to see where the simplification "stalls". This is very different from our usual informal style of mathematical reasoning, where we try to stay at the "strategic" level rather than thinking in terms of "canonicalization". Sometimes it is best to think at a high level but to interact at that sort of lower level – when trying to wrestle with the prover, it's best not to think strategically, but instead, to think in terms of helping the simplifier by proving useful rewrite rules so that it can simplify further than it did before.

To summarize the two main components of Nqthm interaction:

- Decompose the entire theorem into lemmas that yield it immediately, even before descending into the theorem prover. This decomposition isn't necessarily an obvious syntactic one, as we explain below.
- Submit a given theorem to the prover, inspect goals that are stable under simplification (perhaps using the checkpoint tool; see Subsection 2.6), and then prove lemmas (especially, rewrite rules) to help the prover with those failed goals.

In addition (fortunately or unfortunately), sometimes we find it helpful to employ subtle "advanced" techniques in our user interaction. For example, we occasionally use the interactive

[6] and is available with the Pc-Nqthm-1992 distribution; see misc/checkpoints.doc inside that distribution

facility provided by Pc-Nqthm (see Subsection 2.5) for "debugging", and sometimes we look at the *rewrite stack* (see for example page 29). And in fact, sometimes we test our functions using the Nqthm execution tool, `R-LOOP` (see for example page 31), perhaps even running tests that show us that our "theorems" are not theorems after all!

Returning to the technique of providing high-level structure, sometimes it is very useful to think about the underlying mathematics when decomposing theorems into lemmas, in a manner typical of doing mathematics. For example, we proved $MIN + 1 < MAX$ (see Subsection 5.6) by thinking about how we would reason informally, namely, $MIN < AVERAGE < MAX$. Then, our lemma structure reflected this informal reasoning.

In conclusion, our view is that support for interaction with Nqthm is pretty good already, *if* you know what you are doing (for example, if you absorb this paper). It could presumably be improved, but the improvements are not necessarily in the organization of the prover's processes or heuristics. So we pose the following question.

BIG QUESTION:
How could interaction with Nqthm (and similar theorem provers) be improved?

Of course there are no definitive answers to this question. But we believe that in order to make theorem provers useful to an audience wider than the automated reasoning research community, it is important to avoid intertwining interaction and internal prover control.[7] Although the Nqthm user will find it useful to adopt a certain paradigm of interaction, nevertheless we do not think that deep understanding of the theorem prover's internal workings are particularly useful; in fact, they probably confuse the matter more than help. Unfortunately there are occasional tricky points; for example, in the course of the proof below the prover (more precisely, the linear arithmetic decider) is not able to infer that MAX is a natural number from the fact that $MIN < MAX$ (see page 45). So maybe the answer to the **BIG QUESTION** above is: improve the prover to get rid of such rough edges, so that the user is indeed shielded from low-level prover implementation issues. Notice that a *user* is not necessarily an *implementor*! These two classes of individuals have different concerns. So for example, if the prover is not behaving as expected, we take the position that a reasonable user response is "How do I overcome or work around this problem?", while the implementor's response is more likely something like "Why didn't the prover do a better job here?" In other words, we are taking the position that the *user* would prefer not to be bothered with such "why" questions. We will demonstrate below that, in the case of Nqthm, one can actually profit from such an attitude.

4 The hand proof

In this section we give a short hand proof of a formalization of the arithmetic-geometric mean theorem from Section 1. It is easy to imagine a much more detailed hand proof; for example, several observations require the use of mathematical induction for a rigorous proof. Thus, the contrast in length between this hand proof and our mechanized proof is due not only to the cost

[7] Fausto Giunchiglia has made this observation to us. He also pointed out that the central role of interaction in theorem proving was initially advocated by McCarthy and Weyhrauch.

of mechanization, but can be viewed in part as the cost of providing "reasonable" details. That said, we choose to present a short proof in this section, omitting details that might distract one from the main ideas.

In the following, $\vec{x} = \langle x_1, \ldots, x_n \rangle$ denotes a non-empty sequence of natural numbers. Moreover we assume that max and min are functions returning respectively the maximum and the minimum numbers contained in a sequence of natural numbers.

Lemma 1. If $max(\vec{x}) \neq min(\vec{x})$ and there exists a natural number k such that

$$\sum_{i=1}^{n} x_i = n * k$$

then

$$min(\vec{x}) + 1 < max(\vec{x}).$$

Proof. Notice that the hypothesis implies that k is the average of the elements contained in $\vec{x}$. Therefore, from the assumption that the "maximum" and the "minimum" of $\vec{x}$ are different we have that

$$min(\vec{x}) < k < max(\vec{x}).$$

$\triangle$

The following lemma may be viewed as a special case of our goal, where the average is an integer.

Lemma 2. If there exists a natural number k such that

$$\sum_{i=1}^{n} x_i = n * k$$

then

$$k^n \geq \prod_{i=1}^{n} x_i.$$

Proof. By well-founded induction on the lexicographic relation $\prec$ defined as follows

$$\vec{x} \prec \vec{y} \equiv (max(\vec{x}) < max(\vec{y}) \vee (max(\vec{x}) = max(\vec{y}) \wedge occ(max(\vec{x}), \vec{x}) < occ(max(\vec{y}), \vec{y}))),$$

where $occ(a, x)$ equals the number of occurrences of a in the sequence x. It can be easily proved that $\prec$ is well-founded.

We can split the proof in two cases. Very informally, the idea in the "inductive step" (Case 2), where the maximum and minimum are distinct, is to decrease the maximum by 1 and increase the minimum by 1, which does not change the average of the list but can only increase its product (hence, its geometric mean) – and then, the inductive hypothesis applies.

Case 1: $max(\vec{x}) = min(\vec{x})$. It follows that $k = x_i$ for each i, $1 \leq i \leq n$, and hence

$$k^n = \prod_{i=1}^{n} x_i.$$

Case 2: $max(\vec{x}) \neq min(\vec{x})$. Then we may choose distinct k and j such that x_k, x_j are respectively the maximum and the minimum of $\vec{x}$. Consider the sequence $\vec{y} = \langle y_1, \ldots, y_n \rangle$ obtained from $\vec{x}$ by replacing x_k with $x_k - 1$ and x_j with $x_j + 1$. Note that the sums of the elements respectively in $\vec{x}$ and $\vec{y}$ are equal. Moreover, we have:

$$\prod_{i=1}^{n} y_i > \prod_{i=1}^{n} x_i, \tag{1}$$

as from Lemma 1 and simple algebra we can easily prove that $(x_k - 1) * (x_j + 1) > x_k * x_j$. But now the conclusion follows from (1) together with the inductive hypothesis

$$k^n \geq \prod_{i=1}^{n} y_i$$

– once we show that the inductive hypothesis is applicable. That is, it remains to prove $\vec{y} \prec \vec{x}$, which we do in the following two cases.

Case 2.1: $occ(max(\vec{x}), \vec{x})$=1. From Lemma 1 it follows that $max(\vec{y}) < max(\vec{x})$. Thus $\vec{y} \prec \vec{x}$.

Case 2.2: $occ(max(\vec{x}), \vec{x}) > 1$. In this case the maxima of $\vec{y}$ and $\vec{x}$ are equal and from Lemma 1 we have that $occ(max(\vec{y}), \vec{y}) < occ(max(\vec{x}), \vec{x})$. It follows that $\vec{y} \prec \vec{x}$.

△

Theorem 1.

$$(\sum_{i=1}^{n} x_i)^n \geq n^n * \prod_{i=1}^{n} x_i.$$

Proof. Let $y_i = n * x_i$ for each i, $1 < i \leq n$. We have that

$$(\sum_{i=1}^{n} y_i) = n * (\sum_{i=1}^{n} x_i). \tag{2}$$

By applying the previous lemma (with $k = \sum_{i=1}^{n} x_i$ and replacing $\vec{x}$ by $\vec{y}$), we obtain

$$(\sum_{i=1}^{n} x_i)^n \geq \prod_{i=1}^{n} y_i = \prod_{i=1}^{n} (n * x_i) = n^n * \prod_{i=1}^{n} x_i. \tag{3}$$

△

5 A high-level view of the mechanized proof

In order to make our presentation of the main body of the mechanized proof more readable we have divided it into subsections. Our presentation roughly parallels the hand proof in Section 4.

- In Subsection 5.1 we start the Nqthm session defining the sum and the product of a sequence (actually a list) of numbers and we state the theorem to prove.

- In Subsection 5.2 we prove the theorem assuming as an axiom an Nqthm term, that we call the main lemma, corresponding to Lemma 2 in the hand proof.
- At this point we turn to the task of proving the main lemma by induction. Subsection 5.3 contains the formalization of the induction argument by way of an Nqthm definition.
- In Subsection 5.4 we describe our first attempts to admit this definition. These attempts culminate in the discovery of a bug in our first definition of a function that deletes one occurrence of an element from a list. Here we don't prove any lemma that is relevant for the final script. Nevertheless we take this digression, not only because it faithfully represents the order in which we carried out the various parts of the proof, but more importantly, because it illustrates a combination of important techniques in the use of Nqthm.
- In Subsection 5.5 we outline the termination proof for the definition that formalizes the induction argument, splitting into two cases.
- The next thing we do, in Subsection 5.6, is to take a break from the main flow of the proof, and prove a version of Lemma 1 from the hand proof. We really did take this break; in fact, this kind of "jumping around" is not atypical of user interaction with Nqthm.
- Next, in Subsection 5.7, we prove one of the two main cases for termination, corresponding to Case 2.2 from the hand proof in Section 4, followed by...
- ... the proof of the other case for termination, Case 2.1 from the hand proof, in Subsection 5.8.
- Finally, we conclude in Subsection 5.9 that the definition we have been considering, which formalizes the induction argument, actually terminates.
- Subsection 5.10 contains the "rest" of the proof, i.e., we prove the main lemma and we use it to prove the desired theorem as done in Subsection 5.2. The proof of the main lemma is divided in two parts: a "base" case in which we assume that the maximum and the minimum of the sequence are equal, and an "induction step" in which we assume that they are different. They correspond to Case 1 and Case 2 in the hand proof of Lemma 2.
- Subsection 5.11 contains a more elegant form of the theorem obtained using the Nqthm iteration construct `FOR`.

5.1 Formalizing the problem

We believe that it's reasonable to build on existing work when trying to do serious proof development. We therefore begin by taking advantage of an existing library including definitions and proved theorems about natural numbers that others have found useful.[8] The `NOTE-LIB` command instructs Nqthm to include all the facts in this library; the optional second argument

[8] In fact, this library is available in the Nqthm-1992 release, in the file examples/numbers/naturals.events, which has many contributors, including William Bevier, Bob Boyer, Matt Kaufmann, J Moore, and Matt Wilding.

T instructs Nqthm to load a corresponding compiled file, which can be useful in the execution of Lisp functions that correspond to Nqthm definitions.

```
(NOTE-LIB "naturals" T)
```

Perhaps we should add that the first form in the "naturals" library is (NOTE-LIB "bags" T); that is, the "naturals" library includes a library of facts and definitions about bags (multisets).

Let us now define a function that returns the product of the members of a given list. (See page 10 for an introduction to Nqthm definitions.) The following recursive definition says that the product of the members of a non-empty list is the result of multiplying the first member (its CAR) by the product of the rest of its elements (the CDR); and, the product of the empty list is 1.

```
(DEFN PRODLIST (LST)
  (IF (LISTP LST)
      (TIMES (CAR LST)
             (PRODLIST (CDR LST)))
    1))
```

We define analogously a function that returns the sum of the members of a list.

```
(DEFN SUMLIST (LST)
  (IF (LISTP LST)
      (PLUS (CAR LST)
            (SUMLIST (CDR LST)))
    0))
```

Is it really necessary to write such definitions recursively? In Nqthm there is a "quantifier" construct, FOR, which is actually an iterator that can be useful in defining functions such as the PRODLIST and SUMLIST above. However, our experience is that the meta-level machinery needed to support reasoning about FOR is awkward; it seems much simpler to quickly type in a primitive recursive definition, as we do here. (However, see also Section 5.11.)

Let us state what we are trying to prove. Recall our convention from Subsection 1.2: since the following event is displayed in lower case, Nqthm must not yet be able to prove this theorem without further help. We will provide such help in subsequent subsections. The syntax of PROVE-LEMMA events is explained in Subsection 2.3. In particular, the notation "()" following "main" below indicates that we do not intend to make a (rewrite or any other) rule out of this theorem. Also, here EXP is the exponentiation function for natural numbers, which is defined in the "naturals" library that we started with. Indeed, Nqthm employs a logic of total functions,

and hence GEQ must "make sense" on arbitrary Nqthm objects (even Booleans and lists), not just numbers. The Nqthm arithmetic primitives all have the property that they treat values that are not natural numbers exactly as though they were 0. Now, finally, here is a formalization of what we are trying to prove in this tutorial.

```
(prove-lemma main ()
             (let ((n (length a)))
               (geq (exp (sumlist a) n)
                    (times (exp n n) (prodlist a)))))
```

Aside. Notice that this theorem even holds when A is the empty sequence, because in that case the theorem reduces to (GEQ (EXP 0 0) (TIMES (EXP 0 0) 1)), regardless of the value of (EXP 0 0). Remember: Nqthm is a logic of total functions, so (EXP 0 0) must have *some* value.

5.2 A top-down approach

Sometimes it is useful to proceed in a top-down fashion. Thus, let us assume the following "main lemma" as an axiom for now; it corresponds to Lemma 2 in the hand proof of the theorem. Then, we will try to derive the theorem from it. Once we succeed in doing so, we will back up and formulate events that culminate in the proof of the main lemma; and *then*, we will paste down the events that we constructed to prove the theorem from the main lemma.

Here, then, is our main lemma. We choose not to label this as a rewrite rule; we will see below how to explicitly direct the use of a specified instance of this lemma when proving the main theorem. The syntax of ADD-AXIOM events is analogous to the syntax of PROVE-LEMMA events.

```
(add-axiom main-lemma ()
           (let ((n (length x)))
             (implies (equal (sumlist x) (times k n))
                      (not (lessp (exp k n) (prodlist x))))))
```

From the hand proof of the theorem, we can see how this lemma proves MAIN, using two trivial facts about scalar product, used informally in equations 2 and 3 in Section 4:

```
(defn scalar-product (scalar lst)
  (if (listp lst)
      (cons (times scalar (car lst))
            (scalar-product scalar (cdr lst)))
    nil))
```

```
(prove-lemma sumlist-scalar-product (rewrite)
             (equal (sumlist (scalar-product scalar lst))
                    (times scalar (sumlist lst))))

(prove-lemma prodlist-scalar-product (rewrite)
             (equal (prodlist (scalar-product scalar lst))
                    (times (exp scalar (length lst))
                           (prodlist lst))))
```

These prove easily. Now we try to prove `MAIN` from the lemmas above, and we make a number of mistakes. See Point 1 in Appendix A for an explanation of how we use the theorem prover to help us correct those mistakes and formulate an additional lemma.

The resulting formulation of `MAIN` is now as follows.

```
(prove-lemma main ()
             (let ((n (length a)))
               (geq (exp (sumlist a) n)
                    (times (exp n n) (prodlist a))))
             ((use (main-lemma (k (sumlist a))
                               (x (scalar-product (length a) a))))))
```

The last two lines above contain an example of a `USE` hint. They tell the prover that instead of proving the theorem at hand – let us call it `p` for the moment – the prover should instead prove `(IMPLIES hyp p)`, where `hyp` is the result of applying the indicated substitution,

```
{k <- (sumlist a),
 x <- (scalar-product (length a) a)},
```

to the previously-proved theorem called `MAIN-LEMMA`.

The proof succeeds. So, let's undo the events from this Subsection using the `UBT` command (see page 12) and go back and prove the lemma.

```
(ubt main-lemma)
```

Where are we? Nqthm provides a command for seeing the names of any number of the most recent events in the *history* (see page 12): `CH`. The syntax is

```
(CH n)
```

where n is a natural number and the result is to print out the names of the most recent n events of the history.[9] Now: if we execute (CH 6), say, we can see that the most recent events are the definitions of PRODLIST, SUMLIST, and LENGTH, preceded by events from our initial NOTE-LIB command. And, that is where we want to be.

5.3 Formalizing the induction scheme

Our goal now is to prove the main lemma, so we need to formalize the induction scheme that we have written on paper. Before we begin to do so, we need to define a few functions.

The function MAX happens to be built in to Nqthm (and shortly we will display its definition); it returns the maximum of two natural numbers, where arguments that are not natural numbers are treated as zero. In the definition below, for the case that X has only one element, we apply the built-in function FIX, where (FIX X) is equal to X unless X is not a natural number, in which case (FIX X) is 0. Originally we forgot to wrap the call of FIX around (CAR X) in this definition, but during a proof that came later, we found using the checkpoint tool that we needed to prove (NUMBERP (MAXLIST X)) and could not do so. We then inspected the definition of MAXLIST and realized that we had to apply FIX as shown.

```
(DEFN MAXLIST (X)
  (IF (LISTP X)
      (IF (LISTP (CDR X))
          ;; So, X has at least two elements.
          (MAX (CAR X) (MAXLIST (CDR X)))
        ;; Otherwise, X has only one element.
        (FIX (CAR X)))
    ;; We "shouldn't" really reach the case where X is empty, since we
    ;; terminate above when X has one element.  But anyhow, to complete
    ;; the definition in a way that appears reasonable somehow:
    0))
```

MAX is indeed built in, as we see now using the PPE command ("Pretty-Print Event"), which causes the theorem prover to print out the requested event. The last part of the system's response means that MAX is built-in.

```
>(ppe 'max)  ; This line is the input; system output follows.
```

[9] Actually n is optional; when it is omitted, all names from the history are printed out.

```
(DEFN MAX
      (X Y)
      (IF (LESSP X Y) Y (FIX X)))

(***** MAX is a satellite of
       (BOOT-STRAP NQTHM))
```

It turns out that we need an analogous notion of minimum, but `MIN` is not built in. So, let us define `MIN` analogously. Note that we expect that, for example (treating non-numbers again as `0`),

`(MIN T 3) = (MIN 3 T) = 0,`

so we shouldn't be surprised at two uses of `FIX` below.

```
(DEFN MIN (X Y)
  (IF (LESSP X Y) (FIX X) (FIX Y)))
```

And now we can use `MIN` to define a function returning the minimum of a list.

```
(DEFN MINLIST (X)
  (IF (LISTP X)
      (IF (LISTP (CDR X))
          (MIN (CAR X) (MINLIST (CDR X)))
        (FIX (CAR X)))
    0))
```

We want to make the following definition, but in fact (modulo variable names) it turns out to be part of the `"naturals"` library that we included originally (or more precisely, part of the `"bags"` library that is included in the `"naturals"` library).

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

Our goal now is to prove the main lemma. We want to use the sort of induction argument described in Subsection 2.4, although the particular induction scheme differs from the one provided by LENGTH. Recall from our hand proof (Section 4) that the idea is to have two cases, according to whether or not the maximum and minimum of the list are equal. And, in the case where they are equal, we want to assume as an inductive hypothesis the claim that the theorem holds when we replace the list by a new list in which the maximum has been decreased by 1 and the minimum has been increased by 1. Let us call that new list X0, just for the time being.

```
CASE 1:  (EQUAL (MINLIST X) (MAXLIST X))
  This is a ``base case'':  we have no inductive hypothesis for this case.

CASE 2:  (NOT (EQUAL (MINLIST X) (MAXLIST X)))
  In this case, we would like to assume as an inductive hypothesis that
  the theorem holds with X replaced by the list X0 described above.
```

Recall, however, that we need to know that this induction scheme is "sound". Just as Nqthm must accept the definition of LENGTH before it uses the induction scheme corresponding to that definition (see Subsection 2.4), Nqthm must accept an appropriate definition before we can use the induction scheme that we are currently developing. The Nqthm principle of definition requires that X0 is "smaller than" X in an appropriate sense, assuming (NOT (EQUAL (MINLIST X) (MAXLIST X))). In fact, if we look back at the hand proof, we see that we had some additional information available, namely, that (SUMLIST X) is equal to (TIMES K (LENGTH X)). So, let us revise our induction scheme accordingly.

```
CASE 0:  (NOT (EQUAL (SUMLIST X) (TIMES K (LENGTH X))))
  This is a ``base case''; in fact, the main lemma has the negation of this
  case as a hypothesis, so this case will be trivial.

CASE 1:  (EQUAL (MINLIST X) (MAXLIST X))
         and also (EQUAL (SUMLIST X) (TIMES K (LENGTH X)))
  This is a ``base case'':  we have no inductive hypothesis for this case.

CASE 2:  (NOT (EQUAL (MINLIST X) (MAXLIST X)))
         and also (EQUAL (SUMLIST X) (TIMES K (LENGTH X)))
  In this case, we would like to assume as an inductive hypothesis that
  the theorem holds with X replaced by the list X0 described above.
```

Here, then, is an an Nqthm definition whose induction scheme matches the display above, in the sense discussed in Subsection 2.4. Later we will write a suitable expression in place of the mysterious abbreviation X0 introduced above. Note that we may use any expressions in place of

the occurrences of T below, as long as they do not contain a call of INDUCTION-FN, because we are only using this definition to generate the induction argument displayed above.

```
(defn induction-fn (x k)
  (if (not (equal (sumlist x) (times k (length x))))
      t
    (if (equal (minlist x) (maxlist x))
        t
      (induction-fn X0 k))))
```

Let us turn next to the problem of writing a suitable expression in place of X0 above. That is, we wish to formalize the idea of moving the maximum and minimum towards each other by 1 unit, as described in the informal proof, Section 4. Our intuition is that it will be easier to reason about this notion if we can factor the replacement operation into two apparently simpler operations: deleting an element from a list and CONSing a new element on to the front. That thinking gives rise to the following definition, which fleshes out the template above. It uses a function DELETE1 that deletes one occurrence of an element from a list. We will say more about DELETE1 below, and we will explain the last line of the definition below as well.

```
(defn induction-fn (x k)
  (if (not (equal (sumlist x) (times k (length x))))
      t
    (if (equal (minlist x) (maxlist x))
        t
      (induction-fn (cons (sub1 (maxlist x))
                          (cons (add1 (minlist x))
                                (delete1 (maxlist x)
                                         (delete1 (minlist x) x))))
                    k)))
  ((ord-lessp (cons (add1 (maxlist x))
                    (occurrences (maxlist x) x)))))
```

Alternatively, we could have written recursive functions that look for an element and then decrement (or increment) it "in place"; that might well work out fine, but our guess is that our tasks will be simpler if we do the "factoring" indicated above.

The last line of the definition above contains a *measure hint*, which is intended to suggest to Nqthm a reason why it should accept this definition (informally, why the definition terminates). This hint reflects the idea from the hand proof that X0 is "smaller than" X in the sense of the usual lexicographic order on pairs of natural numbers. Actually this order is represented in the Nqthm logic by ORD-LESSP, an order on notations for ordinals less than ϵ_0, see [5]) applied

to the pair consisting first of the maximum of X, and then of the number of occurrences of its maximum in X. See Point 2 in Appendix A for more details about this point.

The definition above seems to take forever to process, so we interrupt it.[10] It isn't surprising that this definition presents a challenge, since the heart of the entire proof is to prove termination of this function. So, let us focus now on that task.

5.4 First attempts at the termination proof

In this Subsection we describe our first attempts to admit the definition of INDUCTION-FN. These attempts culminate in the discovery of a bug in our first definition of the function DELETE1. As mentioned before, we take this digression from the main flow of the proof for pedagogical reasons, as we have found this bug by using a combination of some important techniques and commands of Nqthm.

Here is our first attempt to define a function DELETE1 that deletes one occurrence of an element from a list.

```
(defn delete1 (elt x)
  (if (listp x)
      (if (equal elt (car x))
          ;; Since we are only deleting one occurrence, we do not recur.
          (cdr x)
        (delete1 elt (cdr x)))
    ;; Otherwise x is empty; return either x or nil, it doesn't matter.
    x))
```

It's not difficult to find a bug in this definition once you are told that there is one; but in our actual session with Nqthm, we are working quickly and not paying much attention!

Our initial attempts at admitting INDUCTION-FN are quite fruitless. In lieu of any better ideas at the moment[11], we execute the Nqthm command

```
(maintain-rewrite-path t)
```

which tells the prover to monitor the rewriting process. We will re-submit the definition of INDUCTION-FN, then we will abort the processing of this definition so that we can inspect the *rewrite path* especially to look for looping rewrite rules.

Submitting the definition again causes an error, because you have to execute (UBT) after aborting a DEFN. This is an obnoxious feature of Nqthm, and is the sort of thing that we believe

[10] In our particular environment, this amounts to issuing two control-c characters.

[11] The "moment" is actually the next day. After logging back in, we used the Nqthm utility DO-FILE to process the file of events created so far. A benefit of such breaks is that DO-FILE will fail if we have not put the appropriate events in the file we are constructing, and we would rather find that out now and fix it, rather than waiting until the very end of the proof!

should be minimized in serious user-friendly proof assistants. Fortunately, Nqthm contains very few such idiosyncrasies.

After a while we interrupt the `DEFN` event and submit the form

```
(break-rewrite)
```

which enters the Nqthm tool for inspecting the *rewrite stack*. This notion is explained amply in [5]; here we restrict our explanation to the particular commands that we invoke within this utility.

The command `path`, in response to the prompt ":", yields the following output:

```
: path

*(4482)   0. (top)
 ( 127)   1. Rewriting (EQUAL ...)
 ( 126)   2. Rewriting (ADD1 ...)
 ( 125)   3. Rewriting (MINLIST ...)
 ( 123)   4. Rewriting (IF ...)
 ( 120)   5. Rewriting (IF ...)
 ( 113)   6. Rewriting (MIN ...)
 ( 101)   7. Rewriting (IF ...)
 ( 100)   8. Rewriting (LESSP ...)
 (  71)   9. Rewriting (IF ...)
 (  40)  10. Rewriting (IF ...)
*(  38)  11. Rewriting (IF ...)
 (   7)  12. Rewriting (IF ...)
 (   4)  13. Rewriting (LESSP ...)
 (   3)  14. Rewriting (SUB1 ...)
:
```

Inspection of the next-to-top frame of the rewrite stack yields the following response from the system.

```
---- Frame 1 -----           (persistence 127)
Rewriting the conclusion of the top-level goal:
(EQUAL (ADD1 (MINLIST X)) (MAXLIST X))
under the substitution:
NIL
:
```

This baffles us: why is the prover trying to prove this equality?

Rather than think about that (which seems confusing), we decide to try the definition interactively using the Pc-Nqthm command `VERIFY-DEFN`, by submitting the form `(DEFN INDUCTION-FN ...)` above but with `VERIFY-DEFN` in place of `DEFN`. The system responds by putting us in the Pc-Nqthm interactive loop, where the goal is exactly that which must be proved in order to accept the definition of `INDUCTION-FN`. We issue the `SPLIT` command, since the goal is a rather complicated conjunction, so that the system will simplify the propositional structure to conjunctive normal form, creating one new subgoal for each resulting conjunct. In fact the system creates two subgoals, the first of which is as follows.

```
(ordinalp (cons (add1 (maxlist x))
                (occurrences (maxlist x) x)))
```

We are able to prove this goal automatically, by issuing the `PROVE` command so that Pc-Nqthm invokes the full power of the Nqthm prover to prove this goal.

When we look at the second goal (see Point 3 in Appendix A), we see that there are a number of terms that we might want to prove something about, in order to simplify them. But we can get pretty lost in a sea of such lemmas. Let's think for a moment about our strategy. Recall that we imagine taking two elements out of a list, namely the maximum and the minimum, and moving them to the front after a decrement and an increment. Let us factor all that into simpler steps: move one element to the front, then move another, then modify the first two elements of the list. So, before thinking too much further, we formulate the following lemma, which may be useful:

```
(prove-lemma maxlist-delete1-rearrange (rewrite)
             (implies (member b x)
                      (equal (maxlist (cons b (delete1 b x)))
                             (maxlist x))))
```

Unfortunately, the proof attempt fails. When we look at the failed proof, where do we look? The proof by induction yields several cases, all but the last of which are proved immediately. One might be tempted to look at the last case, but we believe that it is more useful to look at goals that are stable under simplification, so that attention is not wasted on terms that can be further simplified.

Now, when we look at the failed proof, using the checkpoint tool, we find:

```
(implies
 (and (listp x)
      (not (equal (maxlist (cdr x)) (car x)))
      (not (lessp (maxlist (cdr x))
```

```
                  (maxlist (delete1 (maxlist (cdr x)) (cdr x)))))
      (member (maxlist (cdr x)) (cdr x))
      (not (lessp (car x) (maxlist (cdr x)))))
 (not (numberp (car x))))
```

But when we stare at this for awhile, we realize that there doesn't seem to be any reasonable argument for the truth of this goal – after all, the third hypothesis `(NOT (LESSP ...))` is always true, as is the one following it, and we can make the second and fifth ones true by choosing a list whose `CAR` is bigger than the maximum of its `CDR`. So, we might guess that a list containing two numbers, where the first is bigger than the second (for example, `10` followed by `3`), would provide a counterexample. Let us use the execution facility offered by Nqthm, by issuing the command `(R-LOOP)`, to verify that such a list provides a counterexample to the goal above. In the following display, terms following the prompt "*" are the input, and as always, semicolons precede comments.

```
*(let ((x '(10 3))) ; Bind x to the list suggested above.
      (implies
       (and (listp x)
            (not (equal (maxlist (cdr x)) (car x)))
            (not (lessp (maxlist (cdr x))
                        (maxlist (delete1 (maxlist (cdr x)) (cdr x)))))
            (member (maxlist (cdr x)) (cdr x))
            (not (lessp (car x) (maxlist (cdr x)))))
       (not (numberp (car x)))))
 F ; Aha!  The goal is false using the indicated value of x.
*(let ((x '(10 3)) (b 3))
   ; Let us see if we can construct a counterexample to the theorem
   ; maxlist-delete1-rearrange, which we are in the process of trying to prove.
      (implies (member b x)
               (equal (maxlist (cons b (delete1 b x))) (maxlist x))))
 F ; Aha!  The ``theorem'' is false using the indicated values of x and b.
*(let ((x '(10 3)) (b 3))
   ; Let us see what the values of some of the terms are in the
   ; counterexample above.
   (list (maxlist (cons b (delete1 b x)))
         (delete1 b x)
         (maxlist x)))
 '(3 NIL 10)
```

AHA!! We find that `(DELETE1 B X)` returns `NIL` in our example, even though for the indicated values of `B` and `X` we would expect the result to be a list containing `10`. And sure enough, when we look back at the definition of `DELETE1`, we find a bug: when we recur, we need to return

```
(cons (car x) (delete1 elt (cdr x))),
```

not

```
(delete1 elt (cdr x)).
```

So, we undo the definition of DELETE1:

```
(ubt)
```

Here is the correct definition for DELETE1:

```
(DEFN DELETE1 (ELT X)
  (IF (LISTP X)
      (IF (EQUAL ELT (CAR X))
          (CDR X)
        (CONS (CAR X) (DELETE1 ELT (CDR X))))
    X))
```

And now, the proof of MAXLIST-DELETE1-REARRANGE succeeds.

```
(PROVE-LEMMA MAXLIST-DELETE1-REARRANGE (REWRITE)
             (IMPLIES (MEMBER B X)
                      (EQUAL (MAXLIST (CONS B (DELETE1 B X)))
                             (MAXLIST X))))
```

5.5 Outline of termination proof

Our initial attempts to admit the definition of INDUCTION-FN, shown in the previous Subsection, are serious failures. The main discovery during that process is a re-affirmation of an important idea when using Nqthm: try to keep track of where you are going, formulating lemmas that will lead to the goal at hand. In particular, we have found that if we submit the definition by using VERIFY-DEFN in place of DEFN, the Pc-Nqthm interactive loop presents us with two goals, one of which is easy for Nqthm to simplify to T. Let us state the other as an axiom, for now, following the same general approach elucidated earlier in Subsection 5.2: top-down proof. To increase readability we re-introduce the variable X0 to represent the list constructed in the induction scheme.

```
(add-axiom induction-fn-help-2 (rewrite)
  (let ((x0 (cons (sub1 (maxlist x))
                  (cons (add1 (minlist x))
                        (delete1 (maxlist x)
                                 (delete1 (minlist x) x))))))
    (implies
     (and (equal (sumlist x) (times k (length x)))
          (not (equal (minlist x) (maxlist x))))
     (ord-lessp
      (cons (add1 (maxlist x0)) (occurrences (maxlist x0) x0))
      (cons (add1 (maxlist x)) (occurrences (maxlist x) x))))))
```

We formulate this axiom so that Nqthm can immediately admit the definition of `INDUCTION-FN`. In order to make that happen, we would like to instruct Nqthm that it should ignore the definitions of all the functions that appear in the definition of `INDUCTION-FN` or in the axiom above. We do this because we believe that we have already proved (or assumed) exactly what we need, and if Nqthm applies any additional information to the problem, the direction of the proof may somehow change, eliminating the applicability of rewrite rules (especially, of the axiom introduced above). Fortunately, Nqthm supports the capability to control which rules (especially, which definitions and which rewrite rules) are "enabled", i.e., are allowed to be used automatically by the theorem prover. Rules (including definitions) are enabled by default, but the user may disable them.

We may disable the relevant functions individually. However, it is convenient to assign a name, say `INDUCTION-FN-DISABLES`, to the appropriate list of function names.

```
(deftheory induction-fn-disables
 (sumlist times length minlist maxlist delete1 occurrences))
```

Then, we can *disable* all those names at once.

```
(disable-theory induction-fn-disables)
```

At this point Nqthm accepts the definition of `INDUCTION-FN`. For future reference, we note that `ENABLE-THEORY` is the analogous command for telling Nqthm to "remember again" the rules in the specified theory.

We turn now to the task of proving the `ADD-AXIOM` formula above. For, once we replace that event by a corresponding `PROVE-LEMMA` event (and make sure that we have also executed the `DISABLE-THEORY` event above), then Nqthm should (we hope) again accept the definition of `INDUCTION-FN`. At this point we could use our induction scheme to begin the proof of the "main

lemma," deferring until later the proof of the ADD-AXIOM event above. However our preference, though not a particularly strong one, is to proceed as already indicated: back up and prove the relevant axiom. That way we don't leave this loose end to deal with later. So, we undo all events back through the axiom INDUCTION-FN-HELP-2, using the following command.

```
(ubt induction-fn-help-2)
```

Our first step is to break the axiom above into two cases, corresponding to Cases 2.2 and 2.1 in the hand proof in Lemma 2, i.e., corresponding to whether or not the maximum occurs twice in the list. Continuing in a top-down decompositional style, we formulate the two cases now as axioms. We will use these axioms explicitly (see the discussion of USE hints, page 23), so we do not label them for automatic use as rewrite rules.

```
(add-axiom induction-fn-help-2-max-occurs-twice ()
  ; Case 2.2 from the hand proof.
  (let ((x0 (cons (sub1 (maxlist x))
                  (cons (add1 (minlist x))
                        (delete1 (maxlist x)
                                 (delete1 (minlist x) x))))))
    (implies
     (and (equal (sumlist x) (times k (length x)))
          (not (equal (minlist x) (maxlist x)))
          (lessp 1 (occurrences (maxlist x) x)))
     (ord-lessp
      (cons (add1 (maxlist x0)) (occurrences (maxlist x0) x0))
      (cons (add1 (maxlist x)) (occurrences (maxlist x) x))))))

(add-axiom induction-fn-help-2-max-occurs-once ()
  ; Case 2.1 from the hand proof.
  (let ((x0 (cons (sub1 (maxlist x))
                  (cons (add1 (minlist x))
                        (delete1 (maxlist x)
                                 (delete1 (minlist x) x))))))
    (implies
     (and (equal (sumlist x) (times k (length x)))
          (not (equal (minlist x) (maxlist x)))
          (not (lessp 1 (occurrences (maxlist x) x))))
     (ord-lessp
      (cons (add1 (maxlist x0)) (occurrences (maxlist x0) x0))
      (cons (add1 (maxlist x)) (occurrences (maxlist x) x))))))
```

And now we may make the desired conclusion by instructing Nqthm to use the above axioms explicitly. Since we expect the following theorem to follow trivially from the two cases, we instruct the theorem prover to disable all rules in the database, using the `(DISABLE-THEORY T)` hint shown below, except for those rules (such as the definition of `IMPLIES`) that are built in, by using the `(ENABLE-THEORY GROUND-ZERO)` hint.

```
(prove-lemma induction-fn-help-2 (rewrite)
  (let ((x0 (cons (sub1 (maxlist x))
                  (cons (add1 (minlist x))
                        (delete1 (maxlist x)
                                 (delete1 (minlist x) x))))))
    (implies
     (and (equal (sumlist x) (times k (length x)))
          (not (equal (minlist x) (maxlist x))))
     (ord-lessp
      (cons (add1 (maxlist x0)) (occurrences (maxlist x0) x0))
      (cons (add1 (maxlist x)) (occurrences (maxlist x) x)))))
  ((use (induction-fn-help-2-max-occurs-twice)
        (induction-fn-help-2-max-occurs-once))
   (disable-theory t)
   (enable-theory ground-zero)))
```

Finally let us undo, executing the command

```
(ubt induction-fn-help-2-max-occurs-twice)
```

in anticipation of proving the two cases. Now that we are ready to proceed in a "forward" manner, we submit the `DEFTHEORY` event that we "undid" earlier.

```
(DEFTHEORY INDUCTION-FN-DISABLES
  (SUMLIST TIMES LENGTH MINLIST MAXLIST DELETE1 OCCURRENCES))
```

5.6 Proof of Lemma 1 from "hand proof"

Let us prove something that we needed in the hand proof for the case that the maximum occurs once: namely, that if the maximum and minimum of a list are distinct then the maximum is at least 2 greater than the minimum. This corresponds to Lemma 1 in the hand proof, Section 4. Interestingly, we will find this lemma useful for the case that the maximum occurs twice as well, as we will see when we get to Point 7 of Appendix A. But that is an "accident"; in fact, we are starting the proof of Lemma 1 simply because it seems fun at the moment,

and because we have already seen in our hand proof that we'll need this fact. It is often important when using Nqthm not to waste time proving irrelevant facts merely because they *seem* relevant. But, we are confident that we will indeed need the following fact (which we will call MINLIST-LESS-THAN-MAXLIST-MINUS-1).

```
(implies (and (listp x)
              (equal (sumlist x) (times k (length x)))
              (not (equal (minlist x) (maxlist x))))
         (lessp (minlist x) (sub1 (maxlist x))))
```

Although this fact seems obvious, we do not immediately see an obvious proof of this theorem, so we do not expect Nqthm to see a proof either, without some help. What is our plan? We follow the outline presented in the hand proof, page 18. We consider the following two events, which turn out to be useful later.

```
(PROVE-LEMMA MAX-GREATER-THAN-AVERAGE (REWRITE)
             (IMPLIES (AND (LISTP X)
                           (GEQ (SUMLIST X) (TIMES K (LENGTH X)))
                           (NOT (EQUAL (FIX K) (MAXLIST X))))
                      (LESSP K (MAXLIST X))))

(PROVE-LEMMA MIN-LESS-THAN-AVERAGE (REWRITE)
             (IMPLIES (AND (LISTP X)
                           (LEQ (SUMLIST X) (TIMES K (LENGTH X)))
                           (NOT (EQUAL (FIX K) (MINLIST X))))
                      (LESSP (MINLIST X) K)))
```

Clearly their conclusions together show that (MINLIST X) + 1 < (MAXLIST X). However, we confess that probably our thinking isn't clear as we formulate these lemmas; it's not obvious just how they help us. Anyhow, among our thoughts are the following, which illustrate some generally useful ideas.

1. Each of these lemmas only involves one function from {MINLIST, MAXLIST}. Often it is easier to reason about fewer functions at any given time.

2. The equality hypothesis in our main goal doesn't seem amenable to proof by induction, because it is not preserved when we replace X with (CDR X). To be honest: rather than thinking this point through carefully, instead we just press on, hoping that our intuition is correct.

For more details about the proof of MAX-GREATER-THAN-AVERAGE see Point 4 in Appendix A.

At this point we wonder if the lemmas above are enough to prove MINLIST-LESS-THAN--MAXLIST-MINUS-1. We try it, but the proof fails. That is not surprising, because probably first we have to use the lemmas above to show that if (MAXLIST X) ≠ (MINLIST X), then neither one is equal to K. At any rate, after some fumbling around, we decide to prove the following

lemma (that we are able to prove only after proving a lemma stating that (TIMES (LENGTH X) (MINLIST X)) is not greater than (SUMLIST X); see Point 5 in Appendix A):

```
(PROVE-LEMMA MINLIST-NOT-MAXLIST-IMPLIES-MINLIST-LESSP-AVERAGE-LEMMA ()
             (IMPLIES (NOT (EQUAL (MINLIST X) (MAXLIST X)))
                      (LESSP (TIMES (MINLIST X) (LENGTH X)) (SUMLIST X))))
```

Notice that we haven't made it a rewrite rule. This is because we will be USE-ing it to prove the lemma below. If you give a rewrite rule, say R, as a USE hint, then you are proving (IMPLIES R ...), and there is a danger that R will rewrite to T, thus making R USEless!

Then, the following is an immediate consequence of the lemma above, by plugging in the equality hypothesis below into the lemma above. It doesn't prove automatically, so we'll just give a USE hint. The use of FIX here is not important.

```
(PROVE-LEMMA MINLIST-NOT-MAXLIST-IMPLIES-MINLIST-LESSP-AVERAGE (REWRITE)
             (IMPLIES (AND (LISTP X)
                           (EQUAL (SUMLIST X) (TIMES K (LENGTH X)))
                           (NOT (EQUAL (MINLIST X) (MAXLIST X))))
                      (LESSP (MINLIST X) (FIX K)))
             ((USE (MINLIST-NOT-MAXLIST-IMPLIES-MINLIST-LESSP-AVERAGE-LEMMA))))
```

Now we try to prove the analog of the lemma MINLIST-NOT-MAXLIST-IMPLIES-MINLIST--LESSP-AVERAGE-LEMMA, modified for MAXLIST in place of MINLIST. The proof fails, but it succeeds after we have proved an analog of the lemma we needed to prove MINLIST-NOT-MAXLIST--IMPLIES-MINLIST-LESSP-AVERAGE-LEMMA. See for further details Point 6 of Appendix A.

And, the main sub-lemma follows by the analogous argument:

```
(PROVE-LEMMA MINLIST-NOT-MAXLIST-IMPLIES-MAXLIST-GREATERP-AVERAGE (REWRITE)
             (IMPLIES (AND (LISTP X)
                           (EQUAL (SUMLIST X) (TIMES K (LENGTH X)))
                           (NOT (EQUAL (MINLIST X) (MAXLIST X))))
                      (LESSP (FIX K) (MAXLIST X)))
          ((USE (MINLIST-NOT-MAXLIST-IMPLIES-MAXLIST-GREATERP-AVERAGE-LEMMA))))
```

In fact, the lemma below now proves! But, it requires the USE hints that are given to it. But then, why didn't we disable the lemma above (see the comment below MINLIST-NOT-MAXLIST--IMPLIES-MINLIST-LESSP-AVERAGE-LEMMA), since we generally try to disable rewrite rules that are given to USE hints? As explained above, the issue is that we do not want a USEd hypothesis to rewrite to T. However, the theorem prover reports that the rule above is a *linear rule*; actually,

as explained in [5], the theorem prover creates a linear rule when the top-level function symbol in the conclusion is LESSP, or even when the conclusion is the negation of such a term. Linear rules are only used to derive contradictions, not to rewrite, and hence the concern about USE hints is groundless in this case.

Finally: should we modify the form of the following lemma? The theorem prover reports the presence of the free variable K in the hypothesis, when it attempts to process this as a linear lemma. In fact it seems harmless enough to leave this lemma the way it is. We can always apply it with a USE hint if we need to, and in fact, there is some hope that it will apply automatically since the "K" in the hypothesis is (lexically) present in our top-level lemma and in many sublemmas that we are contemplating – so, perhaps the prover will be able to relieve the hypothesis involving K, after all!

```
(PROVE-LEMMA MINLIST-LESS-THAN-MAXLIST-MINUS-1 (REWRITE)
             (IMPLIES (AND (LISTP X)
                           (EQUAL (SUMLIST X) (TIMES K (LENGTH X)))
                           (NOT (EQUAL (MINLIST X) (MAXLIST X))))
                      (LESSP (MINLIST X) (SUB1 (MAXLIST X))))
             ((USE
               (MINLIST-NOT-MAXLIST-IMPLIES-MINLIST-LESSP-AVERAGE)
               (MINLIST-NOT-MAXLIST-IMPLIES-MAXLIST-GREATERP-AVERAGE))))
```

5.7 Proof of hand proof's Case 2.2

Our goal in this section is to prove the lemma INDUCTION-FN-HELP-2-MAX-OCCURS-TWICE, stated in Subsection 5.5. Informally put, this lemma says that if the maximum of X occurs at least twice in X, then X0 (defined earlier, page 26) is "smaller" than X in an appropriate sense. This lemma follows immediately from the following two claims, which say that when X is replaced by X0, then the maximum stays the same, but it occurs fewer times. In fact, after executing these two ADD-AXIOM events, we are able to prove the lemma, with the hint (disable-theory induction-fn-disables):

```
(add-axiom induction-fn-help-2-max-occurs-twice-lemma-1 (rewrite)
  (let ((x0 (cons (sub1 (maxlist x))
                  (cons (add1 (minlist x))
                        (delete1 (maxlist x)
                                 (delete1 (minlist x) x))))))
    (implies
     (and (equal (sumlist x) (times k (length x)))
          (not (equal (minlist x) (maxlist x)))
          (lessp 1 (occurrences (maxlist x) x)))
     (equal (maxlist x0)
            (maxlist x)))))
```

```
(add-axiom induction-fn-help-2-max-occurs-twice-lemma-2 (rewrite)
  (let ((x0 (cons (sub1 (maxlist x))
                  (cons (add1 (minlist x))
                        (delete1 (maxlist x)
                                 (delete1 (minlist x) x))))))
    (implies
     (and (equal (sumlist x) (times k (length x)))
          (not (equal (minlist x) (maxlist x)))
          (lessp 1 (occurrences (maxlist x) x)))
     (equal (lessp (occurrences (maxlist x)
                                x0)
                   (occurrences (maxlist x) x))
            t))))
```

Notice that the conclusion of the second lemma above has the form `(EQUAL ... T)`. When we didn't use this form, the system complained on an earlier version of this lemma (which had a "typo") that it was trying to store this as a linear rule, and reported that it could not do so. Rather than think about this problem, we simply wrote the conclusion as an `EQUAL` term rather than a `LESSP` term, so that Nqthm will store it as an ordinary replacement rule. In fact, the present version can actually be stored as a linear rule, but it seems reasonable to take the simpler route of just using rewriting, which is somehow easier to understand.

The Nqthm rewriter works from the inside out; it rewrites all arguments of a term before rewriting the entire term. Therefore, we apply `MAXLIST` to `X`, not to `X0`, in the second lemma. For if the second lemma's left side (the `LESSP` term in the equality) contained an occurrence of `(MAXLIST X0)`, it would probably not match any term encountered by the rewriter, because the first lemma will already have been used to rewrite `(MAXLIST X0)` to `(MAXLIST X)`.

After undoing these two axioms, we turn to the task of proving the first of them. We think that the theorem is too hard to prove automatically and so we decide to prove some lemmas to simplify the problem. In particular we wish to prove rewrite rules that will simplify the term

```
(maxlist (delete1 (maxlist x) (delete1 (minlist x) x))),
```

since we expect this term to appear after the simplifier applies the definition of `MAXLIST` twice (see Point 7 of Appendix A).

Rather than try to reason directly about this term, we formulate two simple lemmas. The first implies that neither call of `DELETE1` in this term changes the maximum, except that the second lemma is necessary in order to guarantee that the maximum still occurs twice after deletion of the minimum.

```
(PROVE-LEMMA
 DELETE1-PRESERVES-MAXLIST-WHEN-MAXLIST-OCCURS-MORE-THAN-ONCE
 (REWRITE)
 (IMPLIES (LESSP 1 (OCCURRENCES (MAXLIST X) X))
          (EQUAL (MAXLIST (DELETE1 ANY-ELEMENT X)) (MAXLIST X))))

(PROVE-LEMMA DELETE1-OCCURRENCES (REWRITE)
             (EQUAL (OCCURRENCES A (DELETE1 B X))
                    (IF (AND (EQUAL A B)
                             (MEMBER B X))
                        (SUB1 (OCCURRENCES A X))
                      (OCCURRENCES A X))))
```

Details pertaining to the proof of the first lemma above are in Point 8, Appendix A. See Point 9 of Appendix A for a discussion about the form that we use for the second lemma above.

Now we try to prove what we have been working towards, i.e the lemma `INDUCTION-FN-HELP--2-MAX-OCCURS-TWICE-LEMMA-1`. The proof attempt fails, but the checkpoint tool (actually we have to use it twice) suggests to us the lemmas that suffice to prove it (see Point 10 in Appendix A).

Our next goal is to prove `INDUCTION-FN-HELP-2-MAX-OCCURS-TWICE-LEMMA-2`. Maybe we have proved enough lemmas by now so that this will go through automatically. So, we try it, but our attempt fails. Again, as for the previous case, we are able to prove the lemma by making use of the checkpoint tool (see Point 11 in Appendix A for more details). And finally we are able to prove `INDUCTION-FN-HELP-2-MAX-OCCURS-TWICE`, i.e., half of what we need for our termination argument. (See page 33 for explanation of the `DISABLE-THEORY` hint on the last line below.)

```
(PROVE-LEMMA INDUCTION-FN-HELP-2-MAX-OCCURS-TWICE ()
  (LET ((X0 (CONS (SUB1 (MAXLIST X))
                  (CONS (ADD1 (MINLIST X))
                        (DELETE1 (MAXLIST X)
                                 (DELETE1 (MINLIST X) X))))))
    (IMPLIES
     (AND (EQUAL (SUMLIST X) (TIMES K (LENGTH X)))
          (NOT (EQUAL (MINLIST X) (MAXLIST X)))
          (LESSP 1 (OCCURRENCES (MAXLIST X) X)))
     (ORD-LESSP
      (CONS (ADD1 (MAXLIST X0)) (OCCURRENCES (MAXLIST X0) X0))
      (CONS (ADD1 (MAXLIST X)) (OCCURRENCES (MAXLIST X) X)))))
 ((DISABLE-THEORY INDUCTION-FN-DISABLES)))
```

5.8 Proof of hand proof's Case 2.1

An analysis similar to that at the start of the preceding Subsection suggests that the following goal suffices for proving Case 2.1 from the hand proof. In fact our approach is less thoughtful; for details see Point 12 of Appendix A.

```
(add-axiom induction-fn-help-2-max-occurs-once-main-lemma (rewrite)
  (implies
   (and
    (equal (sumlist x) (times k (length x)))
    (not (equal (minlist x) (maxlist x)))
    (not (lessp 1 (occurrences (maxlist x) x))))
   (equal (lessp (maxlist (cons (sub1 (maxlist x))
                                (cons (add1 (minlist x))
                                      (delete1 (maxlist x)
                                               (delete1 (minlist x) x)))))
                 (maxlist x))
          t)))
```

In fact, this axiom suffices to prove `INDUCTION-FN-HELP-2-MAX-OCCURS-ONCE`.

So we focus on proving it as a lemma. After fumbling around trying to do the next proof (details omitted), we realize that we are having trouble getting the theorem prover to apply the lemma `MINLIST-LESS-THAN-MAXLIST-MINUS-1` because its first hypothesis, `(LISTP X)`, is hard to prove in the current setting. But then we realize that this hypothesis is unnecessary; it follows from another hypothesis of that lemma, namely that the `MINLIST` and `MAXLIST` of `X` are distinct. In fact, we are able to prove a better version of `MINLIST-LESS-THAN-MAXLIST--MINUS-1` (see page 38), `MINLIST-LESS-THAN-MAXLIST-MINUS-1-BETTER`, obtained by removing the hypothesis `(LISTP X)` (see Point 13 in Appendix A). Now, let us return to the proof of `INDUCTION-FN-HELP-2-MAX-OCCURS-ONCE-MAIN-LEMMA`. Sometimes it is useful to enter the interactive proof-checker and give a simplification command, rather than to use the checkpoint tool with the theorem prover. Let's try that. The last line in the `VERIFY` form below contains a (one-element) list of initial commands to issue to Pc-Nqthm. We could just as well have waited to enter the interactive loop, omitting that final argument to `VERIFY`, and typed:

```
(bash (disable occurrences minlist delete1))
```

This command invokes the prover's simplifier with a hint to disable (ignore) the definitions of the indicated functions. That is useful as a way of saving the prover the trouble (hence the time!) of exploring the result of expanding these functions.

```
(verify
```

```
(implies
 (and
  (equal (sumlist x) (times k (length x)))
  (not (equal (minlist x) (maxlist x)))
  (not (lessp 1 (occurrences (maxlist x) x))))
 (equal (lessp (maxlist (cons (sub1 (maxlist x))
                              (cons (add1 (minlist x))
                                    (delete1 (maxlist x)
                                             (delete1 (minlist x) x)))))
               (maxlist x))
        t))
((bash (disable occurrences minlist delete1))))
```

The result suggests to us some lemmas (proved with the help of the checkpoint tool) that suffice to prove INDUCTION-FN-HELP-2-MAX-OCCURS-ONCE-MAIN-LEMMA (see Point 14 in Appendix A).

Then we are able to prove our goal:

```
(PROVE-LEMMA INDUCTION-FN-HELP-2-MAX-OCCURS-ONCE ()
   (LET ((X0 (CONS (SUB1 (MAXLIST X))
                   (CONS (ADD1 (MINLIST X))
                         (DELETE1 (MAXLIST X)
                                  (DELETE1 (MINLIST X) X))))))
    (IMPLIES
     (AND (EQUAL (SUMLIST X) (TIMES K (LENGTH X)))
          (NOT (EQUAL (MINLIST X) (MAXLIST X)))
          (NOT (LESSP 1 (OCCURRENCES (MAXLIST X) X))))
     (ORD-LESSP
      (CONS (ADD1 (MAXLIST X0)) (OCCURRENCES (MAXLIST X0) X0))
      (CONS (ADD1 (MAXLIST X)) (OCCURRENCES (MAXLIST X) X)))))
 ((DISABLE OCCURRENCES MINLIST MAXLIST DELETE1)))
```

5.9 Completion of termination proof

Finally, we may prove INDUCTION-FN-HELP-2 and complete the proof of termination. (See also Point 15 in Appendix A.) All we have to do is submit the definition whose termination we have been proving, surrounding it with the theory management that we introduced earlier in this Section. Thus, the first event below instructs the theorem prover to ignore the rules in the theory INDUCTION-FN-DISABLES (which was earlier introduced by a DEFTHEORY event), while the final event returns those rules to their enabled status.

```
(DISABLE-THEORY INDUCTION-FN-DISABLES)

(DEFN INDUCTION-FN (X K)
  (IF (NOT (EQUAL (SUMLIST X) (TIMES K (LENGTH X))))
      T
    (IF (EQUAL (MINLIST X) (MAXLIST X))
        T
      (INDUCTION-FN (CONS (SUB1 (MAXLIST X))
                          (CONS (ADD1 (MINLIST X))
                                (DELETE1 (MAXLIST X)
                                         (DELETE1 (MINLIST X) X))))
                    K)))
  ((ORD-LESSP (CONS (ADD1 (MAXLIST X)) (OCCURRENCES (MAXLIST X) X)))))

(ENABLE-THEORY INDUCTION-FN-DISABLES)
```

5.10 The rest of the proof

Now we attack our main lemma. The hint `(INDUCT (INDUCTION-FN X K))` shown below tells the prover to use the induction scheme suggested by the term `(INDUCTION-FN X K)`, in the sense described in Subsections 5.3 and (more generally) 2.4.

```
(prove-lemma main-lemma ()
             (let ((n (length x)))
               (implies (equal (sumlist x) (times k n))
                        (not (lessp (exp k n) (prodlist x)))))
             ((induct (induction-fn x k))))
```

The proof fails, but that isn't surprising: so far we have only concentrated on the proof of termination of `INDUCTION-FN`. (Admittedly, that is probably by far the hardest part; but, it's not the only part.) We inspect the proof and this suggests that we prove a simple lemma about terms of the form `(SUMLIST (DELETE1 ...))` (see Point 16 in Appendix A).

At this point the proof of the main lemma still fails, and we wake up and say to ourselves: "We haven't even proved the base case yet!" Why should we perhaps do the base case first?

1. It seems reasonable to prove easy things first, because sometimes one develops useful rewrite rules in the process that help simplify the harder goals into more comprehensible forms.

2. Generally speaking, base cases tend to be easier than inductive steps.

Below we show the lemma proved for the base case. We describe the interaction with Nqthm to obtain it in Point 17 of Appendix A.

```
(PROVE-LEMMA MAIN-LEMMA-BASE-CASE (REWRITE)
             (IMPLIES (AND (EQUAL (SUMLIST X) (TIMES K (LENGTH X)))
                           (EQUAL (MINLIST X) (MAXLIST X)))
                      (EQUAL (EXP K (LENGTH X)) (PRODLIST X)))
             ((USE (MAIN-LEMMA-BASE-CASE-LEMMA-1)
                   (MAIN-LEMMA-BASE-CASE-LEMMA-2))))
```

For the sake of completeness we show also the facts stated by the two lemmas in the `USE` hint above.

```
(implies (equal (minlist x) (maxlist x))
         (equal (exp (minlist x) (length x)) (prodlist x)))

(implies (and (listp x)
              (equal (sumlist x) (times k (length x)))
              (equal (minlist x) (maxlist x)))
         (equal (minlist x) (fix k)))
```

We turn now to the inductive step. Since we just returned from a weekend break, we get our bearing by entering the Pc-Nqthm interactive loop; see Point 18 in Appendix A for details. Briefly put, we are led to the following lemma: `(IMPLIES (LISTP X) (MEMBER (MINLIST X) X))`. This is clearly not always true, for example if `X` contains a single non-numeric element (in which case `(MINLIST X)` is 0, which does not belong to `X`).

To fix this problem, let us work in a somewhat more "typed" framework by assuming in our top-level theorem that we have a list of numbers. So, at this point we redo all of our work, but adding the following definition:

```
(DEFN NUMBERP-LISTP (X)
  (IF (LISTP X)
      (AND (NUMBERP (CAR X))
           (NUMBERP-LISTP (CDR X)))
    (EQUAL X NIL)))
```

We'll try to fix things by adding `(NUMBERP-LISTP X)` as a top-level hypothesis when we finally glue all this together.

And now, an adjusted ("typed") version of the lemma mentioned above can be proved.

```
(PROVE-LEMMA MEMBER-MINLIST (REWRITE)
             (IMPLIES (AND (LISTP X)
```

```
                    (NUMBERP-LISTP X))
               (MEMBER (MINLIST X) X)))
```

Interestingly, we see that we have already proved an analogous lemma for `MAXLIST`, i.e., `MEMBER--MAXLIST` (see Point 19 in Appendix A).

Here is the revised version of the main lemma. The `DO-NOT-INDUCT` hint below is simply a device to get the theorem prover to abort the proof rather than push a goal to be proved by a (sub-)induction.

```
(prove-lemma main-lemma ()
             (let ((n (length x)))
               (implies (and (numberp-listp x)
                             (equal (sumlist x) (times k n)))
                        (not (lessp (exp k n) (prodlist x)))))
             ((induct (induction-fn x k))
              (do-not-induct t)))
```

The main lemma fails. A long interaction with the prover (see Point 20 in Appendix A) leads us to the heart of the arithmetic argument, which corresponds in the hand proof to the reasoning that allows us to prove equation (1) in Section 4. We can represent this argument informally on paper as follows, where we write the product of the original list as `min * max * rest`.

```
(min + 1) * (max - 1) * rest
=
[(min * max) + max - min - 1] * rest
=
(min * max * rest) + ([max - min - 1] * rest)
=
prodlist(x) + ([max - min - 1] * rest)
```

As usual, we would like to state facts in terms of useful rewrite rules when possible. When that seems too awkward or difficult, we can try to formulate lemmas that can be used, with `USE` hints, to give the desired result. However, let us see if the direct approach of proving a single rewrite rule that rewrites an appropriate term can be used. The following lemma formalizes the content of the informal equations displayed above . (In Point 21 of Appendix A we show the details of its proof.)

```
(PROVE-LEMMA PRODUCT-OF-MODIFIED-LIST (REWRITE)
```

```
            (IMPLIES (AND (LESSP MIN MAX)
                          (MEMBER MIN X)
                          (MEMBER MAX X))
                     (LET ((REST (PRODLIST (DELETE1 MAX (DELETE1 MIN X)))))
                          (EQUAL (TIMES (ADD1 MIN) (SUB1 MAX) REST)
                                 ;; prodlist(x) + ([max - min - 1] * rest)
                                 (PLUS (PRODLIST X)
                                       (TIMES (DIFFERENCE MAX (ADD1 MIN))
                                              REST)))))
            ((USE (POSITIVE-IMPLIES-NUMBERP))))
```

The lemma `POSITIVE-IMPLIES-NUMBERP` in the `USE` hint above states that if `(LESSP MIN MAX)` then `MAX` is a number. We had to prove this lemma, because the linear arithmetic decider wasn't able to infer it automatically in the course of the proof of `PRODUCT-OF-MODIFIED-LIST`.

And now the main lemma goes through!

```
(PROVE-LEMMA MAIN-LEMMA ()
             (LET ((N (LENGTH X)))
               (IMPLIES (AND (NUMBERP-LISTP X)
                             (EQUAL (SUMLIST X) (TIMES K N)))
                        (NOT (LESSP (EXP K N) (PRODLIST X)))))
             ((INDUCT (INDUCTION-FN X K))
              (DO-NOT-INDUCT T)))
```

Then we paste down the events referred to in Subsection 5.2 (see also Point 1 in Appendix A) to prove the theorem from `MAIN-LEMMA`. See Point 22 of Appendix A.

Note however that we have introduced consideration of `NUMBERP-LISTP` into the problem, which is why a little further work may be required. And indeed, when we try our main theorem (see Point 23 in Appendix A for the details relative to this comment), the proof fails. However, the checkpoint tool suggests to us the lemma (see below) that we need to prove the main theorem!

```
(PROVE-LEMMA NUMBERP-LISTP-SCALAR-PRODUCT (REWRITE)
             (IMPLIES (NUMBERP-LISTP A)
                      (NUMBERP-LISTP (SCALAR-PRODUCT N A))))

(PROVE-LEMMA MAIN ()
             (IMPLIES (NUMBERP-LISTP A)
                      (LET ((N (LENGTH A)))
                        (GEQ (EXP (SUMLIST A) N)
                             (TIMES (EXP N N) (PRODLIST A)))))
             ((USE (MAIN-LEMMA (K (SUMLIST A))
```

```
                        (X (SCALAR-PRODUCT (LENGTH A) A))))))
```

5.11 Refining the theorem

Let us introduce a version of this theorem using the iteration construct `FOR` provided by Nqthm. We do not explain `FOR` here, except to say that it's similar to looping constructs in Lisp and other languages, and is documented of course in [5]; we believe that its meaning is reasonably self-evident. Our approach here is simply to note that the new version of `MAIN` is an immediate consequence of `MAIN` once we tell the rewriter how to write `SUMLIST` and `PRODLIST` in terms of `FOR`.

```
(PROVE-LEMMA SUMLIST-FOR (REWRITE)
             (EQUAL (SUMLIST A)
                    (FOR X IN A SUM X)))

(PROVE-LEMMA PRODLIST-FOR (REWRITE)
             (EQUAL (PRODLIST A)
                    (FOR X IN A MULTIPLY X)))

(PROVE-LEMMA MAIN-AGAIN ()
             (IMPLIES (AND (NUMBERP-LISTP A)
                           (EQUAL N (LENGTH A)))
                      (GEQ (EXP (FOR X IN A SUM X) N)
                           (TIMES (EXP N N)
                                  (FOR X IN A MULTIPLY X))))
             ((USE (MAIN))))
```

A More details of the mechanized proof

Point 1.

We try the following; see page 23 for explanation of the syntax of the `USE` hint.

```
(prove-lemma main ()
             (let ((n (length a)))
               (geq (exp (sumlist a) n)
                    (times (exp n n) (prodlist a))))
             ((use (main-lemma (k (sumlist a))
                               (x (scalar-product n a))
                               (n n)))))
```

The proof fails, or at least seems to; we abort when the prover tries to use induction, because we don't need induction at this level. Why does the proof fail? The checkpoint tool helps us to find the following goal.

```
(implies (and (not (equal (sumlist a) 0))
              (numberp n)
              (not (equal n (length (scalar-product n a)))))
         (not (lessp (exp (sumlist a) (length a))
                     (times (prodlist a)
                            (exp (length a) (length a)))))).
```

Look at the third hypothesis. It's obviously false; or is it? Our first instinct is to formulate a lemma that "canonicalizes" the `LENGTH` term in that hypothesis, even before we think hard about that hypothesis in its entirety. So, we formulate the rewrite rule shown below.

```
(prove-lemma length-scalar-product (rewrite)
             (equal (length (scalar-product n a)) (length a)))
```

The prover proves this lemma easily, but as we formulate it, we realize that the third hypothesis in the goal above would then simplify to `(NOT (EQUAL N (LENGTH A)))` rather than to what we had imagined, namely `(NOT (EQUAL N N))`, or `F`. And in fact, `N` does not occur anywhere else in the goal (except trivially in the second hypothesis). So, what's wrong? The problem is that "`N`" in the main lemma is merely an abbreviation for `(LENGTH A)`, bound by a `LET` form.

Now we try `MAIN` using the following hint:

```
(use (main-lemma (k (sumlist a))
                 (x (scalar-product (length x) a))))
```

But the proof still fails. The checkpoint tool suggests us the following goal:

```
(implies (and (not (equal (sumlist a) 0))
              (not (equal (length x) (length a))))
         (not (lessp (exp (sumlist a) (length a))
                     (times (prodlist a)
                            (exp (length a) (length a))))),
```

Why are `X` and `A` both in this goal? One variable is used in the lemma, the other in the theorem – it's odd to see them both in the same conjecture! Answer: our hint is still wrong. The lemma

uses X, which we want to instantiate in terms of A (which is used in the theorem). So, finally we are able to find the correct hint, shown in Subsection 5.2, and the proof of MAIN goes through.

Point 2.

Our first attempt at admitting the definition of INDUCTION-FN (see page 27 and the ensuing discussion) actually is a form that contains the following measure hint:

```
(ord-lessp
 (cons (maxlist x) (occurrences (maxlist x) x)))
```

Nqthm prints, among other things, the following response:

```
WARNING:  The formula:
      (ORDINALP (CONS (MAXLIST X) (OCCURRENCES (MAXLIST X) X)))
has not been proved.  This often happens because the first component,
(MAXLIST X), of the alleged ordinal is not strictly positive.
```

So, we replace (MAXLIST X) with (ADD1 (MAXLIST X)) in the definition of INDUCTION-FN, in order to meet the objection printed by the system.

Point 3.

The command TH, in response to the prompt "->:", yields the following output:

```
->: th ;; print current goal ("theorem")

*** Active top-level hypotheses:
H1.  (EQUAL (SUMLIST X)
            (TIMES K (LENGTH X)))
H2.  (NOT (EQUAL (MINLIST X) (MAXLIST X)))

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

The current subterm is:
(ORD-LESSP
   (CONS (ADD1 (MAXLIST (CONS (SUB1 (MAXLIST X))
                              (CONS (ADD1 (MINLIST X))
                                    (DELETE1 (MAXLIST X)
                                             (DELETE1 (MINLIST X) X))))))
         (OCCURRENCES (MAXLIST (CONS (SUB1 (MAXLIST X))
                                     (CONS (ADD1 (MINLIST X))
                                           (DELETE1 (MAXLIST X)
```

```
                                                  (DELETE1 (MINLIST X) X)))))
                     (CONS (SUB1 (MAXLIST X))
                           (CONS (ADD1 (MINLIST X))
                                 (DELETE1 (MAXLIST X)
                                          (DELETE1 (MINLIST X) X))))))
   (CONS (ADD1 (MAXLIST X))
         (OCCURRENCES (MAXLIST X) X)))
->:
```

Point 4.

We attempt to prove the lemma `MAX-GREATER-THAN-AVERAGE`; see page 36 for the statement of this lemma, except that initially there is `K` rather than `(FIX K)` in the third hypothesis. However, the proof fails. Using the checkpoint tool, we discover the goal

```
(implies (and (listp x)
              (not (listp (cdr x)))
              (not (numberp k)))
         (numberp (car x)))
```

which suggests that we might want to eliminate this case by applying `FIX` to `K` in the third hypothesis – in fact, the lemma is false otherwise. Then after fixing the third hypothesis, we find the checkpoint

```
(implies (and (not (listp x)) (numberp k))
         (equal k 0))
```

which suggests that we may need to assume that `X` is not empty – in fact, the theorem is false otherwise (consider `K = 92` and `X = NIL`). Thus, we add the hypothesis `(LISTP X)`. Then, the lemma is proved automatically, with those two modifications.

Point 5.

We start by trying to prove:

```
(prove-lemma minlist-not-maxlist-implies-minlist-lessp-average-lemma
             (rewrite)
             (implies (and (listp x)
                           (not (equal (minlist x) (maxlist x))))
                      (lessp (times (minlist x) (length x)) (sumlist x))))
```

That fails, but without thinking much about the failed proof, we simply decide to try again without the (LISTP X) hypothesis. It's generally easier, funny enough, to prove a stronger theorem in that sense: in other words, unnecessary hypotheses often confuse the theorem prover more than they help it.

```
(prove-lemma minlist-not-maxlist-implies-minlist-lessp-average-lemma
             (rewrite)
             (implies (not (equal (minlist x) (maxlist x)))
                      (lessp (times (minlist x) (length x)) (sumlist x))))
```

This proof also fails. Using the checkpoint tool and proving some facts by hand, we find that we have to prove the following lemmas (we omit here the details; see Point 1 in Appendix B). Here, the lemma LESSP-TIMES-PRESERVED-IN-FIRST-ARG is needed for MINLIST-MAIN-PROPERTY – it's just barely outside linear arithmetic! And the proof of LESSP-TIMES-PRESERVED-IN-FIRST--ARG, in turn, takes advantage of the first one below.

```
(PROVE-LEMMA TIMES-MONOTONE-1 (REWRITE)
             (IMPLIES (NOT (LESSP U V))
                      (NOT (LESSP (TIMES U Y) (TIMES V Y)))))

(PROVE-LEMMA LESSP-TIMES-PRESERVED-IN-FIRST-ARG (REWRITE)
             (IMPLIES (AND (NOT (LESSP A (TIMES U Y)))
                           (NOT (LESSP U V)))
                      (NOT (LESSP A (TIMES V Y)))))

(PROVE-LEMMA MINLIST-MAIN-PROPERTY (REWRITE)
             (NOT (LESSP (SUMLIST X)
                         (TIMES (LENGTH X) (MINLIST X)))))
```

And now, we are able to prove the lemma MINLIST-NOT-MAXLIST-IMPLIES-MINLIST-LESSP--AVERAGE-LEMMA.

Point 6.

We prove:

```
(PROVE-LEMMA MAXLIST-MAIN-PROPERTY (REWRITE)
             (NOT (LESSP (TIMES (LENGTH X) (MAXLIST X)) (SUMLIST X))))
```

And in fact, our proof by analogy succeeds!

```
(PROVE-LEMMA MINLIST-NOT-MAXLIST-IMPLIES-MAXLIST-GREATERP-AVERAGE-LEMMA ()
             (IMPLIES (NOT (EQUAL (MINLIST X) (MAXLIST X)))
                      (LESSP (SUMLIST X) (TIMES (MAXLIST X) (LENGTH X)))))
```

Point 7.

Consider the lemma INDUCTION-FN-HELP-2-MAX-OCCURS-TWICE-LEMMA-1; see page 38. Why is it true? We could try it automatically, but it looks too hard. In fact, we do try it automatically, and the proof appears to be failing, in that the prover uses generalization and pushes more than 10 goals to prove by induction by the time we abort. Informally, it seems that the problem is that there are two or three distinct pieces of reasoning that one naturally makes to prove this lemma, and it is awkward to try to "fold" them into the same proof.

Part of "the Nqthm approach" is to prove "canonicalization rules", in other words, rewrite rules that simplify a problem until it is obviously true. So for example, it is natural to ask how we can simplify a term of the form (MAXLIST (CONS A LST)), when (as in the lemma above) we know that A < (MAXLIST LST). In fact, we probably do *not* need such a lemma, because MAXLIST is defined by induction on LST, and such maxlist-of-cons terms will be expanded by the prover. (But we could prove such a lemma anyhow, if that made us feel more confident.)

In fact, if we submit this theorem to Pc-Nqthm, and then give the BASH command so that the prover simplifies the goal as much as possible (but does not use other Nqthm "processes" such as elimination or generalization, and does not use induction), we obtain 14 goals, but none of them has a call of CONS. So, the prover already appears to be handling adequately terms of the form (MAXLIST (CONS A LST)).

So, instead of concerning ourselves with how to simplify

```
(maxlist (cons (sub1 (maxlist x))
               (cons (add1 (minlist x))
                     (delete1 (maxlist x)
                              (delete1 (minlist x) x)))))
```

let us consider how to simplify what this clearly reduces to, because both (SUB1 (MAXLIST X)) and (ADD1 (MINLIST X)) are less than the maximum of "what is left" (from lemma MINLIST--LESS-THAN-MAXLIST-MINUS-1), informally speaking:

```
(maxlist (delete1 (maxlist x) (delete1 (minlist x) x)))
```

Point 8.

We try to prove DELETE1-PRESERVES-MAXLIST-WHEN-MAXLIST-OCCURS-MORE-THAN-ONCE (see page 39), but the proof attempt fails. The checkpoint tool helps us to formulate the following lemma (see Point 2 in Appendix B):

```
(PROVE-LEMMA MAXLIST-DELETE1-LEQ (REWRITE)
             (NOT (LESSP (MAXLIST LST) (MAXLIST (DELETE1 A LST)))))
```

Our next try of DELETE1-PRESERVES-MAXLIST-WHEN-MAXLIST-OCCURS-MORE-THAN-ONCE fails again. This time using the checkpoint tool we find that we need the following lemma (see Point 3 in Appendix B):

```
(PROVE-LEMMA MEMBER-IMPLIES-MAXLIST-GEQ (REWRITE)
             (IMPLIES (MEMBER A X)
                      (NOT (LESSP (MAXLIST X) A))))
```

And now, the proof of DELETE1-PRESERVES-MAXLIST-WHEN-MAXLIST-OCCURS-MORE-THAN-ONCE succeeds!

Point 9.

Note that there are cases, depending on whether the element we are deleting is the same as the element for which we are counting occurrences. Generally, it is better to formulate unconditional rewrite rules in Nqthm rather than conditional rules. Let us explain.

Consider the following two possible formulations.

```
(prove-lemma delete1-occurrences (rewrite)
             (implies (and (member b x)
                           (not (equal a b)))
                      (equal (occurrences a (delete1 b x))
                             (sub1 (occurrences a x)))))

(prove-lemma delete1-occurrences (rewrite)
             (equal (occurrences a (delete1 b x))
                    (if (and (equal a b)
                             (member b x))
                        (sub1 (occurrences a x))
                      (occurrences a x))))
```

Notice that in the first lemma we assume that A and B are distinct, but that this should suffice for our purposes, where it will be the case that A and B will be instantiated by (MAXLIST X) and (MINLIST X), respectively, which will be assumed to be distinct. (However, we could formulate a lemma dealing with the other case if necessary.)

The problem with the first formulation is that in order for the lemma to be applicable, the prover has to check that (MEMBER B X) and (NOT (EQUAL A B)). If the prover fails to rewrite

both of these conditions to `T` (under the appropriate substitution, which presumably will map `A` to `(MAXLIST X)` and `B` to `(MINLIST X)`), then the lemma will not apply. In that case, we might not really notice what is wrong. However, the second lemma has no hypotheses, so it will always apply. If the prover can prove `(MEMBER B X)` and `(NOT (EQUAL A B))`, then it will simplify the right-hand side to the same term that it would have obtained using the first formulation. But if it cannot prove both of these, then it will still apply the lemma, and we will observe a case split that will make the problem visible.

Point 10.

We try to prove `INDUCTION-FN-HELP-2-MAX-OCCURS-TWICE-LEMMA-1` as a lemma (see page 38). The proof fails (actually, it goes into induction, but we did all that work above so that it would be proved by simplification alone!), and using the checkpoint tool we conclude that we need the following lemma (see Point 4 in Appendix B):

```
(PROVE-LEMMA OCCURRENCE-IMPLIES-LISTP (REWRITE)
             (IMPLIES (LESSP 1 (OCCURRENCES A X))
                      (EQUAL (LISTP (DELETE1 A X)) T)))
```

The proof of `INDUCTION-FN-HELP-2-MAX-OCCURS-TWICE-LEMMA-1`, however, still fails. The checkpoint tool suggests that we prove the following lemma (see Point 5 in Appendix B):

```
(PROVE-LEMMA MAXLIST-GEQ-MINLIST (REWRITE)
             (NOT (LESSP (MAXLIST X) (MINLIST X))))
```

And now, our goal succeeds!!

```
(PROVE-LEMMA INDUCTION-FN-HELP-2-MAX-OCCURS-TWICE-LEMMA-1 (REWRITE)
             (IMPLIES
              (AND (EQUAL (SUMLIST X) (TIMES K (LENGTH X)))
                   (NOT (EQUAL (MINLIST X) (MAXLIST X)))
                   (LESSP 1 (OCCURRENCES (MAXLIST X) X)))
              (EQUAL (MAXLIST (CONS (SUB1 (MAXLIST X))
                                    (CONS (ADD1 (MINLIST X))
                                          (DELETE1 (MAXLIST X)
                                                   (DELETE1
                                                    (MINLIST X)
                                                    X)))))
                     (MAXLIST X))))
```

Point 11.

We try to prove INDUCTION-FN-HELP-2-MAX-OCCURS-TWICE-LEMMA-2 as a lemma (see page 38). The proof fails; more precisely, it uses induction and appears to be failing, so we abort. The checkpoint tool leads us to prove the lemma below (see Point 6 in Appendix B. Notice that we formulate it as unconditional rewrite rule. There are occasions where advanced Nqthm users might deliberately use a conditional rewrite rule even when it is possible to state an unconditional one instead; typically this happens when it is important to avoid case explosion. But, when that consideration is not clearly applicable, then we tend to prefer to use unconditional rules.

```
(PROVE-LEMMA MEMBER-DELETE1 (REWRITE)
             (EQUAL (MEMBER A (DELETE1 B C))
                    (IF (EQUAL A B)
                        (LESSP 1 (OCCURRENCES B C))
                      (MEMBER A C))))
```

When we try INDUCTION-FN-HELP-2-MAX-OCCURS-TWICE-LEMMA-2 again, the proof still fails. This time, the checkpoint tool leads us to prove the lemma below (see Point 7 in Appendix B).

```
(PROVE-LEMMA MEMBER-IMPLIES-LISTP (REWRITE)
             (IMPLIES (MEMBER A X) (LISTP X)))

(PROVE-LEMMA INDUCTION-FN-HELP-2-MAX-OCCURS-TWICE-LEMMA-2 (REWRITE)
 (IMPLIES
  (AND (EQUAL (SUMLIST X) (TIMES K (LENGTH X)))
       (NOT (EQUAL (MINLIST X) (MAXLIST X)))
       (LESSP 1 (OCCURRENCES (MAXLIST X) X)))
  (EQUAL (LESSP (OCCURRENCES (MAXLIST X)
                             (CONS (SUB1 (MAXLIST X))
                                   (CONS (ADD1 (MINLIST X))
                                         (DELETE1 (MAXLIST X)
                                                  (DELETE1 (MINLIST X) X)))))
                (OCCURRENCES (MAXLIST X) X))
         T)))
```

Notice that it is dangerous to have a rewrite rule around such as MEMBER-IMPLIES-LISTP, because the conclusion is likely to match many, many terms, thus slowing down the prover. So, we disable this rule as it has accomplished its purpose. The DISABLE event instructs the theorem prover to ignore the indicated rule (which may be a function definition or a rewrite rule) from now on, unless overridden by a later ENABLE or ENABLE-THEORY event or hint.

```
(DISABLE MEMBER-IMPLIES-LISTP)
```

Point 12.

We turn now to the other case in our termination argument, i.e., the lemma `INDUCTION-FN--HELP-2-MAX-OCCURS-ONCE` introduced in Subsection 5.5. Let us try to prove this automatically now; we have some hope, since by now we have proved so many relevant lemmas. The theorem prover runs what seems to be a very long time without giving any output, and inspection of the rewrite stack (see page 28, for explanation of the rewrite stack) shows the prover expanding `OCCURRENCES`. So, we give the `DISABLE` hint shown below.

```
(prove-lemma induction-fn-help-2-max-occurs-once ()
   (let ((x0 (cons (sub1 (maxlist x))
                   (cons (add1 (minlist x))
                         (delete1 (maxlist x)
                                  (delete1 (minlist x) x))))))
    (implies
     (and (equal (sumlist x) (times k (length x)))
          (not (equal (minlist x) (maxlist x)))
          (not (lessp 1 (occurrences (maxlist x) x))))
     (ord-lessp
      (cons (add1 (maxlist x0)) (occurrences (maxlist x0) x0))
      (cons (add1 (maxlist x)) (occurrences (maxlist x) x)))))
 ((disable occurrences minlist maxlist delete1)))
```

Now the prover does give some output, but it attempts to use induction, which is not what we want to happen with such a complicated theorem. Instead, we had hoped for a proof using only simplification (essentially, rewriting and linear arithmetic). The checkpoint tool takes us to the following goal, which as usual we write here using our abbreviation `X0`. (The `LET` construct never appears in actual prover output, but we find it useful in this presentation.)

```
(let ((x0 (cons (sub1 (maxlist x))
                (cons (add1 (minlist x))
                      (delete1 (maxlist x)
                               (delete1 (minlist x) x))))))
  (implies
   (and
    (equal (sumlist x)
           (times k (length x)))
    (not (equal (minlist x) (maxlist x)))
    (not (lessp 1 (occurrences (maxlist x) x)))
    (not (lessp (maxlist x0)
                (maxlist x))))
   (lessp (occurrences (maxlist x) x0)
```

```
          (occurrences (maxlist x) x))))
```

Perhaps it is most natural to look first at the conclusion of this IMPLIES term, which is a perfectly reasonable thing to do, but further inspection leads us to notice that the last hypothesis is false! That is: in the current case, where there is only one occurrence of (MAXLIST X) in X, what we should be trying to prove is that the maximum decreases when we remove (MAXLIST X) from X. So, our real goal seems to prove the lemma INDUCTION-FN-HELP-2-MAX-OCCURS-ONCE-MAIN--LEMMA (see previous level of the proof). Notice that we state it as a rewrite rule, rather than as a linear rule, simply because the linear arithmetic procedure of Nqthm behaves in a way that seems rather mysterious, but rewriting is a very simple notion. As we also explain on page 38, an idiosyncrasy of Nqthm is that when the conclusion of a "rewrite" rule is of the form (LESSP x y) or (NOT (LESSP x y)), then the rule is actually stored as a "linear rule" for use by the linear arithmetic decision procedure, rather than as a rewrite rule. Therefore we write the conclusion as an equality.

We decide to proceed top-down: we first make INDUCTION-FN-HELP-2-MAX-OCCURS-ONCE--MAIN-LEMMA an axiom and we see that it suffices for proving INDUCTION-FN-HELP-2-MAX--OCCURS-ONCE.

Point 13.

When we prove the following lemma

```
(PROVE-LEMMA MAXLIST-NOT-MINLIST-IMPLIES-LISTP ()
             (IMPLIES (NOT (EQUAL (MINLIST X) (MAXLIST X)))
                      (LISTP X)))
```

we can then prove the following improvement of MINLIST-LESS-THAN-MAXLIST-MINUS-1:

```
(PROVE-LEMMA MINLIST-LESS-THAN-MAXLIST-MINUS-1-BETTER (REWRITE)
             (IMPLIES (AND (EQUAL (SUMLIST X) (TIMES K (LENGTH X)))
                           (NOT (EQUAL (MINLIST X) (MAXLIST X))))
                      (LESSP (MINLIST X) (SUB1 (MAXLIST X))))
             ((USE (MAXLIST-NOT-MINLIST-IMPLIES-LISTP))))
```

Why clutter the database? Let's disable the old version.

```
(DISABLE MINLIST-LESS-THAN-MAXLIST-MINUS-1)
```

Point 14.

The BASH command leads us to the following goal:

```
(implies (and (equal (maxlist (delete1 (minlist x) x))
                     (maxlist x))
              (equal (sumlist x)
                     (times k (length x)))
              (not (equal (minlist x) (maxlist x)))
              (not (lessp 1 (occurrences (maxlist x) x)))
              (listp (delete1 (maxlist x)
                              (delete1 (minlist x) x)))
              (not (equal (maxlist (delete1 (maxlist x)
                                            (delete1 (minlist x) x)))
                          0))
              (lessp (minlist x)
                     (sub1 (maxlist (delete1 (maxlist x)
                                             (delete1 (minlist x) x)))))
              (lessp (sub1 (maxlist x))
                     (maxlist (delete1 (maxlist x)
                                       (delete1 (minlist x) x)))))
         (lessp (maxlist (delete1 (maxlist x)
                                  (delete1 (minlist x) x)))
                (maxlist x)))
```

The following inequality chains yield the conclusion from the hypotheses in this term:

```
(maxlist (delete1 (maxlist x) (delete1 (minlist x) x)))
<=
(maxlist (delete1 (maxlist x) x))
<
(maxlist x)
```

The first inequality proves easily:

```
(PROVE-LEMMA MAXLIST-DELETE1-DELETE1 (REWRITE)
             (NOT (LESSP (MAXLIST (DELETE1 B X))
                         (MAXLIST (DELETE1 B (DELETE1 A X))))))
```

Now we turn to the other inequality:

```
(maxlist (delete1 (maxlist x) x)) < (maxlist x)
```

The second lemma below formalizes this inequality, while the first is suggested by the checkpoint tool in order to help prove it (see Point 8 of Appendix B).

```
(PROVE-LEMMA MEMBER-MAXLIST (REWRITE)
             (IMPLIES (NOT (EQUAL (MAXLIST Z) 0))
                      (MEMBER (MAXLIST Z) Z)))

(PROVE-LEMMA LESSP-MAXLIST-DELETE1-MAXLIST (REWRITE)
             (IMPLIES (AND (LESSP 0 (MAXLIST X))
                           (NOT (LESSP 1 (OCCURRENCES (MAXLIST X) X))))
                      (LESSP (MAXLIST (DELETE1 (MAXLIST X) X))
                             (MAXLIST X))))
```

And finally we have:

```
(PROVE-LEMMA INDUCTION-FN-HELP-2-MAX-OCCURS-ONCE-MAIN-LEMMA (REWRITE)
 (IMPLIES
  (AND
   (EQUAL (SUMLIST X) (TIMES K (LENGTH X)))
   (NOT (EQUAL (MINLIST X) (MAXLIST X)))
   (NOT (LESSP 1 (OCCURRENCES (MAXLIST X) X))))
  (EQUAL (LESSP (MAXLIST (CONS (SUB1 (MAXLIST X))
                               (CONS (ADD1 (MINLIST X))
                                     (DELETE1 (MAXLIST X)
                                              (DELETE1 (MINLIST X) X)))))
                (MAXLIST X))
         T)))
```

Point 15.

```
(PROVE-LEMMA INDUCTION-FN-HELP-2 (REWRITE)
  (LET ((X0 (CONS (SUB1 (MAXLIST X))
                  (CONS (ADD1 (MINLIST X))
                        (DELETE1 (MAXLIST X)
                                 (DELETE1 (MINLIST X) X))))))
    (IMPLIES
     (AND (EQUAL (SUMLIST X) (TIMES K (LENGTH X)))
          (NOT (EQUAL (MINLIST X) (MAXLIST X))))
     (ORD-LESSP
      (CONS (ADD1 (MAXLIST X0)) (OCCURRENCES (MAXLIST X0) X0))
```

```
      (CONS (ADD1 (MAXLIST X)) (OCCURRENCES (MAXLIST X) X)))))
 ((USE (INDUCTION-FN-HELP-2-MAX-OCCURS-TWICE)
       (INDUCTION-FN-HELP-2-MAX-OCCURS-ONCE))
  (DISABLE-THEORY T)
  (ENABLE-THEORY GROUND-ZERO)))
```

The last two lines in the hint above instruct the theorem prover to disable all the rules in the database except those that are built in. We have introduced these lines because we have already arranged that the proof follows immediately from the two lemmas in the `USE` hint, and as we pointed out earlier, in such cases it is useful to disable rules that could otherwise get in the way.

Point 16.

A quick inspection of the proof finds the term

```
(sumlist (delete1 (maxlist x) (delete1 (minlist x) x)))
```

that suggests us that we prove the second lemma below (the first was found by using the checkpoint tool when inspecting a failed proof attempt of `SUMLIST-DELETE1`; see for more detail Point 9 in Appendix B).

```
(PROVE-LEMMA SUMLIST-DELETE1-PLUS-VERSION ()
             (EQUAL (PLUS A (SUMLIST (DELETE1 A X)))
                    (IF (MEMBER A X)
                        (SUMLIST X)
                      (PLUS A (SUMLIST X)))))

(PROVE-LEMMA SUMLIST-DELETE1 (REWRITE)
             (EQUAL (SUMLIST (DELETE1 A X))
                    (IF (MEMBER A X)
                        (DIFFERENCE (SUMLIST X) A)
                      (SUMLIST X)))
             ((USE (SUMLIST-DELETE1-PLUS-VERSION))))
```

Point 17.

The base case, as presented by the checkpoint tool when it tries to prove the main lemma, is as follows:

```
(implies (and (equal (sumlist x)
                     (times k (length x)))
```

```
          (equal (minlist x) (maxlist x)))
     (not (lessp (exp k (length x))
                 (prodlist x))))
```

Since every element of the list `X` is the same (modulo `FIX`) if `(MINLIST X)` is equal to `(MAXLIST X)`, it seems reasonable to prove the following theorem, which feels simpler since it avoids linear reasoning. Besides, this appears to be the sort of theorem that yields more easily to proof when it is strengthened, since (sometimes) the prover is good at using equalities.

```
(prove-lemma main-lemma-base-case (rewrite)
             (implies (and (equal (sumlist x) (times k (length x)))
                           (equal (minlist x) (maxlist x)))
                      (equal (exp k (length x)) (prodlist x))))
```

The proof fails and the checkpoint tool suggests to us the following lemma (see Point 10 in Appendix B):

```
(PROVE-LEMMA MAXLIST-0-IS-SUMLIST-0 (REWRITE)
             (EQUAL (EQUAL (MAXLIST X) 0) (EQUAL (SUMLIST X) 0)))
```

The proof of `MAIN-LEMMA-BASE-CASE` still fails, yielding the checkpointed goal of:

```
(implies (and (equal (maxlist (cdr x)) (car x))
              (listp x)
              (listp (cdr x))
              (not (equal (sumlist (cdr x))
                          (times k (length (cdr x)))))
              (equal (plus (car x) (sumlist (cdr x)))
                     (plus k (times k (length (cdr x)))))
              (not (lessp (car x) (car x)))
              (equal (minlist (cdr x)) (car x)))
         (equal (times k (exp k (length (cdr x))))
                (times (car x) (prodlist (cdr x)))))
```

This appears to us to be kind of awkward to reason about. Maybe we should go back to thinking of a high-level strategy.

We may be tempted to prove that under the hypotheses of the base case, every member of `X` is equal to `K`. The problem is: how do we get the theorem prover to use that information? Note

that the function symbol MEMBER does not appear in the goal, nor do we imagine any way for it to be introduced (by rewriting, say) into the goal.

But another high-level strategy is to see if there is an "obviously simpler" version of the theorem to be proved that is still a theorem. Here is such a version, which is simpler in the sense that it does not mention K.

```
(PROVE-LEMMA MAIN-LEMMA-BASE-CASE-LEMMA-1 ()
             (IMPLIES (EQUAL (MINLIST X) (MAXLIST X))
                      (EQUAL (EXP (MINLIST X) (LENGTH X)) (PRODLIST X))))
```

Now we try the following. (Why? It's a bit hard to completely reconstruct our thinking on this particular point, but it was probably that MAIN-LEMMA-BASE-CASE follows quickly from these, if you ignore the FIX in the conclusion below and the first hypothesis (LISTP X) below. Why are they there? Without FIX, the lemma below simply wouldn't be true; what if X is a non-empty list of Booleans? Without the first hypothesis the lemma below also would fail to be true; consider the case that X is the empty list and K is a positive integer. Fortunately, we expect that when we direct use of the lemma below and above in a USE hint, the cases where (LISTP X) or (NUMBERP X) fails will fall out trivially.)

```
(prove-lemma main-lemma-base-case-lemma-2 ()
             (implies (and (listp x)
                           (equal (sumlist x) (times k (length x)))
                           (equal (minlist x) (maxlist x)))
                      (equal (minlist x) (fix k))))
```

Unfortunately, the proof fails. Using the checkpoint tool we find a goal that the following lemma should help to prove (see Point 11 in Appendix B).

```
(PROVE-LEMMA MAIN-LEMMA-BASE-CASE-LEMMA-2-LEMMA ()
             (IMPLIES (EQUAL (MINLIST X) (MAXLIST X))
                      (EQUAL (SUMLIST X) (TIMES (MINLIST X) (LENGTH X)))))
```

Now we try to prove MAIN-LEMMA-BASE-CASE-LEMMA-2 with this hint:

```
(use (main-lemma-base-case-lemma-2-lemma))
```

It fails with the following checkpoint:

```
(implies (and (equal (times k (length x))
                     (sumlist x))
              (listp x)
              (equal (times (length x) (maxlist x))
                     (sumlist x))
              (equal (minlist x) (maxlist x))
              (numberp k))
         (equal (maxlist x) k))
```

The first and third hypotheses should obviously yield the desired result, since we know that `(LENGTH X)` is not zero, by the second hypothesis. It seems that controlling equality reasoning such as this is one of the hardest things to do in Nqthm. So, let's just take a brutally inelegant approach.

Here is a lemma that yields the goal above (using `N = (LENGTH X)`), but Nqthm cannot prove it.

```
(prove-lemma main-lemma-base-case-lemma-2-hack ()
             (implies (and (equal (times k n) (sumlist x))
                           (not (zerop n))
                           (equal (times n (maxlist x)) (sumlist x)))
                      (equal k (maxlist x))))
```

So, let us simplify this "brutal hack" lemma by generalizing `(SUMLIST X)` and `(MAXLIST X)` to `SUMLIST` and `MAXLIST`, respectively. We also notice that there is no guarantee that `K` is a number, so we apply `FIX` to it, and also to the new variable `MAXLIST` since, unlike `(MAXLIST X)`, we have no guarantee that `MAXLIST` is a number. Who knows, maybe this generalization will help. Again, this kind of equality reasoning is often a hit-or-miss proposition in our Nqthm experience.

```
(PROVE-LEMMA MAIN-LEMMA-BASE-CASE-LEMMA-2-HACK ()
             (IMPLIES (AND (EQUAL (TIMES K N) SUMLIST)
                           (NOT (ZEROP N))
                           (EQUAL (TIMES N MAXLIST) SUMLIST))
                      (EQUAL (FIX K) (FIX MAXLIST))))
```

We try again `MAIN-LEMMA-BASE-CASE-LEMMA-2`, but the proof attempt fails again. Using the checkpoint tool we see that we need the lemma below (see Point 12 in Appendix B). Note that once again, we use an unconditional rewrite rule.

```
(PROVE-LEMMA EQUAL-LENGTH-0 (REWRITE)
             (EQUAL (EQUAL (LENGTH X) 0) (NOT (LISTP X))))

(PROVE-LEMMA MAIN-LEMMA-BASE-CASE-LEMMA-2 ()
             (IMPLIES (AND (LISTP X)
                           (EQUAL (SUMLIST X) (TIMES K (LENGTH X)))
                           (EQUAL (MINLIST X) (MAXLIST X)))
                      (EQUAL (MINLIST X) (FIX K)))
             ((USE (MAIN-LEMMA-BASE-CASE-LEMMA-2-LEMMA)
                   (MAIN-LEMMA-BASE-CASE-LEMMA-2-HACK
                    (SUMLIST (SUMLIST X))
                    (MAXLIST (MAXLIST X))
                    (N (LENGTH X))
                    (K K)))))
```

And finally we are able to prove the lemma `MAIN-LEMMA-BASE-CASE`.

Point 18.

Let us enter the interactive "proof-checker," with the theorem from `MAIN-LEMMA` as our goal.

```
(verify (let ((n (length x)))
          (implies (equal (sumlist x) (times k n))
                   (not (lessp (exp k n) (prodlist x))))))
```

We apply the induction scheme to the goal and after some simplifications (see for further details Point 13 in Appendix B) we obtain a subgoal that contains a term of the form `(LENGTH (DELETE1 ...))`. The following lemma simplifies such terms by eliminating this call of `DELETE1`.

```
(PROVE-LEMMA LENGTH-DELETE1 (REWRITE)
             (EQUAL (LENGTH (DELETE1 A X))
                    (IF (MEMBER A X)
                        (SUB1 (LENGTH X))
                      (LENGTH X))))
```

Then we re-enter the proof-checker loop with `(VERIFY)`. We type the `REPLAY` command to re-run all the commands in the presence of the new lemma `LENGTH-DELETE1`. We find a goal (see Point 14 in Appendix B) that can prove if we prove:

```
(prove-lemma member-minlist (rewrite)
```

```
            (implies (listp x) (member (minlist x) x)))
```

The proof fails, and the checkpoint tool yields the goal shown below.

```
(implies (and (not (numberp (car x)))
              (not (listp (cdr x))))
         (not (listp x)))
```

Hmmmm.... what does `(NUMBERP (CAR X))` have to do with this? Or we could ask: is the lemma `MEMBER-MINLIST` actually `NOT` a theorem? *Aha* – if `X` is a one-element list containing a non-number, then `(MINLIST X)` is `0`, which is not a member of `X`. We can ask: where did the term `(MEMBER (MINLIST X) X)` come from in the goal above? It seems likely that it came from an application of the lemma `LENGTH-DELETE1` displayed above, which introduces an `IF` expression whose first argument is a `MEMBER` term. And now perhaps we see: the induction that we are performing removes one occurrence of `(MINLIST X)` from `X`, but if a Boolean (say) is in `X` but `0` is not in `X`, then in fact we haven't really removed anything! To recall the definition of `MINLIST`:

```
(defn minlist (x)
  (if (listp x)
      (if (listp (cdr x))
          (min (car x)
               (minlist (cdr x)))
        (fix (car x)))
    0))
```

So, when we apply `FIX` to `(CAR X)`, we obtain a value that may not actually belong to `X`, and hence our induction is "odd".

Point 19.

When we try to prove the analogous theorem for `MAXLIST`,

```
(prove-lemma member-maxlist (rewrite)
             (implies (and (listp x)
                           (numberp-listp x))
                      (member (maxlist x) x)))
```

the system responds with:

```
ERROR:  Name currently in use: MEMBER-MAXLIST.
```

So, apparently we proved such a lemma before – but it must have had some hypothesis stronger than `(LISTP X)`, for the same reason that the analogous theorem about `MINLIST` needed a stronger hypothesis. Here is a small bit of interaction, then, with the system.

```
>(ppe 'member-maxlist) ;; Pretty-print event MEMBER-MAXLIST.
```

The system responds by printing out the following:[12]

```
(prove-lemma member-maxlist
             (rewrite)
             (implies (not (equal (maxlist z) 0))
                      (member (maxlist z) z)))
```

Perhaps this lemma suffices; if not, then we can prove a more direct analog to `MEMBER-MINLIST`. Probably in fact this existing version really is good enough, since we are proceeding under the assumption that `(MINLIST X)` and `(MAXLIST X)` are distinct, and hence the hypothesis is likely to hold when the prover is ready to apply this lemma.

Point 20.

We notice that the prover applies the lemma `MAIN-LEMMA-BASE-CASE`, which is good. However, the proof fails; the prover pushes a goal for proof by induction, and our `DO-NOT-INDUCT` hint prevents the proof attempt from continuing.

The checkpoint tool leads us to a goal that should be provable once we prove the following lemma (see Point 15 in Appendix B):

```
(PROVE-LEMMA NUMBERP-LISTP-DELETE1 (REWRITE)
             (IMPLIES (NUMBERP-LISTP X)
                      (NUMBERP-LISTP (DELETE1 A X))))
```

But the proof of the main lemma still fails. The checkpoint tool leads us to a goal that will be proved if we show that (see for more details Point 16 of Appendix B)

```
(add1 (plus (minlist x) (sub1 (maxlist x)))) <= (sumlist x).
```

[12] actually in upper case, but that's not important, and it might be confusing to print uppercase output here, given the convention we set forth in Subsection 1.2

So, let us prove the lemmas ADD1-PLUS-SUB1-SECOND and SUMLIST-GEQ-MINLIST-PLUS--MAXLIST below:

```
(PROVE-LEMMA ADD1-PLUS-SUB1-SECOND (REWRITE)
             (IMPLIES (NOT (ZEROP Y))
                      (EQUAL (ADD1 (PLUS X (SUB1 Y))) (PLUS X Y))))
```

To prove SUMLIST-GEQ-MINLIST-PLUS-MAXLIST below, the checkpoint tool has suggested to us the following two lemmas (details omitted).

```
(PROVE-LEMMA SUMLIST-GEQ-MAXLIST (REWRITE)
             (NOT (LESSP (SUMLIST X) (MAXLIST X))))

(PROVE-LEMMA SUMLIST-GEQ-MINLIST (REWRITE)
             (NOT (LESSP (SUMLIST X) (MINLIST X))))
```

And then the proof succeeds:

```
(PROVE-LEMMA SUMLIST-GEQ-MINLIST-PLUS-MAXLIST (REWRITE)
             (IMPLIES (NOT (EQUAL (MINLIST X) (MAXLIST X)))
                      (NOT (LESSP (SUMLIST X)
                                  (PLUS (MINLIST X) (MAXLIST X))))))
```

At this point we wonder if we have perhaps proved enough lemmas to make the main lemma's proof succeed, so we try it again. But the theorem prover reports:

```
Error: Bind stack overflow.
Fast links are on: do (use-fast-links nil) for debugging
Error signalled by PROVE-LEMMA-FN.
Broken at APPLY.  Type :H for Help.
>>
```

So, we submit the form (MAINTAIN-REWRITE-PATH T) and try again, letting the stack overflow and then inspecting the rewrite path.

After the interaction with the prover, we decide to disable the (recursive) definition of PLUS (for further details see Point 17 in Appendix B).

```
(DISABLE PLUS)
```

So, we execute the form `(MAINTAIN-REWRITE-PATH NIL)` to get the theorem prover back up to full speed, and then try the main lemma again. It fails, and the second checkpoint we visit is:

```
(implies (and (equal (sumlist x)
                     (plus (maxlist x) (minlist x)))
              (equal (sumlist x)
                     (times k (length x)))
              (not (equal (minlist x) (maxlist x)))
              (not (equal (sumlist x) 0))
              (listp x)
              (not (equal (length x) 1))
              (not (lessp (maxlist x)
                          (difference (sumlist x) (minlist x))))
              (not (equal (sumlist x)
                          (plus k (times k (sub1 (length x))))))
              (numberp-listp x))
         (not (lessp (exp k (length x))
                     (prodlist x))))
```

The second hypothesis and the next-to-last hypothesis clearly contradict each other (if we know that `(LENGTH X)` is non-zero, which follows from `(LISTP X)`). So, let us prove a "folding" rule for `TIMES` that is analogous to the rule we proved for `PLUS`, and having learned our lesson earlier, follow that by disabling `TIMES` in order to prevent the rewriter from looping.

```
(PROVE-LEMMA PLUS-TIMES-SUB1-SECOND (REWRITE)
             (IMPLIES (NOT (ZEROP Y))
                      (EQUAL (PLUS X (TIMES X (SUB1 Y))) (TIMES X Y))))

(DISABLE TIMES)
```

We try the main lemma again. The second checkpoint we visit is as follows.

```
(implies (and (equal (sumlist x)
                     (plus (maxlist x) (minlist x)))
              (not (equal (minlist x) (maxlist x)))
```

```
            (not (equal (sumlist x) 0))
            (equal (length x) 1)
            (not (lessp (maxlist x)
                        (difference (sumlist x) (minlist x))))
            (numberp-listp x)
            (listp x))
       (not (lessp (sumlist x) (prodlist x))))
```

Here, the second and fourth hypotheses are clearly contradictory. Here is the appropriate rewrite rule.

```
(PROVE-LEMMA MINLIST-NOT-MAXLIST-IMPLIES-LENGTH-AT-LEAST-2 (REWRITE)
             (IMPLIES (NOT (EQUAL (MINLIST X) (MAXLIST X)))
                      (LESSP 1 (LENGTH X))))
```

Again, we obtain a binding stack overflow (but much later in the proof this time!). Rather than inspect the rewrite stack, we ask, executing the command `(GET 'PLUS 'LEMMAS)`, whether there are lemmas that act like the definition of `PLUS`, thus looping with the rewrite rule `ADD1--PLUS-SUB1-SECOND`. We probably need to do a similar thing for `TIMES` as well.

```
(DISABLE PLUS-ADD1-ARG1)
(DISABLE PLUS-ADD1-ARG2)
(DISABLE TIMES-ADD1)
```

But still, the stack overflows (see Point 18 of Appendix A for the details of the interaction relative to this comment). The rewrite path suggests us to do the following:

```
(DISABLE MAXLIST-0-IS-SUMLIST-0)
```

We've made real progress (but the proof of the main lemma still fails). This time, the checkpoint tool yields:

```
(implies
 (and
  (equal (sumlist x)
         (times k (length x)))
  (not (equal (minlist x) (maxlist x)))
  (not (equal (maxlist x) 0))
```

```
  (listp x)
  (not (equal (length x) 1))
  (not (lessp (times k (exp k (sub1 (length x))))
              (times (add1 (minlist x))
                     (sub1 (maxlist x))
                     (prodlist (delete1 (maxlist x)
                                        (delete1 (minlist x) x))))))
  (numberp-listp x))
 (not (lessp (times k (exp k (sub1 (length x))))
             (prodlist x))))
```

So, let us prove a lemma simplifying the large `TIMES` term in the next-to-last hypothesis. In a sense, this is intuitively the heart of the arithmetic argument, as explained on page 45 and in the next point below.

Point 21.
Consider the term:

```
(times (add1 (minlist x))
       (sub1 (maxlist x))
       (prodlist (delete1 (maxlist x)
                          (delete1 (minlist x) x))))
```

Using the sequence of equations in the main body of the proof (page 45), we write a lemma of the form:

```
(prove-lemma product-of-modified-list (rewrite)
             (implies ???
                      (equal (times (add1 (minlist x))
                                    (sub1 (maxlist x))
                                    (prodlist (delete1 (maxlist x)
                                                       (delete1
                                                        (minlist x)
                                                        x))))
                             ???)))
```

In fact, there is nothing really special about `(MINLIST X)` and `(MAXLIST X)` here. As usual, it is preferable to state syntactically simple theorems – they are often easier to prove, and often more general. So we generalize `(MINLIST X)` and `(MAXLIST X)` to new variables `MAX` and `MIN`, respectively.

Do we need the hypothesis that `X` is a list of numbers? Probably not – at least, we see no reason to believe that we would need that hypothesis. Maybe, though, such a hypothesis could eliminate the need for some of the other hypotheses. But at the moment, that doesn't seem to be the case either.

```
(prove-lemma product-of-modified-list (rewrite)
             (implies (and (not (equal min max))
                           (member min x)
                           (member max x)
                           (not (zerop max)))
                      (let ((rest (prodlist (delete1 max
                                                     (delete1 min x)))))
                        (equal (times (add1 min)
                                      (sub1 max)
                                      rest)
                               ;; prodlist(x) + ([max - min - 1] * rest)
                               (plus (prodlist x)
                                     (times (difference max (add1 min))
                                            rest))))))
```

The proof fails, and when we look at a checkpoint we get very confused. So, let us invent a high-level strategy. Our thinking goes something like this. We tend to prefer rewrite rules that are clean, elegant, and simple. Thus, in order to simplify terms appearing above, we are tempted to formulate a rewrite rule whose left hand side is `(PRODLIST (DELETE1 A X))`. However, the rule we imagine would have `QUOTIENT` on the right-hand side, and our experience is that `QUOTIENT` is a more complicated function to reason about than `TIMES`. Moreover, `QUOTIENT` does not already appear in the problem, and introducing a new function can complicate the situation. So, we formulate such a lemma using `TIMES` in a way that avoids the need to introduce `QUOTIENT` into the statement, as follows.

```
(PROVE-LEMMA TIMES-PRODLIST-DELETE1 (REWRITE)
             (IMPLIES (MEMBER A X)
                      (EQUAL (TIMES A (PRODLIST (DELETE1 A X)))
                             (PRODLIST X))))

(PROVE-LEMMA PRODUCT-OF-MODIFIED-LIST-LEMMA (REWRITE)
             (IMPLIES (AND (NOT (EQUAL MIN MAX))
                           (MEMBER MIN X)
                           (MEMBER MAX X))
                      (LET ((REST (PRODLIST (DELETE1 MAX
                                                     (DELETE1 MIN X)))))
```

```
                (EQUAL (TIMES MAX MIN REST)
                       (PRODLIST X)))))
```

Originally the first two arguments of TIMES in the left hand side of the equation above were switched. But we found later that this prohibited application of the rule, perhaps because some kind of commutativity lemma had been applied.

Let us disable the preceding rule, since it has already served its purpose and we don't want it to get in the way.

```
(DISABLE TIMES-PRODLIST-DELETE1)
```

Now if we can prove the following rewrite rule, then our goal PRODUCT-OF-MODIFIED-LIST should fall right out, by the application of the following rule to the left side of the conclusion of PRODUCT-OF-MODIFIED-LIST, followed by application of the rewrite rule PRODUCT-OF-MODIFIED--LIST-LEMMA shown above. The ENABLE hint below is intended to re-enable the definitions and rewrite rules that we recently disabled, since we may need them for this particular proof. The DISABLE hint is intended to turn off new rules that we feel aren't necessary for the present proof – they were in fact invented for specialized purposes only, and they seem to have roughly opposite effects from TIMES-ADD1 and PLUS-ADD1-ARG2. As explained earlier, if all these rules are enabled at once, then the rewriter might loop.

```
(PROVE-LEMMA PRODUCT-OF-MODIFIED-LIST-LEMMA-2 (REWRITE)
             (IMPLIES (AND (LESSP MIN MAX)
                           (NOT (ZEROP MAX)))
                      (EQUAL (TIMES (ADD1 MIN)
                                    (SUB1 MAX)
                                    REST)
                             (PLUS (TIMES MIN MAX REST)
                                   (TIMES (DIFFERENCE MAX (ADD1 MIN))
                                          REST))))
             ((ENABLE TIMES-ADD1 TIMES
                      PLUS-ADD1-ARG1 PLUS-ADD1-ARG2 PLUS)
              (DISABLE PLUS-TIMES-SUB1-SECOND
                       ADD1-PLUS-SUB1-SECOND)))
```

So now we try PRODUCT-OF-MODIFIED-LIST again, but with the hypothesis of (NOT (EQUAL MIN MAX)) replaced by (LESSP MIN MAX), since that was necessary for the lemma immediately above. The proof still fails, because our (conditional) rewrite rules were not applied as expected. When that happens, it is often useful to enter the proof-checker and try to see what the problem is; probably the prover was unable to relieve some of these lemmas' hypotheses.

We investigate with Pc-Nqthm, and fix PRODUCT-OF-MODIFIED-LIST-LEMMA as indicated above (see the remark about commutativity). And then, we are able to prove the theorem "by hand" in Pc-Nqthm. In Point 19 of Appendix B we show the transcript of the session. However, this investigation is rather useless, really. Usually such investigations with Pc-Nqthm lead to failure, but with a good idea of just why the failure occurs. In this case the proof succeeds in Pc-Nqthm, so we don't really have any useful information that helps us to find an Nqthm proof except that, at least, what we are trying to prove is really a theorem.

So, let us see what is wrong, using the commands:

```
(maintain-rewrite-path t)

(break-lemma 'PRODUCT-OF-MODIFIED-LIST-LEMMA-2)

(break-lemma 'PRODUCT-OF-MODIFIED-LIST-LEMMA)
```

When we inspect the rewrite path (details omitted), we find that the problem is that the prover does not quite realize the following obvious fact. In fact we tried first to prove this as a rewrite rule, but it was not applied. So rather than think hard about this problem, we simply prove it as a non-rewrite-rule theorem and supply it in a USE hint.

```
(PROVE-LEMMA POSITIVE-IMPLIES-NUMBERP ()
             (IMPLIES (LESSP MIN MAX) (NUMBERP MAX)))
```

We can "unbreak" the lemmas with the command (UNBREAK-LEMMA), and now the lemma PRODUCT-OF-MODIFIED-LIST succeeds.

Point 22.

Here are events leading to the proof of the theorem from MAIN-LEMMA, as promised in Subsection 5.10.

```
(DEFN SCALAR-PRODUCT (SCALAR LST)
  (IF (LISTP LST)
      (CONS (TIMES SCALAR (CAR LST)) (SCALAR-PRODUCT SCALAR (CDR LST)))
    NIL))

(PROVE-LEMMA SUMLIST-SCALAR-PRODUCT (REWRITE)
             (EQUAL (SUMLIST (SCALAR-PRODUCT SCALAR LST))
                    (TIMES SCALAR (SUMLIST LST))))

(PROVE-LEMMA PRODLIST-SCALAR-PRODUCT (REWRITE)
```

```
              (EQUAL (PRODLIST (SCALAR-PRODUCT SCALAR LST))
                     (TIMES (EXP SCALAR (LENGTH LST))
                            (PRODLIST LST))))

(PROVE-LEMMA LENGTH-SCALAR-PRODUCT (REWRITE)
              (EQUAL (LENGTH (SCALAR-PRODUCT N A))
                     (LENGTH A)))
```

Point 23.
And now, our main theorem!

```
(prove-lemma main ()
              (implies (numberp-listp a)
                       (let ((n (length a)))
                         (geq (exp (sumlist a) n)
                              (times (exp n n) (prodlist a)))))
              ((use (main-lemma (k (sumlist a))
                                (x (scalar-product (length a) a))))))
```

But it fails. Using the checkpoint tool we see that we need

```
(implies (and (not (numberp-listp (scalar-product (length a) a)))
              (numberp-listp a))
         (not (lessp (exp (sumlist a) (length a))
                     (times (prodlist a)
                            (exp (length a) (length a))))))
```

Obviously we need one final lemma that states that if `(NUMBERP-LISTP A)` is true then also `(NUMBERP-LISTP (SCALAR-PRODUCT N A))` is true. This is necessary because we added the `NUMBERP-LISTP` hypothesis that wasn't there at the start of this effort.

B The rest of the details of the mechanized proof

Point 1.
The output from our attempt to prove `MINLIST-NOT-MAXLIST-IMPLIES-MINLIST-LESSP--AVERAGE-LEMMA` contains the following checkpoint:

```
(implies (and (listp x)
```

```
            (listp (cdr x))
            (equal (minlist (cdr x))
                   (maxlist (cdr x)))
            (not (lessp (car x) (maxlist (cdr x))))
            (numberp (car x))
            (not (equal (maxlist (cdr x)) (car x))))
       (lessp (plus (maxlist (cdr x))
                    (times (length (cdr x))
                           (maxlist (cdr x))))
              (plus (car x) (sumlist (cdr x)))))
```

Why is this goal true? Well, the 4th and 6th (last) hypotheses imply that `(CAR X)` > `(MAXLIST (CDR X))`, so the conclusion follows immediately if we can see that

```
(sumlist (cdr x)) >= (times (length (cdr x)) (maxlist (cdr x)))
```

But by the third hypothesis, this is equivalent to

```
(sumlist (cdr x)) >= (times (length (cdr x)) (minlist (cdr x)))
```

and that in turn is an obvious property of `MINLIST`. In fact this property has nothing to do with `CDR`; so, we generalize by hand to obtain the following lemma. Note that since the `"naturals"` library contains a rewrite rule stating the commutativity of `TIMES`, and since `L` precedes `M` alphabetically, then `LENGTH` precedes `MINLIST` alphabetically and therefore we write `(TIMES (LENGTH X) (MINLIST X))` below rather than `(TIMES (MINLIST X) (LENGTH X))`, since Nqthm's rewriter is more likely to present terms that match the former pattern than the latter.

```
(prove-lemma minlist-main-property (rewrite)
             (not (lessp (sumlist x)
                         ;; note l < m in alphabetic order (see line below)
                         (times (length x) (minlist x)))))
```

When we try to prove `MINLIST-MAIN-PROPERTY`, the proof fails, and the checkpoint tool points us to the goal shown below.

```
(implies (and (listp x)
              (listp (cdr x))
              (not (lessp (sumlist (cdr x))
```

```
                    (times (length (cdr x))
                           (minlist (cdr x)))))
          (lessp (car x) (minlist (cdr x)))
          (numberp (car x)))
     (not (lessp (sumlist (cdr x))
                 (times (car x) (length (cdr x))))))
```

When we stare at such a goal, a good strategy is to try to prove it "by hand"; if the proof fails, then sometimes that helps us formulate a counterexample. Anyhow, we see how to prove the goal above by hand as follows (reading "`(NOT (LESSP A B))`" as "`A` $>=$ `B`"). Since (by the third hypothesis)

```
(sumlist (cdr x)) >= (times (length (cdr x)) (minlist (cdr x))))
```

then since `(MINLIST (CDR X))` $>$ `(CAR X)` (by the fourth hypothesis), we have (by monotonicity of `TIMES` in its second argument):

```
(sumlist (cdr x)) >= (times (length (cdr x)) (car x)))
```

which (by commutativity of `TIMES`) is exactly the conclusion.

But: The prover couldn't see this. On the other hand, this was Case 3.2, and Case 3.1, Case 2, and Case 1 all succeeded, leaving only the final goal from Case 3.2 to be proved. So: we decide to focus entirely on the problem of getting Nqthm to see the line of inequality reasoning (which is not quite linear) presented above. In fact, the goal resulting from Case 3.2 that Nqthm pushes for proof by induction is essentially the lemma `LESSP-TIMES-PRESERVED-IN-FIRST-ARG` (see previous appendix) – we simply clean it up a bit to avoid needless `NUMBERP` hypotheses (but probably that was not necessary).

Point 2.

Using the checkpoint tool we find:

```
(implies (and (listp x)
              (equal 0 (car x))
              (not (lessp 1 (occurrences 0 (cdr x))))
              (equal (maxlist (cdr x)) 0)
              (member 0 (cdr x))
              (not (equal any-element 0))
              (listp (delete1 any-element (cdr x))))
         (equal (maxlist (delete1 any-element (cdr x)))
                0))
```

The above will be taken care of once we prove that

```
(maxlist (delete1 a lst)) <= (maxlist lst).
```

Another checkpoint is:

```
(implies (and (listp x)
              (not (lessp (car x) (maxlist (cdr x))))
              (numberp (car x))
              (not (lessp 1
                          (occurrences (maxlist (cdr x))
                                       (cdr x))))
              (member (car x) (cdr x))
              (not (equal any-element (car x)))
              (listp (delete1 any-element (cdr x)))
              (lessp (car x)
                     (maxlist (delete1 any-element (cdr x)))))
         (equal (maxlist (delete1 any-element (cdr x)))
                (car x)))
```

The reasoning for this case is as follows:

```
(car x)
 >= {by second hypothesis}
(maxlist (cdr x))
 >= {by the same lemma that we will prove above}
(maxlist (delete1 any-element (cdr x)))
 >  {by last hypothesis}
(car x)
```

which is a contradiction.

In fact, probably many Nqthm users would not even look for a second checkpoint, as we did above. Instead, they would try to prove a lemma that would eliminate the first checkpoint, and then hope that this same lemma would also eliminate other checkpoints. If we had followed that methodology, we could have saved ourselves the trouble of carrying out the argument that we just did, because the lemma proved for the first checkpoint would have eliminated the second one as well.

So, let us prove that lemma right now –

```
(maxlist (delete1 a lst)) <= (maxlist lst).
```

– and stop our investigation of the failed proof.

Point 3.
We find the following checkpoint:

```
(implies (and (listp x)
              (not (lessp (car x) (maxlist (cdr x))))
              (numberp (car x))
              (not (lessp 1
                          (occurrences (maxlist (cdr x))
                                       (cdr x))))
              (member (car x) (cdr x)))
         (equal (maxlist (cdr x)) (car x)))
```

Now,

```
(car x) >= (maxlist (cdr x)))
```

by the second hypothesis, and yet

```
(car x) <= (maxlist (cdr x)))
```

by the last hypothesis. The conclusion follows (subtle point: because we know that `(CAR X)` is a number, by the third hypothesis).

Point 4.
The checkpoint tool leads us to consider the subgoal shown below.

```
(implies (and (equal (sumlist x)
                     (times k (length x)))
              (not (equal (minlist x) (maxlist x)))
              (lessp 1 (occurrences (maxlist x) x))
              (not (listp (delete1 (maxlist x)
                                   (delete1 (minlist x) x))))
              (not (equal (maxlist x) 0))
              (not (equal (maxlist x) 1))
```

```
            (not (lessp (sub1 (sub1 (maxlist x)))
                        (minlist x))))
       (equal (sub1 (maxlist x))
              (maxlist x)))
```

The fourth hypothesis is contradictory! That is, we should be able to prove

```
(listp (delete1 (maxlist x) (delete1 (minlist x) x)))
```

from the other hypotheses. Informal argument: since `(MINLIST X)` and `(MAXLIST X)` are distinct, we know that after removing them both from `X`, there is still at least one occurrence of `(MAXLIST X)`, by the third hypothesis. In fact, the lemma `OCCURRENCE-IMPLIES-LISTP` (see previous level of the proof) allows the prover to simplify terms of the form `(LISTP (DELETE1 ....))`, and if we are lucky, that will be all that we need.

Point 5.

The checkpoint tool suggests that we consider the following goal:

```
(implies (and (equal (sumlist x)
                     (times k (length x)))
              (not (equal (minlist x) 0))
              (lessp 1 (occurrences 0 x)))
         (not (equal (maxlist x) 0)))
```

This is obvious if only we know that `(MAXLIST X)` >= `(MINLIST X)`. How come we don't already know that somehow, given that the hypothesis of the lemma we are proving states that `(MAXLIST X)` and `(MINLIST X)` are distinct? Well, "distinct" doesn't specify an order.

Point 6.

The checkpoint tool leads us to the following goal:

```
(implies (and (equal (sumlist x)
                     (times k (length x)))
              (not (equal (minlist x) (maxlist x)))
              (lessp 1 (occurrences (maxlist x) x))
              (not (equal (maxlist x)
                          (sub1 (maxlist x))))
              (equal (maxlist x) (add1 (minlist x)))
              (not (member (maxlist x)
                           (delete1 (minlist x) x))))
```

```
        (lessp (occurrences (maxlist x) x)
               (sub1 (occurrences (maxlist x) x))))
```

The last hypothesis is clearly false: that is,

```
(member (maxlist x) (delete1 (minlist x) x))
```

is true, since

```
(member (maxlist x) x)
```

is true and `(MAXLIST X)` does not equal `(MINLIST X)`.

Point 7.
Consider the following goal:

```
(implies (and (equal (sumlist x)
                     (times k (length x)))
              (not (equal (minlist x) (maxlist x)))
              (lessp 1 (occurrences (maxlist x) x))
              (not (equal (maxlist x)
                          (sub1 (maxlist x))))
              (equal (maxlist x) (add1 (minlist x)))
              (member (maxlist x) x))
         (lessp (occurrences (maxlist x) x)
                (occurrences (maxlist x) x)))
```

The next-to-last (fifth) hypothesis seems in contradiction of the lemma `MINLIST-LESS-THAN--MAXLIST-MINUS-1` (see page 38, for the form of this lemma). What is wrong? Perhaps all we need to do is prove that in the present context, `(LISTP X)` is true.

Point 8. We'd like to have a hypothesis of

```
(not (lessp 1 (occurrences (maxlist x) x)))
```

in the following lemma, but then the lemma would be false for the empty list `X`. It is perhaps natural, then, to use

```
(equal 1 (occurrences (maxlist x) x))
```

as that hypothesis instead; but still, we can get a counterexample using a list whose maximum is `0` (which would have to be of length `1`, actually). Since we are ultimately interested in the case that `(MAXLIST X)` and `(MINLIST X)` are distinct, we know that `(MAXLIST X)` will be positive; so we use that in our hypothesis.

```
(prove-lemma lessp-maxlist-delete1-maxlist (rewrite)
             (implies (and (lessp 0 (maxlist x))
                           (not (lessp 1 (occurrences (maxlist x) x))))
                      (lessp (maxlist (delete1 (maxlist x) x))
                             (maxlist x))))
```

The proof fails, and the first checkpoint we visit after the start of the induction doesn't give us much help, but the one below it,

```
(implies (and (listp z)
              (lessp (maxlist (delete1 (maxlist z) z))
                     (maxlist z))
              (not (equal (maxlist z) 0))
              (not (member (maxlist z) z)))
         (lessp (maxlist z) (maxlist z)))
```

helps a great deal! The following lemma formalizes the idea that the final hypothesis above is false.

```
(prove-lemma member-maxlist (rewrite)
             (implies (listp z) (member (maxlist z) z)))
```

However, the proof fails, and a moment's reflection tells us why: `Z` could be a list of non-numbers with maximum `0` which, therefore, does not belong to the list! However, the checkpoint shown above gives us another hypothesis that we can use instead. We formalize the appropriate lemma (see previous appendix) and now, the proof of `LESSP-MAXLIST-DELETE1-MAXLIST` succeeds.

Point 9.

We try to prove `SUMLIST-DELETE1` (see page 60). The proof fails, and the checkpoint tool yields the following goal (after induction has started), which we find confusing.

```
(implies (and (lessp (sumlist (cdr x)) a)
              (listp x)
              (not (equal a (car x)))
              (member a (cdr x))
              (equal (sumlist (delete1 a (cdr x)))
                     0)
              (not (lessp a (sumlist (cdr x))))
              (numberp (car x)))
         (equal (car x)
                (difference (plus (car x) (sumlist (cdr x)))
                            a)))
```

In fact, the fifth hypothesis –

```
(equal (sumlist (delete1 a (cdr x))) 0)
```

– implies (together with the fourth hypothesis) that `(SUMLIST (CDR X))` equals `(FIX A)`, which contradicts the first hypothesis. BUT, that seems to be a rather complicated line of reasoning; at least, that's our intuition. So, let us formulate this lemma first in terms of `PLUS` rather than `DIFFERENCE`, deriving the `DIFFERENCE` form as a corollary. It is unfortunate, but nevertheless true, that Nqthm is sensitive to the way theorems are stated. In this case, the problem is probably that `DIFFERENCE` has a more complicated pattern of recursion than does `PLUS`.

Point 10.

The checkpoint tool leads us to the following goal:

```
(implies (and (listp x)
              (listp (cdr x))
              (not (equal (sumlist (cdr x))
                          (times k (length (cdr x)))))
              (equal (sumlist (cdr x))
                     (plus k (times k (length (cdr x)))))
              (equal (maxlist (cdr x)) 0)
              (not (numberp (car x)))
              (equal (minlist (cdr x)) 0)
              (not (equal k 0)))
         (not (numberp k)))
```

This goal is perhaps sufficiently complicated so that we are tempted to give up on proving `MAIN-LEMMA-BASE-CASE` directly by induction, and instead using a higher level strategy to view

it as a consequence of other lemmas. That is, rather than prove MAIN-LEMMA-BASE-CASE by induction, perhaps we should prove some appropriate supporting lemmas by induction. For example, we proved that MINLIST + 1 < MAXLIST by proving two other lemmas by induction instead.

BUT: amazingly, the goal displayed above does suggest a lemma to be proved. From

```
(equal (maxlist (cdr x)) 0)
```

we should be able to simplify (SUMLIST (CDR X)) to 0. As usual, we tend to prefer unconditional rewrite rules, which explains our statement of MAXLIST-0-IS-SUMLIST-0 on page 61.

Point 11.

Using the checkpoint tool we find the following goal:

```
(implies (and (listp x)
              (equal (sumlist x)
                     (times k (length x)))
              (equal (minlist x) (maxlist x))
              (numberp k))
         (equal (maxlist x) k))
```

The following lemma should help (i.e., the goal above should then follow from a simple property of TIMES and equality reasoning):

```
(prove-lemma main-lemma-base-case-lemma-2-lemma (rewrite)
             (implies (equal (minlist x) (maxlist x))
                      (equal (sumlist x) (times (minlist x) (length x)))))
```

Now we try again MAIN-LEMMA-BASE-CASE-LEMMA-2. OH NO, binding stack overflows! So, let's pop the events chronology:

```
(ubt)
```

Maybe it will help avoid the stack overflow if we instruct the system not to store MAIN-LEMMA--BASE-CASE-LEMMA-2-LEMMA as a rewrite rule this time. (Presumably we could simply have disabled this rule; but we didn't.) So, that is what we do.

Point 12.

So, does the following work now?

```
(prove-lemma main-lemma-base-case-lemma-2 ()
             (implies (and (listp x)
                           (equal (sumlist x) (times k (length x)))
                           (equal (minlist x) (maxlist x)))
                      (equal (minlist x) (fix k)))
             ((use (main-lemma-base-case-lemma-2-lemma)
                   (main-lemma-base-case-lemma-2-hack
                    (sumlist (sumlist x))
                    (maxlist (maxlist x))
                    (n (length x))
                    (k k)))))
```

No! Here is the checkpoint:

```
(implies (and (equal (length x) 0)
              (listp x)
              (equal (sumlist x) 0)
              (equal (minlist x) (maxlist x))
              (numberp k))
         (equal (maxlist x) k))
```

The first two hypotheses are obviously contradictory. So, we only need one more lemma, and then `MAIN-LEMMA-BASE-CASE-LEMMA-2` will be proved.

Point 13.

We give the induction scheme with the following command:

```
(induct (induction-fn x k))
```

The proof-checker creates three subgoals. We look the current goal using the `TH` command. The system responds as shown below.

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

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

The current subterm is:
```

```
(IMPLIES (NOT (EQUAL (SUMLIST X)
                     (TIMES K (LENGTH X))))
         (IMPLIES (EQUAL (SUMLIST X)
                         (TIMES K (LENGTH X)))
                  (NOT (LESSP (EXP K (LENGTH X))
                              (PRODLIST X)))))
```

This case is trivial, so we just simplify it using the `S` command. The goal is proved and now the proof-checker considers the second goal. We use `TH` as above and the system prints out the following.

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

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

The current subterm is:
(IMPLIES (AND (EQUAL (SUMLIST X)
                     (TIMES K (LENGTH X)))
              (EQUAL (MINLIST X) (MAXLIST X)))
         (IMPLIES (EQUAL (SUMLIST X)
                         (TIMES K (LENGTH X)))
                  (NOT (LESSP (EXP K (LENGTH X))
                              (PRODLIST X)))))
```

Since we just proved the "base case," we should be able to prove this goal. In fact, the `PROVE` command succeeds. Now we have to consider the last goal. Let's look it using `TH`. The system responds:

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

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

The current subterm is:
(IMPLIES
 (AND
  (EQUAL (SUMLIST X)
         (TIMES K (LENGTH X)))
```

```
  (NOT (EQUAL (MINLIST X) (MAXLIST X)))
  (IMPLIES
     (EQUAL (SUMLIST (CONS (SUB1 (MAXLIST X))
                           (CONS (ADD1 (MINLIST X))
                                 (DELETE1 (MAXLIST X)
                                          (DELETE1 (MINLIST X) X)))))
            (TIMES K
                   (LENGTH (CONS (SUB1 (MAXLIST X))
                                 (CONS (ADD1 (MINLIST X))
                                       (DELETE1 (MAXLIST X)
                                                (DELETE1 (MINLIST X) X)))))))
     (NOT (LESSP (EXP K
                      (LENGTH (CONS (SUB1 (MAXLIST X))
                                    (CONS (ADD1 (MINLIST X))
                                          (DELETE1 (MAXLIST X)
                                                   (DELETE1 (MINLIST X) X))))))
                 (PRODLIST (CONS (SUB1 (MAXLIST X))
                                 (CONS (ADD1 (MINLIST X))
                                       (DELETE1 (MAXLIST X)
                                                (DELETE1 (MINLIST X) X)))))))))
 (IMPLIES (EQUAL (SUMLIST X)
                 (TIMES K (LENGTH X)))
          (NOT (LESSP (EXP K (LENGTH X))
                      (PRODLIST X)))))
```

What a mess above! It's better to split into "pretty" subcases using the `SPLIT` command. The system creates two new subgoals. Let's look the first using `TH`.

```
*** Active top-level hypotheses:
H1.  (NOT (EQUAL (MINLIST X) (MAXLIST X)))
H2.  (NOT (LESSP (EXP K
                      (LENGTH (CONS (SUB1 (MAXLIST X))
                                    (CONS (ADD1 (MINLIST X))
                                          (DELETE1 (MAXLIST X)
                                                   (DELETE1 (MINLIST X) X))))))
                 (PRODLIST (CONS (SUB1 (MAXLIST X))
                                 (CONS (ADD1 (MINLIST X))
                                       (DELETE1 (MAXLIST X)
                                                (DELETE1 (MINLIST X) X)))))))
H3.  (EQUAL (SUMLIST X)
            (TIMES K (LENGTH X)))
```

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

The current subterm is:
(NOT (LESSP (EXP K (LENGTH X))
            (PRODLIST X)))
```

Now we change to the other subgoal using the CG command. We type TH and the system responds:

```
*** Active top-level hypotheses:
H1.  (NOT (EQUAL (MINLIST X) (MAXLIST X)))
H2.  (NOT
      (EQUAL (SUMLIST (CONS (SUB1 (MAXLIST X))
                            (CONS (ADD1 (MINLIST X))
                                  (DELETE1 (MAXLIST X)
                                           (DELETE1 (MINLIST X) X)))))
             (TIMES K
                    (LENGTH (CONS (SUB1 (MAXLIST X))
                                  (CONS (ADD1 (MINLIST X))
                                        (DELETE1 (MAXLIST X)
                                                 (DELETE1 (MINLIST X) X))))))))
H3.  (EQUAL (SUMLIST X)
            (TIMES K (LENGTH X)))

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

The current subterm is:
(NOT (LESSP (EXP K (LENGTH X))
            (PRODLIST X)))
```

Let's call the theorem prover to simplify this goal using the BASH command. The system responds as follows.

```
***** Now entering the theorem prover's rewriter - simplifier *****

The goal has been simplified using COMMUTATIVITY2-OF-PLUS, DIFFERENCE-LEQ-ARG1,
EQUAL, LESSP, CORRECTNESS-OF-CANCEL-DIFFERENCE-PLUS, FIX,
MAXLIST-0-IS-SUMLIST-0, COMMUTATIVITY-OF-PLUS, MAXLIST-GEQ-MINLIST, CDR-CONS,
```

```
CAR-CONS, SUMLIST, PLUS-ADD1-ARG1, ZERO,
MINLIST-LESS-THAN-MAXLIST-MINUS-1-BETTER
MEMBER-MAXLIST, MEMBER-DELETE1, SUMLIST-DELETE1, ZEROP, PLUS-ZERO-ARG2,
PLUS-DIFFERENCE-ARG2, PLUS-ADD1-ARG2, LENGTH, TIMES-ADD1, NOT, AND, and
IMPLIES.

Creating 4 new subgoals, (((MAIN . 3) . 2) . 1), (((MAIN . 3) . 2) . 2),
(((MAIN . 3) . 2) . 3), and (((MAIN . 3) . 2) . 4).

The proof of the current goal, ((MAIN . 3) . 2), has been completed.  However,
the following subgoals of ((MAIN . 3) . 2) remain to be proved:
(((MAIN . 3) . 2) . 1), (((MAIN . 3) . 2) . 2), (((MAIN . 3) . 2) . 3), and
(((MAIN . 3) . 2) . 4).
Now proving (((MAIN . 3) . 2) . 1).
```

The current subgoal is printed in response to the TH command:

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

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

The current subterm is:
(IMPLIES
   (AND (NOT (EQUAL (MINLIST X) (MAXLIST X)))
        (NOT (EQUAL (SUMLIST X) 0))
        (NOT (MEMBER (MINLIST X) X))
        (LESSP (MAXLIST X) (SUMLIST X))
        (LESSP (MAXLIST X)
               (PLUS (MINLIST X) (SUMLIST X)))
        (NOT (EQUAL (ADD1 (DIFFERENCE (PLUS (MINLIST X)
                                            (SUMLIST X)
                                            (SUB1 (MAXLIST X)))
                                      (MAXLIST X)))
                    (PLUS K K
                          (TIMES K
                                 (LENGTH (DELETE1 (MAXLIST X)
                                                  (DELETE1 (MINLIST X) X)))))))
        (EQUAL (SUMLIST X)
               (TIMES K (LENGTH X))))
   (NOT (LESSP (EXP K (LENGTH X))
```

```
                (PRODLIST X))))
```

We notice the term of the form `(LENGTH (DELETE1 ...))` in the next-to-last hypothesis above. There are other terms we *could* usually notice, but we happen to notice this one first, and it seems that we can prove a lemma to simplify terms of this form. Recall that this is the usual low-level mode of interaction with Nqthm: prove lemmas that simplify (canonicalize, if you will) terms that appear during proofs. So, at this point we end the session by typing `EXIT`.

Point 14.

The first goal to be proved is now:

```
(implies (and (not (equal (minlist x) (maxlist x)))
              (not (equal (sumlist x) 0))
              (not (member (minlist x) x))
              (listp x)
              (lessp (maxlist x) (sumlist x))
              (lessp (maxlist x)
                     (plus (minlist x) (sumlist x)))
              (not (equal (add1 (difference (plus (minlist x)
                                                  (sumlist x)
                                                  (sub1 (maxlist x)))
                                            (maxlist x)))
                          (plus k (sumlist x))))
              (equal (sumlist x)
                     (times k (length x))))
         (not (lessp (exp k (length x))
                     (prodlist x))))
```

But notice that the third and fourth hypotheses are contradictory!

Point 15.

The checkpoint tool produces:

```
(implies (and (not (equal (minlist x) (maxlist x)))
              (not (equal k 0))
              (numberp k)
              (listp x)
              (not (numberp-listp (delete1 (maxlist x)
                                           (delete1 (minlist x) x))))
              (numberp-listp x))
         (not (lessp (exp k (length x))
                     (prodlist x))))
```

The fourth and fifth hypotheses are obviously contradictory. The lemma shown in the previous level is the relevant fact to prove.

Point 16.

The first checkpoint given to us by the tool doesn't seem helpful to us, but consider the second one:

```
(implies
 (and (equal (sumlist x)
             (times k (length x)))
      (not (equal (minlist x) (maxlist x)))
      (not (equal (sumlist x) 0))
      (listp x)
      (not (equal (length x) 1))
      (not (lessp (maxlist x)
                  (difference (sumlist x) (minlist x))))
      (not (equal (add1 (plus (minlist x) (sub1 (maxlist x))))
                  (plus k (times k (sub1 (length x))))))
      (numberp-listp x))
 (not (lessp (exp k (length x))
             (prodlist x))))
```

The sixth hypothesis says:

```
(maxlist x) >= (sumlist x) - (minlist x)
```

i.e.

```
(maxlist x) + (minlist x) >= (sumlist x).
```

But it is clear on general principles that (as from the assumptions `X` contains more than one element)

```
(maxlist x) + (minlist x) <= (sumlist x).
```

Therefore, we should be able to deduce that

```
(*)    (maxlist x) + (minlist x) = (sumlist x).
```

But consider the next-to-last hypothesis,

```
(not (equal (add1 (plus (minlist x) (sub1 (maxlist x))))
            (plus k (times k (sub1 (length x))))))
```

The first argument of EQUAL can be simplified:

```
(not (equal (plus (minlist x) (maxlist x))
            (plus k (times k (sub1 (length x))))))
```

But the second is the same as (TIMES K (LENGTH X)), so we obtain:

```
(not (equal (plus (minlist x) (maxlist x))
            (times k (length x))))
```

and this contradicts (*) above.

NOTE: From a high-level point of view we might be tempted to back up and simply say: What we need to prove here is that when we pull the minimum and maximum out of a list (using DELETE1) and then put them back on the front, but first incrementing the minimum and decrementing the maximum, the SUMLIST remains unchanged. But at this point, we seem to have a method of pushing ahead in the traditional Nqthm style (of proving simplification rules, especially rewrite rules) without moving to such a high-level strategy.

So, let us take the strategy that we want to replace

```
(add1 (plus (minlist x) (sub1 (maxlist x))))
```

with

```
(sumlist x)
```

in the present setting, and then hope that the simplifier can complete the proof from there.

Point 17.

After the binding stack overflow, we type :q to return to the lisp top level. We submit the form (BREAK-REWRITE) (see page 29) and then the command PATH to inspect the current rewrite path. The system prints out the following, which suggests to us that the rewriter may be in an infinite loop.

```
 (196435)   0. (top)
 (196434)   1. (initializing the linear database)
 (196428)   2. Applying  SUMLIST-GEQ-MINLIST-PLUS-MAXLIST
 (196268)   3. Rewriting (PLUS ...)
 (195620)   4. Applying  COMMUTATIVITY-OF-PLUS
 (195619)   5. Rewriting (PLUS ...)
 (195410)   6. Rewriting (IF ...)
 (195046)   7. Rewriting (ADD1 ...)
 (194024)   8. Applying  ADD1-PLUS-SUB1-SECOND
 (194021)   9. Rewriting (PLUS ...)
 (193882)  10. Applying  COMMUTATIVITY-OF-PLUS
 (193881)  11. Rewriting (PLUS ...)
 (193674)  12. Rewriting (ADD1 ...)
 (192652)  13. Applying  ADD1-PLUS-SUB1-SECOND
 (192649)  14. Rewriting (PLUS ...)
 (192510)  15. Applying  COMMUTATIVITY-OF-PLUS
 (192509)  16. Rewriting (PLUS ...)
 (192302)  17. Rewriting (ADD1 ...)
<<< and so on >>>
```

We give below further interaction with the system, annotating our input with comments. As usual, comments extend from each semicolon (;) to the end of the line. Some whitespace has been added. Careful inspection of this display shows where the loop is coming from:

```
: 3 ;; Show the 3th frame of the rewrite path.

---- Frame 3 -----           (persistence 196268)
Rewriting the right hand side of the conclusion of
SUMLIST-GEQ-MINLIST-PLUS-MAXLIST:
(PLUS (MINLIST X) (MAXLIST X))
under the substitution:
X <- X
: 4 ;; Show the 4th frame of the rewrite path.

---- Frame 4 -----           (persistence 195620)
Attempting to apply the replacement rule COMMUTATIVITY-OF-PLUS
using the substitution:
Y <- (MAXLIST X)
X <- (MINLIST X)
: frame! ;; Print the current frame again, in more detail.
```

```
---- Frame 4 -----           (persistence 195620)
Attempting to apply the replacement rule COMMUTATIVITY-OF-PLUS:
(EQUAL (PLUS X Y) (PLUS Y X))
to the target term:
(PLUS (MINLIST X) (MAXLIST X))
using the substitution:
Y <- (MAXLIST X)
X <- (MINLIST X)
: nx ;; Move one level away from the top level.

---- Frame 5 -----           (persistence 195619)
Rewriting the right hand side of the conclusion of COMMUTATIVITY-OF-PLUS:
(PLUS Y X)
under the substitution:
Y <- (MAXLIST X)
X <- (MINLIST X)
: nx ;; Move one more level away from the top level.

---- Frame 6 -----           (persistence 195410)
Rewriting the result of rewriting the body of PLUS:
(IF (EQUAL (SUMLIST X) 0)
    (MINLIST X)
    (ADD1 (PLUS # #)))
under the substitution:
NIL
: frame! ;; Show preceding frame in more detail.

---- Frame 6 -----           (persistence 195410)
Rewriting the result of rewriting the body of PLUS:
(IF (EQUAL (SUMLIST X) 0)
    (MINLIST X)
    (ADD1 (PLUS (SUB1 (MAXLIST X))
                (MINLIST X))))
under the substitution:
NIL
:
```

And without going further, we can see that the third argument of the `IF` expression above will rewrite to `(PLUS (MAXLIST X) (MINLIST X))`, which is the term we saw in frame 3, using the lemma `ADD1-PLUS-SUB1-SECOND` (after a rewrite rule stating commutativity of `PLUS` has been applied, not shown here, as that has already been popped off the rewrite stack). One solution: disable the recursive definition of `PLUS`.

Point 18.

We submit the form `(MAINTAIN-REWRITE-PATH T)` to maintain the rewrite path. Then we try the main lemma again. When we inspect the rewrite path (see page 29), this time we find:

```
...
 (  479) 862. Applying  MAXLIST-0-IS-SUMLIST-0
 (  478) 863. Rewriting (EQUAL ...)
 (  446) 864. Applying  MAXLIST-0-IS-SUMLIST-0
 (  445) 865. Rewriting (EQUAL ...)
 (  413) 866. Applying  MAXLIST-0-IS-SUMLIST-0
 (  412) 867. Rewriting (EQUAL ...)
 (  380) 868. Applying  MAXLIST-0-IS-SUMLIST-0
 (  379) 869. Rewriting (EQUAL ...)
 (  347) 870. Applying  MAXLIST-0-IS-SUMLIST-0
 (  346) 871. Rewriting (EQUAL ...)
...
```

Point 19.

Let us try using the Pc-Nqthm interactive loop on the lemma `PRODUCT-OF-MODIFIED-LIST`.

```
(verify (implies (and (lessp min max)
                      (member min x)
                      (member max x))
                 (let ((rest (prodlist (delete1 max (delete1 min x)))))
                   (equal (times (add1 min)
                                 (sub1 max)
                                 rest)
                          (plus (prodlist x)
                                (times (difference max (add1 min))
                                       rest))))))
```

We use `BASH` to simplify our goal. The proof-checker creates a subgoal and we look it using `TH`. The system responds as shown below:

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

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

The current subterm is:
```

```
(IMPLIES (AND (LESSP MIN MAX)
              (MEMBER MIN X)
              (MEMBER MAX X))
         (EQUAL (TIMES (ADD1 MIN)
                       (SUB1 MAX)
                       (PRODLIST (DELETE1 MAX (DELETE1 MIN X))))
                (PLUS (PRODLIST X)
                      (TIMES (SUB1 (DIFFERENCE MAX MIN))
                             (PRODLIST (DELETE1 MAX (DELETE1 MIN X)))))))
```

Let's treat the left hand side of this `IMPLIES` term as top-level hypotheses, by using the command `PROMOTE`. Then we submit the command `P` to look the current term and the system responds as shown below. (If we had cared to see the three new top-level hypotheses, we should have used the command `TH` instead, for example.)

```
(EQUAL (TIMES (ADD1 MIN)
              (SUB1 MAX)
              (PRODLIST (DELETE1 MAX (DELETE1 MIN X))))
       (PLUS (PRODLIST X)
             (TIMES (SUB1 (DIFFERENCE MAX MIN))
                    (PRODLIST (DELETE1 MAX (DELETE1 MIN X))))))
```

Now we move to the left hand side of this equality by typing `1`. Then we type `P` and the system responds:

```
(TIMES (ADD1 MIN)
       (SUB1 MAX)
       (PRODLIST (DELETE1 MAX (DELETE1 MIN X))))
```

Let's find out what rules apply to this term using the command `SR`, or `SHOW-REWRITES`. The system prints out the following information.

```
1. PRODUCT-OF-MODIFIED-LIST-LEMMA-2
     New term:
      (PLUS (TIMES MIN MAX
                   (PRODLIST (DELETE1 MAX (DELETE1 MIN X))))
            (TIMES (DIFFERENCE MAX (ADD1 MIN))
                   (PRODLIST (DELETE1 MAX (DELETE1 MIN X)))))

     Hypotheses: ((NUMBERP MAX) (NOT (EQUAL MAX 0)))
```

```
2. COMMUTATIVITY2-OF-TIMES
     New term:
      (TIMES (SUB1 MAX)
             (ADD1 MIN)
             (PRODLIST (DELETE1 MAX (DELETE1 MIN X))))

     Hypotheses:  <none>
<<< and so on >>>
```

Now we apply the first rule to the current term submitting the form `(R 1)`. The system creates two subgoals (corresponding to the hypotheses of the rule just applied). We use the `TOP` command to go to the top of the current goal. The command `TH` yields the following output:

```
*** Active top-level hypotheses:
H1.  (LESSP MIN MAX)
H2.  (MEMBER MIN X)
H3.  (MEMBER MAX X)

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

The current subterm is:
(EQUAL (PLUS (TIMES MIN MAX
                    (PRODLIST (DELETE1 MAX (DELETE1 MIN X))))
             (TIMES (DIFFERENCE MAX (ADD1 MIN))
                    (PRODLIST (DELETE1 MAX (DELETE1 MIN X)))))
       (PLUS (PRODLIST X)
             (TIMES (SUB1 (DIFFERENCE MAX MIN))
                    (PRODLIST (DELETE1 MAX (DELETE1 MIN X))))))
```

We try to prove this goal using `PROVE`. As we see that the proof goes into induction, we interrupt it. Then we simplify the current goal using `BASH`. The system creates a subgoal. As done above, we promote the hypotheses using `PROMOTE` and then we look the result typing `TH`.

```
*** Active top-level hypotheses:
H1.  (LESSP MIN MAX)
H2.  (MEMBER MIN X)
H3.  (MEMBER MAX X)

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

```
The current subterm is:
(EQUAL (TIMES MAX MIN
              (PRODLIST (DELETE1 MAX (DELETE1 MIN X))))
       (PRODLIST X))
```

We type `1` in order to consider the left hand side of this equality. Now, let's use `SR` to obtain the rewrite rules that can be applied to this term. This is the output from the system.

```
1. PRODUCT-OF-MODIFIED-LIST-LEMMA
     New term: (PRODLIST X)
     Hypotheses: ((NOT (EQUAL MIN MAX)))
2. COMMUTATIVITY2-OF-TIMES
     New term: (TIMES MIN MAX (PRODLIST (DELETE1 MAX (DELETE1 MIN X))))
     Hypotheses:  <none>
<<< and so on >>>
```

We apply the first rule to the current term submitting the form `(R 1)`. The system applies the rule and creates a new subgoal (corresponding to the hypothesis of the rule). At this point the current goal is trivial (as we can see going to the top with `TOP` and then typing `TH`), so we simplify it using the `S` command. The goal is proved and now the proof checker goes to the next goal on the stack. This case is trivial and we prove it using the `PROVE` command. We are able to do the same also with the next goal. So, the last goal to prove is:

```
*** Active top-level hypotheses:
H1.  (LESSP MIN MAX)
H2.  (MEMBER MIN X)
H3.  (MEMBER MAX X)

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

The current subterm is:
(NOT (EQUAL MAX 0))
```

We type `PROVE` and the proof-checker prints out the following:

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

```
This formula simplifies, using linear arithmetic, to:

      T.

Q.E.D.

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

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

Finally we have proved our lemma! Let's end the session submitting to the system the following form:

```
(exit product-of-modified-list (rewrite))
```

This is the output from the system:

```
The indicated goal has been proved.  Here is the desired event:
(PROVE-LEMMA PRODUCT-OF-MODIFIED-LIST
      (REWRITE)
      (IMPLIES (AND (LESSP MIN MAX)
                    (MEMBER MIN X)
                    (MEMBER MAX X))
               (EQUAL (TIMES (ADD1 MIN)
                             (SUB1 MAX)
                             (PRODLIST (DELETE1 MAX (DELETE1 MIN X))))
                      (PLUS (PRODLIST X)
                            (TIMES (DIFFERENCE MAX (ADD1 MIN))
                                   (PRODLIST (DELETE1 MAX (DELETE1 MIN X))))))))
      ((INSTRUCTIONS BASH PROMOTE
                     (DIVE 1)
                     (REWRITE PRODUCT-OF-MODIFIED-LIST-LEMMA-2)
                     TOP BASH PROMOTE
                     (DIVE 1)
                     (REWRITE PRODUCT-OF-MODIFIED-LIST-LEMMA)
                     TOP S PROVE PROVE PROVE)))
Do you want to submit this event?
Y (Yes), R (Yes and replay commands), or N (No) ?
```

We answer `N` to the question above, as we don't want to submit this event to the prover.

C The list of events

The following is the list of commands to be issued to the NQTHM prover in order to obtain our mechanized proof of the Arithmetic-Geometric Mean Theorem.

```
(note-lib "naturals" t)

(defn prodlist (lst)
  (if (listp lst)
      (times (car lst)
             (prodlist (cdr lst)))
    1))

(defn sumlist (lst)
  (if (listp lst)
      (plus (car lst)
            (sumlist (cdr lst)))
    0))

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

(defn maxlist (x)
  (if (listp x)
      (if (listp (cdr x))
          (max (car x) (maxlist (cdr x)))
        (fix (car x)))
    0))

(defn min (x y)
  (if (lessp x y) (fix x) (fix y)))

(defn minlist (x)
  (if (listp x)
      (if (listp (cdr x))
          (min (car x)
               (minlist (cdr x)))
        (fix (car x)))
    0))

(defn delete1 (elt x)
  (if (listp x)
```

```
      (if (equal elt (car x))
          (cdr x)
        (cons (car x) (delete1 elt (cdr x))))
    x))

(prove-lemma maxlist-delete1-rearrange (rewrite)
             (implies (member b x)
                      (equal (maxlist (cons b (delete1 b x)))
                             (maxlist x))))

(deftheory induction-fn-disables
  (sumlist times length minlist maxlist delete1 occurrences))

(prove-lemma max-greater-than-average (rewrite)
             (implies (and (listp x)
                           (geq (sumlist x)
                                (times k (length x)))
                           (not (equal (fix k) (maxlist x))))
                      (lessp k (maxlist x))))

(prove-lemma min-less-than-average (rewrite)
             (implies (and (listp x)
                           (leq (sumlist x)
                                (times k (length x)))
                           (not (equal (fix k) (minlist x))))
                      (lessp (minlist x) k)))

(prove-lemma times-monotone-1 (rewrite)
             (implies (not (lessp u v))
                      (not (lessp (times u y)
                                  (times v y)))))

(prove-lemma lessp-times-preserved-in-first-arg (rewrite)
             (implies (and (not (lessp a (times u y)))
                           (not (lessp u v)))
                      (not (lessp a (times v y)))))

(prove-lemma minlist-main-property (rewrite)
             (not (lessp (sumlist x)
                         (times (length x) (minlist x)))))

(prove-lemma minlist-not-maxlist-implies-minlist-lessp-average-lemma
             ()
```

```
             (implies (not (equal (minlist x) (maxlist x)))
                      (lessp (times (minlist x) (length x))
                             (sumlist x))))

(prove-lemma minlist-not-maxlist-implies-minlist-lessp-average (rewrite)
             (implies (and (listp x)
                           (equal (sumlist x)
                                  (times k (length x)))
                           (not (equal (minlist x) (maxlist x))))
                      (lessp (minlist x) (fix k)))
             ((use (minlist-not-maxlist-implies-minlist-lessp-average-lemma))))

(prove-lemma maxlist-main-property (rewrite)
             (not (lessp (times (length x) (maxlist x))
                         (sumlist x))))

(prove-lemma minlist-not-maxlist-implies-maxlist-greaterp-average-lemma ()
             (implies (not (equal (minlist x) (maxlist x)))
                      (lessp (sumlist x)
                             (times (maxlist x) (length x)))))

(prove-lemma
 minlist-not-maxlist-implies-maxlist-greaterp-average
 (rewrite)
 (implies (and (listp x)
               (equal (sumlist x)
                      (times k (length x)))
               (not (equal (minlist x) (maxlist x))))
          (lessp (fix k) (maxlist x)))
 ((use (minlist-not-maxlist-implies-maxlist-greaterp-average-lemma))))

(prove-lemma minlist-less-than-maxlist-minus-1 (rewrite)
             (implies (and (listp x)
                           (equal (sumlist x)
                                  (times k (length x)))
                           (not (equal (minlist x) (maxlist x))))
                      (lessp (minlist x) (sub1 (maxlist x))))
             ((use
               (minlist-not-maxlist-implies-minlist-lessp-average)
               (minlist-not-maxlist-implies-maxlist-greaterp-average))))

(prove-lemma maxlist-delete1-leq (rewrite)
             (not (lessp (maxlist lst)
```

```
                      (maxlist (delete1 a lst)))))

(prove-lemma member-implies-maxlist-geq (rewrite)
             (implies (member a x)
                      (not (lessp (maxlist x) a))))

(prove-lemma
 delete1-preserves-maxlist-when-maxlist-occurs-more-than-once
 (rewrite)
 (implies (lessp 1 (occurrences (maxlist x) x))
          (equal (maxlist (delete1 any-element x))
                 (maxlist x))))

(prove-lemma delete1-occurrences (rewrite)
             (equal (occurrences a (delete1 b x))
                    (if (and (equal a b)
                             (member b x))
                        (sub1 (occurrences a x))
                      (occurrences a x))))

(prove-lemma occurrence-implies-listp (rewrite)
             (implies (lessp 1 (occurrences a x))
                      (equal (listp (delete1 a x)) t)))

(prove-lemma maxlist-geq-minlist (rewrite)
             (not (lessp (maxlist x) (minlist x))))

(prove-lemma induction-fn-help-2-max-occurs-twice-lemma-1 (rewrite)
             (implies
              (and (equal (sumlist x)
                          (times k (length x)))
                   (not (equal (minlist x) (maxlist x)))
                   (lessp 1 (occurrences (maxlist x) x)))
              (equal (maxlist (cons (sub1 (maxlist x))
                                    (cons (add1 (minlist x))
                                          (delete1 (maxlist x)
                                                   (delete1
                                                    (minlist x)
                                                    x)))))
                     (maxlist x))))

(prove-lemma member-delete1 (rewrite)
             (equal (member a (delete1 b c))
```

```
                    (if (equal a b)
                        (lessp 1 (occurrences b c))
                      (member a c))))

(prove-lemma member-implies-listp (rewrite)
             (implies (member a x)
                      (listp x)))

(prove-lemma
 induction-fn-help-2-max-occurs-twice-lemma-2
 (rewrite)
 (implies
  (and (equal (sumlist x)
              (times k (length x)))
       (not (equal (minlist x) (maxlist x)))
       (lessp 1 (occurrences (maxlist x) x)))
  (equal (lessp (occurrences (maxlist x)
                             (cons (sub1 (maxlist x))
                                   (cons (add1 (minlist x))
                                         (delete1 (maxlist x)
                                                  (delete1 (minlist x) x)))))
                (occurrences (maxlist x) x))
         t)))

(disable member-implies-listp)

(prove-lemma induction-fn-help-2-max-occurs-twice ()
  (let ((x0 (cons (sub1 (maxlist x))
                  (cons (add1 (minlist x))
                        (delete1 (maxlist x)
                                 (delete1 (minlist x) x))))))
    (implies
     (and (equal (sumlist x) (times k (length x)))
          (not (equal (minlist x) (maxlist x)))
          (lessp 1 (occurrences (maxlist x) x)))
     (ord-lessp
      (cons (add1 (maxlist x0)) (occurrences (maxlist x0) x0))
      (cons (add1 (maxlist x)) (occurrences (maxlist x) x)))))
 ((disable-theory induction-fn-disables)))

(prove-lemma maxlist-not-minlist-implies-listp ()
             (implies (not (equal (minlist x) (maxlist x)))
                      (listp x)))
```

```
(prove-lemma minlist-less-than-maxlist-minus-1-better (rewrite)
             (implies (and (equal (sumlist x)
                                  (times k (length x)))
                           (not (equal (minlist x) (maxlist x))))
                      (lessp (minlist x) (sub1 (maxlist x))))
             ((use (maxlist-not-minlist-implies-listp))))

(disable minlist-less-than-maxlist-minus-1)

(prove-lemma maxlist-delete1-delete1 (rewrite)
             (not (lessp (maxlist (delete1 b x))
                         (maxlist (delete1 b (delete1 a x))))))

(prove-lemma member-maxlist (rewrite)
             (implies (not (equal (maxlist z) 0))
                      (member (maxlist z) z)))

(prove-lemma lessp-maxlist-delete1-maxlist (rewrite)
             (implies (and (lessp 0 (maxlist x))
                           (not (lessp 1 (occurrences (maxlist x) x))))
                      (lessp (maxlist (delete1 (maxlist x) x))
                             (maxlist x))))

(prove-lemma
 induction-fn-help-2-max-occurs-once-main-lemma
 (rewrite)
 (implies
  (and
   (equal (sumlist x)
          (times k (length x)))
   (not (equal (minlist x) (maxlist x)))
   (not (lessp 1 (occurrences (maxlist x) x))))
  (equal (lessp (maxlist (cons (sub1 (maxlist x))
                               (cons (add1 (minlist x))
                                     (delete1 (maxlist x)
                                              (delete1 (minlist x) x)))))
                (maxlist x))
         t)))

(prove-lemma induction-fn-help-2-max-occurs-once ()
   (let ((x0 (cons (sub1 (maxlist x))
                   (cons (add1 (minlist x))
```

```
                       (delete1 (maxlist x)
                                (delete1 (minlist x) x))))))
    (implies
     (and (equal (sumlist x) (times k (length x)))
          (not (equal (minlist x) (maxlist x)))
          (not (lessp 1 (occurrences (maxlist x) x))))
     (ord-lessp
      (cons (add1 (maxlist x0)) (occurrences (maxlist x0) x0))
      (cons (add1 (maxlist x)) (occurrences (maxlist x) x)))))
 ((disable occurrences minlist maxlist delete1)))

(prove-lemma induction-fn-help-2 (rewrite)
  (let ((x0 (cons (sub1 (maxlist x))
                  (cons (add1 (minlist x))
                        (delete1 (maxlist x)
                                 (delete1 (minlist x) x))))))
    (implies
     (and (equal (sumlist x) (times k (length x)))
          (not (equal (minlist x) (maxlist x))))
     (ord-lessp
      (cons (add1 (maxlist x0)) (occurrences (maxlist x0) x0))
      (cons (add1 (maxlist x)) (occurrences (maxlist x) x)))))
 ((use (induction-fn-help-2-max-occurs-twice)
       (induction-fn-help-2-max-occurs-once))
  (disable-theory t)
  (enable-theory ground-zero)))

(disable-theory induction-fn-disables)

(defn induction-fn (x k)
  (if (not (equal (sumlist x) (times k (length x))))
      t
    (if (equal (minlist x) (maxlist x))
        t
      (induction-fn (cons (sub1 (maxlist x))
                          (cons (add1 (minlist x))
                                (delete1 (maxlist x)
                                         (delete1 (minlist x) x))))
                    k)))
  ((ord-lessp
    (cons (add1 (maxlist x))
          (occurrences (maxlist x) x)))))
```

```
(enable-theory induction-fn-disables)

(prove-lemma sumlist-delete1-plus-version ()
             (equal (plus a (sumlist (delete1 a x)))
                    (if (member a x)
                        (sumlist x)
                      (plus a (sumlist x)))))

(prove-lemma sumlist-delete1 (rewrite)
             (equal (sumlist (delete1 a x))
                    (if (member a x)
                        (difference (sumlist x) a)
                      (sumlist x)))
             ((use (sumlist-delete1-plus-version))))

(prove-lemma maxlist-0-is-sumlist-0 (rewrite)
             (equal (equal (maxlist x) 0)
                    (equal (sumlist x) 0)))

(prove-lemma main-lemma-base-case-lemma-1 ()
             (implies (equal (minlist x) (maxlist x))
                      (equal (exp (minlist x) (length x))
                             (prodlist x))))

(prove-lemma main-lemma-base-case-lemma-2-lemma ()
             (implies (equal (minlist x) (maxlist x))
                      (equal (sumlist x)
                             (times (minlist x) (length x)))))

(prove-lemma main-lemma-base-case-lemma-2-hack ()
             (implies (and (equal (times k n)
                                  sumlist)
                           (not (zerop n))
                           (equal (times n maxlist)
                                  sumlist))
                      (equal (fix k) (fix maxlist))))

(prove-lemma equal-length-0 (rewrite)
             (equal (equal (length x)
                           0)
                    (not (listp x))))

(prove-lemma main-lemma-base-case-lemma-2 ()
```

```
             (implies (and (listp x)
                           (equal (sumlist x)
                                  (times k (length x)))
                           (equal (minlist x) (maxlist x)))
                      (equal (minlist x) (fix k)))
             ((use (main-lemma-base-case-lemma-2-lemma)
                   (main-lemma-base-case-lemma-2-hack
                    (sumlist (sumlist x))
                    (maxlist (maxlist x))
                    (n (length x))
                    (k k)))))

(prove-lemma main-lemma-base-case (rewrite)
             (implies (and (equal (sumlist x)
                                  (times k (length x)))
                           (equal (minlist x) (maxlist x)))
                      (equal (exp k (length x))
                             (prodlist x)))
             ((use (main-lemma-base-case-lemma-1)
                   (main-lemma-base-case-lemma-2))))

(prove-lemma length-delete1 (rewrite)
             (equal (length (delete1 a x))
                    (if (member a x)
                        (sub1 (length x))
                      (length x))))

(defn numberp-listp (x)
  (if (listp x)
      (and (numberp (car x))
           (numberp-listp (cdr x)))
    (equal x nil)))

(prove-lemma member-minlist (rewrite)
             (implies (and (listp x)
                           (numberp-listp x))
                      (member (minlist x) x)))

(prove-lemma numberp-listp-delete1 (rewrite)
             (implies (numberp-listp x)
                      (numberp-listp (delete1 a x))))

(prove-lemma add1-plus-sub1-second (rewrite)
```

```
            (implies (not (zerop y))
                     (equal (add1 (plus x (sub1 y)))
                            (plus x y))))

(prove-lemma sumlist-geq-maxlist (rewrite)
             (not (lessp (sumlist x)
                         (maxlist x))))

(prove-lemma sumlist-geq-minlist (rewrite)
             (not (lessp (sumlist x)
                         (minlist x))))

(prove-lemma sumlist-geq-minlist-plus-maxlist (rewrite)
             (implies (not (equal (minlist x) (maxlist x)))
                      (not (lessp (sumlist x)
                                  (plus (minlist x) (maxlist x))))))

(disable plus)

(prove-lemma plus-times-sub1-second (rewrite)
             (implies (not (zerop y))
                      (equal (plus x (times x (sub1 y)))
                             (times x y))))

(disable times)

(prove-lemma minlist-not-maxlist-implies-length-at-least-2 (rewrite)
             (implies (not (equal (minlist x) (maxlist x)))
                      (lessp 1 (length x))))

(disable plus-add1-arg1)
(disable plus-add1-arg2)

(disable times-add1)

(disable maxlist-0-is-sumlist-0)

(prove-lemma times-prodlist-delete1 (rewrite)
             (implies (member a x)
                      (equal (times a (prodlist (delete1 a x)))
                             (prodlist x))))

(prove-lemma product-of-modified-list-lemma (rewrite)
```

```
            (implies (and (not (equal min max))
                          (member min x)
                          (member max x))
                     (equal (times max
                                   min
                                   (prodlist (delete1 max
                                                      (delete1
                                                       min
                                                       x))))
                            (prodlist x))))

(disable times-prodlist-delete1)

(prove-lemma product-of-modified-list-lemma-2 (rewrite)
            (implies (and (lessp min max)
                          (not (zerop max)))
                     (equal (times (add1 min)
                                   (sub1 max)
                                   rest)
                            (plus (times min max rest)
                                  (times (difference max (add1 min))
                                         rest))))
            ((enable times-add1 times
                     plus-add1-arg1 plus-add1-arg2 plus)
             (disable plus-times-sub1-second
                      add1-plus-sub1-second)))

(prove-lemma positive-implies-numberp ()
            (implies (lessp min max)
                     (numberp max)))

(prove-lemma product-of-modified-list (rewrite)
            (implies (and (lessp min max)
                          (member min x)
                          (member max x))
                     (let ((rest (prodlist (delete1 max
                                                    (delete1
                                                     min
                                                     x)))))
                          (equal (times (add1 min)
                                        (sub1 max)
                                        rest)
                                 (plus (prodlist x)
```

```
                                        (times (difference max (add1 min))
                                               rest)))))
             ((use (positive-implies-numberp))))

(prove-lemma main-lemma ()
             (let ((n (length x)))
               (implies (and (numberp-listp x)
                             (equal (sumlist x)
                                    (times k n)))
                        (not (lessp (exp k n)
                                    (prodlist x))))))
             ((induct (induction-fn x k))
              (do-not-induct t)))

(defn scalar-product (scalar lst)
  (if (listp lst)
      (cons (times scalar (car lst))
            (scalar-product scalar (cdr lst)))
    nil))

(prove-lemma sumlist-scalar-product (rewrite)
             (equal (sumlist (scalar-product scalar lst))
                    (times scalar (sumlist lst))))

(prove-lemma prodlist-scalar-product (rewrite)
             (equal (prodlist (scalar-product scalar lst))
                    (times (exp scalar (length lst))
                           (prodlist lst))))

(prove-lemma length-scalar-product (rewrite)
             (equal (length (scalar-product n a))
                    (length a)))

(prove-lemma numberp-listp-scalar-product (rewrite)
             (implies (numberp-listp a)
                      (numberp-listp (scalar-product n a))))

(prove-lemma main ()
             (implies (numberp-listp a)
                      (let ((n (length a)))
                        (geq (exp (sumlist a) n)
                             (times (exp n n)
                                    (prodlist a))))))
```

```
            ((use (main-lemma (k (sumlist a))
                              (x (scalar-product (length a) a))))))

(prove-lemma sumlist-for (rewrite)
             (equal (sumlist a)
                    (for x in a sum x)))

(prove-lemma prodlist-for (rewrite)
             (equal (prodlist a)
                    (for x in a multiply x)))

(prove-lemma main-again ()
             (implies (and (numberp-listp a)
                           (equal n (length a)))
                      (geq (exp (for x in a sum x) n)
                           (times (exp n n)
                                  (for x in a multiply x))))
             ((use (main))))
```

D Glossary

The following two tables contain the Nqthm commands, and the Nqthm built-in functions and constants that we have used in the mechanized proof. Each line of the first table contains the syntax of a command and the pages where the reader can find information about that command. Each line of the second table contains an informal description of a built-in object and the page where the object has been first introduced. For further details see [5] and [10]; however note that the commands `DEFTHEORY`, `DISABLE-THEORY`, `ENABLE-THEORY` and `DO-FILE` have been added to Nqthm since the time of publication of [5], and are documented (along with all Nqthm commands) in the "doc/" subdirectory of the Nqthm-1992 release.

Nqthm command	Syntax	Described on pages
ADD-AXIOM	(ADD-AXIOM name types formula)	22
BREAK-LEMMA	(BREAK-LEMMA name)	73
BREAK-REWRITE	(BREAK-REWRITE)	29
CH	(CH number)	23
DEFN	(DEFN name arguments body)	10
DEFTHEORY	(DEFTHEORY name list-of-names)	33
DISABLE	(DISABLE name)	55
DISABLE-THEORY	(DISABLE-THEORY name)	33
DO-FILE	(DO-FILE filename)	28
ENABLE	(ENABLE name)	72
ENABLE-THEORY	(ENABLE-THEORY name)	33
NOTE-LIB	(NOTE-LIB name flag)	20
MAINTAIN-REWRITE-PATH	(MAINTAIN-REWRITE-PATH flag)	28
PPE	(PPE 'name)	24
PROVE-LEMMA	(PROVE-LEMMA name types formula)	10, 13, 21, 23,45
R-LOOP	(R-LOOP)	31
UBT	(UBT), (UBT name)	12, 23
UNBREAK-LEMMA	(UNBREAK-LEMMA name)	73
VERIFY	(VERIFY formula), (VERIFY)	14, 41
VERIFY-DEFN	(VERIFY-DEFN formula)	30

Figure 2: Nqthm commands

Nqthm symbol	Informal description	Introduced on page
ADD1	successor function	7
AND	Boolean connective for conjunction	6
CAR	returns the head of a list	7
CDR	returns the tail of a list	7
CONS	list constructor	7
EQUAL	equality function	6
EXP	exponentiation function (from library naturals)	8
F	Boolean constant for falsity	6
FIX	returns the value of the argument (0 if the argument is not number)	24
GEQ	"greater than or equal to" function	8
IF	if-then-else function	6
IMPLIES	Boolean connective for implication	6
LEQ	"less than or equal to" function	8
LESSP	"less than" function	8
LET	construct for introducing local variables	7
LISTP	recognizer for non-empty lists	7
MAX	maximum function	24
MIN	minimum function	25
NIL	constant for empty list	7
NOT	Boolean connective for negation	6
OCCURRENCES	returns the number of occurrences of an element in a list (from library naturals)	25
OR	Boolean connective for disjunction	6
ORD-LESSP	order on notations for ordinals less than ϵ_0	27
PLUS	addition function	7
SUB1	predecessor function	7
T	Boolean constant for truth	6
TIMES	multiplication function	7
ZERO	base object for natural numbers	7

Figure 3: Nqthm built-in functions and constants

References

[1] David Basin and Matt Kaufmann. "The Boyer-Moore Prover and Nuprl: An Experimental Comparison" Proceedings of Workshop for Basic Research Action, Logical Frameworks. Antibes, France, May, 1990.

[2] William R. Bevier, Warren A. Hunt, Jr., J Strother Moore, and William D. Young. "An Approach to Systems Verification." Journal of Automated Reasoning, November, 1989.

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

[4] Robert S. Boyer and J Strother Moore. "Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures." In *The Correctness Problem in Computer Science*, ed. Robert S. Boyer and J Strother Moore, Academic Press, London, 1981.

[5] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook.* Academic Press, Boston, 1988.

[6] Robert S. Boyer, Matt Kaufmann, and J Strother Moore. "The Boyer-Moore Theorem Prover and Its Interactive Enhancement." (Submitted.)

[7] Alan Bundy. Talk in "Challenge Problems" section of: "Workshop on the automation of proof by mathematical induction," co-sponsored by MInd and IndUS, July 11-12, 1993; at AAAI-93 Eleventh National Conference on Artificial Intelligence, Washington DC, USA.

[8] Donald I. Good and William D. Young. "Mathematical Methods for Digital Systems Development." In *VDM'91 Formal Software Development Methods*, ed. S. Prehn, W.J. Toetenel, Springer-Verlag Lecture Notes in Computer Science 552, pp. 406–430, 1991. Also appears as Technical Report 67, Computational Logic, Inc., August, 1991.

[9] Scott D. Johnson and John Nagle. *Pascal-F Verifier User's Manual, Version 2.* Ford Aerospace & Communications Corporation, Palo Alto, California, 1986.

[10] Matt Kaufmann. "A User's Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover." Technical Report 19, Computational Logic, Inc., May, 1988.

[11] Matt Kaufmann. "Addition of Free Variables to the PC-NQTHM Interactive Enhancement of the Boyer-Moore Theorem Prover." Technical Report 42, Computational Logic, Inc., March, 1990.

[12] Matt Kaufmann. "Response to FM91 Survey of Formal Methods: Nqthm and Pc-Nqthm." Technical Report 75, Computational Logic, Inc., March, 1992.

[13] Matt Kaufmann. "An Assistant for Reading Nqthm Proof Output." Technical Report 85, Computational Logic, Inc., November, 1992.

[14] Matt Kaufmann. "An Example in NQTHM: Ramsey's Theorem." Internal Note 100, Computational Logic, Inc., November, 1988.

[15] Matt Kaufmann. "An Instructive Example for Beginning Users of the Boyer-Moore Theorem Prover." Internal Note 185, Computational Logic, Inc., April, 1990.

[16] Matt Kaufmann. "Generalization in the Presence of Free Variables: A Mechanically-Checked Correctness Proof for One Algorithm." *Journal of Automated Reasoning*, 7(1991), pp. 109-158.

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

[18] Richard M. Stallman. *GNU EMACS Manual*, Sixth Edition. Free Software Foundation, March, 1987.