Generalization in the Presence of Free Variables: a Mechanically-Checked Correctness Proof for One Algorithm

Matt Kaufmann

Technical Report #53 April, 1990

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

This research was supported in part by ONR Contract N00014-88-C-0454. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Office of Naval Research or the U.S. Government.

1. Introduction

The motivation for this work began with a concern for the correctness of an implementation of logic. The system PC-NQTHM[1] is an interactive ‘‘proof-checker’’ enhancement of the Boyer-Moore Theorem Prover [3], and is documented in [10]. In [11] we report on an extension of this system that admits a notion of *free variables*. Roughly, free variables are ones that the user is allowed to instantiate in the course of a proof. An earlier version of this extension for free variables had a soundness bug in one of the commands, called GENERALIZE. (This command allows one to replace terms by new variables and proceed by proving the stronger, generalized version of the goal. Thus, it corresponds to the inference rule of universal instantiation). In fact the bug was easily corrected and the correctness of the resulting GENERALIZE command was checked on paper. However, the rude shock of having made a soundness mistake in the previous version led to the following goal: formalize the new version of the GENERALIZE command in the Boyer-Moore logic, and mechanically check a proof of correctness of this formalization.

In this paper we present a mechanically-checked proof of correctness for a generalization algorithm. Although the theorem itself is probably new (at least, we are unaware of any existing statement of it), the interest here lies not particularly in the theorem *per se* but, rather, lies in the demonstration of the use of mechanical verification for assisting in the reliability of detailed proofs and software. In particular, we believe that this exercise strongly suggests the feasibility of creating a verified version of PC-NQTHM, i.e. one which is proved correct in the Boyer-Moore theorem prover or in some successor of that system.

Thus, this paper could be viewed as a contribution to the study of *metatheoretically extensible* systems. Some reports of research in this spirit can be found in works of Davis and Schwartz [6], Weyhrauch [18], Boyer and Moore [2], Shankar [16], Knoblock and Constable [14, 13], Howe [9], and Quaife [15]. However, we also view this paper as an exposition which provides a rather detailed look at the practice of using the Boyer-Moore theorem prover and PC-NQTHM to proof-check mathematical arguments.

Although the development here is intended to capture the behavior of PC-NQTHM, it is actually an abstraction of that behavior. Hence, no familiarity with PC-NQTHM is required for an understanding of this document. Moreover, little particular understanding of the Boyer-Moore logic (cf. [1, 3]) should be necessary for a comfortable reading of this paper (although for those interested, a complete treatment of the Boyer-Moore

[1]‘‘PC’’ for ‘‘proof-checker’’, ‘‘NQTHM’’ for the name commonly given to the current Boyer-Moore theorem prover

theorem prover and the enhancements used here can be found in [1, 3, 10, 12, 4, 11]). A summary of the basics needed in order to follow the treatment in this paper may be found in the first subsection 1.1 immediately below. We follow this with a very general discussion of the methodology employed in the use of the Boyer-Moore theorem prover and PC-NQTHM in Subsection 1.2. A brief view of the main theorem and the high-level structure of its proof may be found in Subsection 1.3. We conclude this introduction with a summary of the remainder of the paper.

1.1 Introduction to the Boyer-Moore logic and theorem prover

For a description of the Boyer-Moore logic and theorem prover we refer the reader to the careful description in [3]. For now let us simply point out a few aspects of the logic and theorem prover.

One may simply view the Boyer-Moore logic as a version of first-order logic that has an induction rule of inference. Further details will be provided as needed during the presentation below. For now, let us simply note that a session with the Boyer-Moore theorem prover consists of a sequence of so-called *events*, which are generally either definitions or lemmas/theorems. A sequence of events stored at a given moment is called a *history*. Thus, this paper can be viewed as the presentation of a particular history that culminates in a lemma event stating the correctness of the algorithm in question.

There are a few built-in function symbols which, together with corresponding axioms, are part of the logic's basic (built-in) theory, i.e. are part of every history. Here is a summary of some of those that we will use in this paper. In each case we write terms in two ways. First, we write them in official s-expression (Lisp) notation, i.e. in the form $\mathtt{(G\ t_1\ ...\ t_n)}$ where each $\mathtt{t_i}$ is a term in that notation and **G** is a function symbol (of the current history). Second, we write them in informal, more traditional notation. We will follow this convention throughout this paper. Moreover, we will write s-expressions using upper-case characters and traditional notation using lower-case characters. Here, then, are the primitives promised above.

- **(CONS X Y)** or **<x,y>**: the ordered pair formed from **x** and **y**. **CONS** is also used to represent lists (sequences), in which case the atom **NIL** represents the empty list and **(CONS X Y)** represents the sequence whose first element is **X** and whose remaining elements (in order) form the sequence **Y**.
- **(CAR Z)** or **1**st**(z)**: the first component of the ordered pair **z**
- **(CDR Z)** or **2**nd**(z)**: the second component of the ordered pair **z**
- **(LISTP Z)** or **listp(z)**: **z** is an ordered pair
- **T**: the boolean *true*
- **F**: the boolean *false*
- **(LESSP X Y)** or **x < y**: **x** is less than **y**

- **(MEMBER A X)** or **a** $\in$ **x**: **a** is a member of **x**

The basic logic does not contain first-order quantification, so one often expresses quantified concepts using primitive recursion. Consider the following (irrelevant but instructive) definition of a predicate that holds of a list if and only if all of its elements are ordered pairs.

Definition *of* **ALL-LISTP**

```
all-listp(x) ≡
(∀ y ∈ x) listp(y)

(DEFN ALL-LISTP (X)
  (IF (LISTP X)
      (AND (LISTP (CAR X))
           (ALL-LISTP (CDR X)))
    T))
```

The first version of this definition is informal. In fact **x** is (presumably) a list, not a set (there is no built-in set type), so the predicate $\in$ doesn't really make precise sense here, though it's highly suggestive. We'll continue in this style throughout this paper.

The theorem prover contains a number of ''processes'', but most of the work is done by its *simplifier*, whose main component is a *rewriter*. The user labels certain lemmas as *rewrite rules*, and the system then rewrites using them. For example, consider the following rule, which says how the function **ALL-LISTP** above applies to a **CONS**.

Lemma **ALL-LISTP-CONS**

```
listp(a) →
( all-listp({a} ∪ x) ≡ all-listp(x) )

(IMPLIES (LISTP A)
         (EQUAL (ALL-LISTP (CONS A X))
                (ALL-LISTP X)))
```

Again, the first version is merely suggestive, since the $\cup$ operator applies to sets, not lists. The *name* of this lemma is indicated to be ALL-LISTP-CONS. If we label it to be a *rewrite rule* then the theorem prover's rewriter will simplify any term of the form **(ALL-LISTP (CONS A X))** to the term **(ALL-LISTP X)** provided it can establish **(LISTP A)**. Again, while this extremely brief introduction to the logic and theorem prover should suffice as a prerequisite for the rest of the paper, the reader is welcome to consult [3] for a much more thorough treatment.

1.2 Remarks on methodology

General hints on how to use the Boyer-Moore theorem prover may be found in the user's manual [3], particularly in Chapter 13. We also felt free to use PC-NQTHM, an interactive enhancement of the Boyer-Moore theorem prover described in [10, 11], to help explore some of the more difficult theorems. (Examples of such use may be found in [10].) However, the final proof script ultimately does *not* depend on PC-NQTHM, but only on the Boyer-Moore theorem prover with the enhancements for theories, **LET**, quantifiers, and functional variables mentioned above.

Our first completed proof was rather ugly[2] in a number of places. Apparently this phenomenon is rather typical for users of the Boyer-Moore theorem prover, since one is still discovering the proper abstractions and proof structure while carrying out the proofs. In fact, the helpful output of the system can also distract one towards proving lemmas that are geared specifically to allow a particular proof attempt to succeed rather than towards proving elegant, general lemmas. Our first proof did, however, generate a number of basic definitions and rules for the files "sets.events", "alists.events", and "terms.events" which can be found (in their current forms) in the Appendix. So that we could obtain a proof script amenable to this exposition, we did the proof again, starting with those three files. Having those files already loaded allowed many of the proofs to go through automatically, which freed our attention for more substantive matters. In the course of the new proof a few additional basic rules were discovered and the three aforementioned libraries were suitably enhanced during this ‘‘polishing’’ process. Not surprisingly, when we moved some of those new basic rules up to those three files from our final file, some proofs in the final file no longer succeeded; when a rewrite rule is moved in front of a PROVE-LEMMA event, it can affect the course of the event's automatic proof. But we were able to find a few more useful rules for the three preliminary files, without undue difficulty. The resulting proof as it exists in the final file, "generalize.events", is reasonably concise. An advantage of this conciseness is that the result is quite amenable for description in the final two sections of this paper. Perhaps a disadvantage is that some of the struggles in completing the proof have been hidden, though we do make a few remarks about such difficulties where they came up.

We should be honest that although the lemmas stated in the final file "generalize.events" form the heart of the proof (in our view), still many of the supporting lemmas in the other three files are crucial too. A number of those lemmas were not only crucial to the main proof, but in fact were only discovered while trying to do that

[2] even compared to the final version!

proof.[3] The point here is that although the lemmas have been arranged into files for expository purposes, one should *not* be left with the impression that the first three files were created in isolation and then a fairly natural proof evolved without difficulty, as represented in "generalize.events". An unfortunate amount of sweat went into that proof! On the other hand, the original proof took well under a month, including the libraries and the time required to think about the theorem. So although our experience is that this kind of program verification remains a less-than-automatic activity, still we are not too disappointed by the amount of effort required. The exposition in this paper, however, is a different matter; it seemed quite time-consuming. We don't recommend such detailed expositions in general, although we hope that this one has pedagogical value.

We did not keep the set of lemmas in those first three files at a minimum. Rather, we were happy to build up less-than-minimal but useful libraries of rules. Therefore the thickness of the first three files in the Appendix is not entirely indicative of what is truly necessary for the successful processing of the events in the final file. On the other hand, we view the events in the first three files as being sufficiently fundamental that many or all of them should be usable in possible future work that involves notions such as lists, terms and substitutions.

Another obligation arising from honesty requires us to point out that hints to the Boyer-Moore prover have been omitted from the exposition below (although they do appear in the appendix). We simply felt that the hints would distract the reader from more substantive considerations, and would even be misleading in the absence of explanation.

Finally, let us remark that the time required to automatically replay the events constructed for this exercise was roughly an hour and a quarter on a Sun 3/60 with 20 megabytes main memory. Slightly under a half hour was spent on the events in the three preliminary files; the rest was spent on the events in "generalize.events".

1.3 Outline of theorem and proof

The main theorem is stated precisely in Section 4. However, here is a very informal version.

We want to model a proof development methodology similar to the one in PC-NQTHM [10, 11], as explained at the start of the introduction. (In fact, similar ‘‘proof refinement’’ methodologies have been

[3] People familiar with the Boyer-Moore prover will correctly guess that many of these lemmas were thought up by reading failed proof transcripts and thinking about what might be useful to prove as rewrite rules. Others were discovered by crawling around through terms using PC-NQTHM.

implemented in systems preceding PC-NQTHM as well, for example LCF [7] and its ‘‘descendents’’ HOL [8] and Nuprl [5].) In the PC-NQTHM methodology, the user starts with a proof state consisting of a single goal, namely the goal to be proved, and proceeds to create new proof states by ‘‘refining’’ goals into subgoals and simplified goals. The proof is complete when all goals of the state are simply **T** (*true*). Let us explain this more carefully (but still informally).

First imagine a situation where one has a formula in some logic that he wishes to show is a theorem. One approach would be to replace that formula with a list new formulas whose conjunction implies the given formula. (Such a step may be called a ‘‘refinement step’’.) The resulting formulas are then the *goals* that remain to be proved. The first formula in this list, which we will call the *current* (or *top*) goal, may then be similarly *refined* into subgoals that imply *it*, leaving one with those new goals, together with the existing goals other than that current goal. Once a current goal is simply the formula *T* (*true*), it is replaced by the empty list of goals. One would hope to be able to continue this process until there are no goals left, in which case one can conclude that the original goal is a theorem. Such a sequence of steps will be called a ‘‘proof’’, though it is perhaps better viewed as a demonstration that a proof exists in that logic.

We might call the current list of (as yet unproved) goals the ‘‘current *proof state*’’. However, imagine a slightly more general paradigm in which a proof state consists not only of unproved goals but also of a list of variables called the *free variables* of that proof state. The idea is that one should be free to substitute for the free variables. For example, suppose there is a single goal, of the form $t_1 < t_2$. Clearly it suffices to find some z for which $t_1 < z$ and $z < t_2$. So, it should be legal to replace the current goal $t_1 < t_2$ with a list of the two goals $t_1 < z$ and $z < t_2$, with the stipulation that z is to be considered *free*. Then if we are able to find some term u for which we can prove $t_1 < u$ and $u < t_2$, then we will be allowed to substitute u for z and carry out that proof.

Suppose a proof state has the property that there is some way of substituting terms for its free variables, into its goals, such that the resulting goals are all theorem. Such a proof state will be called *valid*. The ‘‘key lemma’’ for a proof of correctness of such a refinement-based system would establish that each refinement transformation has the following property: whenever the new state is ‘‘valid’’ in an appropriate sense, then the given state is ‘‘valid’’. For then an easy induction would let one conclude that if one performs a series of such state transformations, starting with the user’s given goal and resulting in a state where all goals are disposed of, then (as such a final state is presumably ‘‘valid’’) the original state is ‘‘valid’’ -- and hence, presumably, the original goal is a theorem.

Such a refinement system may have a number of legal refinement steps, so for a correctness result of the type described in the previous paragraph, one would have to prove a ‘‘key lemma’’ for each of these. We confine ourselves in this paper to such a proof for a single refinement step that we call *generalization*. The idea is that if one wishes to prove a goal **g** containing a subterm **t**, it should be legal to replace **t** in **g** by a new variable. Standard logics have the property that if the result is a theorem, then the original goal is a theorem.

There is a subtlety which makes this correctness proof not completely trivial, namely, generalization in this sense is not *sound* in general, i.e. the aforementioned key lemma may fail to hold. The problem has to do with free variables, and examples are given in Subsection 4.1. Rather than get into details at this point, let us simply state that there is a way to define generalization so that it is correct and reasonable.

The main theorem in this paper states the correctness of a formalization of generalization in this context.

62. Theorem. **GENERALIZE-IS-CORRECT**

```
generalize-okp(sg,state) ∧ valid-state(generalize(sg,state))
→ valid-state(state)

(IMPLIES (AND (GENERALIZE-OKP SG STATE)
              (VALID-STATE (GENERALIZE SG STATE)))
         (VALID-STATE STATE))
```

Here **GENERALIZE-OKP** is a predicate which may be viewed as a precondition under which the user is allowed to apply the **GENERALIZE** refinement rule. We also prove the much simpler ‘‘sanity’’ theorem, saying that if generalization is legal then the result is still a state. We’ll say no more about this, except to mention that it could be useful in case we wish (someday) to extend the current theorem to handle a sequence of PC-NQTHM-like commands.

13. Proposition. **GENERALIZE-STATEP**

```
(IMPLIES (GENERALIZE-OKP SG STATE)
         (STATEP (GENERALIZE SG STATE)))
```

The function **GENERALIZE** is actually rather subtle, and the proof is more subtle than one might initially expect. Our approach in the mechanized proof-checking exercise was to break this theorem into major subtheorems, some of which were broken down further, and so on. In each case we checked that the theorem followed from its subtheorems, by adding the subtheorems as (temporary) axioms and running the Boyer-Moore theorem prover on the desired theorem (after proving minor subtheorems on which the theorem also depends; these are omitted in the diagram below). This approach will appear upon inspection of the file "generalize.events" in the Appendix. In fact this top-level structure of the proof is rather evident upon inspection of the final file "generalize.events" in the Appendix, and is also evident in the structure of the final section of the paper. Here is a brief summary, for convenience. We refer to the theorems by name as well as by the numbers associated with them in the file "generalize.events".

```
                              generalize-is-correct
                                        |
                                        |
                              main-theorem-1
                                        |
                                        |
                                     main-theorem-1-case-4
                                   /                       \
                                  /                         \
                  main-hyps-relieved                       main-hyps-suffice
                   /            \                          /              \
                  /              \                        /                \
main-hyps-relieved-5 main-hyps-relieved-6                /                  \
                     /             \    main-hyps-suffice-first main-hyps-suffice-rest
                    /               \
                   /                 \
      main-hyps-relieved-6-first   .....main-hyps-relieved-6-rest.....
                                    /               |                 \
                                   /                |                  \
                                  /  main-hyps-relieved-6-rest-lemma-2  \
      main-hyps-relieved-6-rest-lemma-1        main-hyps-relieved-6-rest-generalization
```

1.4 Summary of the rest of the paper

It's problematic how best to describe a proof checked with the Boyer-Moore prover. The appendix at the end of this report contains a complete list of events, including supporting events about sets, alists, terms, and proof theory. However, most readers will only find this list helpful for reference, at best. In the paper proper we outline a proof of the main theorem with a liberal amount of explanation. The development will refer to events in the appendix, but (as indicated above) will also display the events using conventional mathematical notation. Therefore, familiarity with Lisp notation is *not* a prerequisite for being able to follow the treatment here. There actually is one exception that we mention now: semicolons (;) denote the start of comments, so that all characters from a semicolon up to the end of the line should be viewed as informal comments only.

The following section (Section 2) presents the underlying logical preliminaries such as the notion of *term*. That is followed by a presentation in Section 3 of some basic but important lemmas about these notions. Section 4 then presents further notions specific to the theorem in question, culminating with a statement of that theorem. Finally, Section 5 contains a proof of the theorem that closely follows the mechanically-checked proof. Thus, one may view the final section either as being simply an informal proof of the theorem in English or as being a guide to the mechanically-checked proof.

The appendices contain four files of events that replay in the Boyer-Moore theorem prover as extended by notions of theories and **LET** notation (as described in [10]), first-order quantifiers (as described in [12]), and functional variables (as described in [4]). The first three of these sequences of events can be viewed as basic supporting libraries, corresponding to Sections 2 and 3 below. The last file may be viewed as the proof proper, including relevant definitions, and thus corresponds to the final two sections below.

2. Basic Notions

This section presents a number of primitive notions such as those of a *variable*, a *term*, and a *substitution*. Though these are quite standard, we state here the definitions of these notions used in the mechanically-checked proof development. We divide into subsections corresponding to the event files "sets.events", "alists.events", and "terms.events", all of which may be found in the Appendix, where complete definitions may be found. A brief introduction to the Boyer-Moore logic and to some of our conventions in this paper may be found in Subsection 1.1 above.

2.1 Sets

The event file "sets.events" forms the lowest-level foundation for our proof development. Here is a brief and very informal description of some of the functions defined in that file. The reader is referred to the Appendix for the actual definitions and for a number of basic lemmas. For convenience we indicate ordinary mathematical notation which ‘‘corresponds’’ to these notions. The correspondence isn’t quite accurate since we will feel free to ignore the distinction between sets and lists for this purpose.

- **(LENGTH L)** or **|l|**: the number of elements in the list **L**
- **(SUBSETP X Y)** or **x** $\subseteq$ **y**: equals **T** if every member of the list **x** is a member of the list **y**, otherwise returns **F**
- **(DELETE X L)** or **l \ {x}**: the result of deleting the first occurrence of **X** from the list **L**
- **(DISJOINT X Y)** or **x** $\cap$ **y =** $\emptyset$: equals **T** if **X** and **Y** share no common member, else **F**
- **(INTERSECTION X Y)** or **x** $\cap$ **y**: the subsequence of the list **x** consisting of members of the list **y**
- **(SET-DIFF X Y)** or **x \ y**: the subsequence of the list **x** obtained by removing members of the list **y**
- **(SETP X)** or **setp(x)**: equals **T** if the list **x** contains no duplicates, else **F**
- **(MAKE-SET X)** or **make-set(x)**: a list with no duplicates that has the same members as does **x**

2.2 Alists

Here is a brief and very informal description of some of the functions defined in the file "alists.events". The reader is referred to the Appendix for the actual definitions and for a number of lemmas.

- **(ALISTP X)** or **alistp(x)**: equals **T** if **x** is an *association list* (*alist*), i.e. a list of ordered pairs
- **(DOMAIN MAP)** or **domain(map)**: a list of all first components of ordered pairs from **map**
- **(RANGE MAP)** or **range(map)**: a list of all second components of ordered pairs from **MAP**

- **(VALUE X MAP)** or **map(x)**: the second component of the first ordered pair in **map** whose first component is **x**; we speak of this as being the *value associated with* **x** *in* **map**
- **(INVERT MAP)** or $\mathbf{map^{-1}}$: returns the alist obtained by switching the first and second components of every ordered pair belonging to **map**.
- **(MAPPING MAP)** or **mapping(map)**: equals **T** if **map** is an alist whose **domain** has no duplicates, else **F**
- **(RESTRICT S NEW-DOMAIN)** or **S | new-domain**: the subsequence of **s** consisting of pairs whose first components are members of **new-domain**.
- **(CO-RESTRICT S NEW-DOMAIN)** or **S |~ new-domain**: the subsequence of **s** consisting of pairs whose first components are *not* members of **new-domain**.

2.3 Terms

One typically defines the notion of *term* by recursion: a term is either a *variable* or the *application* of a *function symbol* to a list of terms (of an appropriate length). Our formal definitions of *term* and of various auxiliary notions will parallel this informal recursive one. This subsection is a summary of the file "terms.events", which may be found in the Appendix.

We'll begin with the notion of a *variable*. We could *define* the function **VARIABLEP**, thus specifying it as a unique function. However, we prefer to add an axiom asserting only some reasonable properties of this function, so as not to over-specify the notion of variable. Since the act of simply adding an axiom[4] does not guarantee in general that the resulting theory is consistent, instead we will use an extension of the Boyer-Moore logic reported in [4] which allows an event form called CONSTRAIN. Perhaps the best way to explain CONSTRAIN is in the context of the example displayed below. The event below has name VARIABLEP-INTRO, and the designation (REWRITE) indicates that it is to be stored as one or more rewrite rules. It asserts that no **LISTP** object (i.e. ordered pair) is a variable, and that **VARIABLEP** returns a boolean value. The last argument of CONSTRAIN below, namely **((VARIABLEP NLISTP))**, instructs the system to show that this axiom is consistent by showing that it holds when **VARIABLEP** is replaced by the function **NLISTP** (which is a predicate holding of objects that are not ordered pairs). Thus, we'll refer to this argument of a CONSTRAIN event as the *witnessing alist*. In fact, use of CONSTRAIN guarantees more than consistency -- it guarantees *conservativity*, in that no new theorems can be proved for the existing history in the presence of this axiom (see [4] for more on this).

---

[4]with the Boyer-Moore event type ADD-AXIOM

Introduction *of* **VARIABLEP.**

¬ variablep(<a,b>) ∧ (variablep(x) = T ∨ variablep(x) = F)

```
(CONSTRAIN VARIABLEP-INTRO (REWRITE)
           (AND (IMPLIES (LISTP X)
                         (NOT (VARIABLEP X)))
                (OR (TRUEP (VARIABLEP X))
                    (FALSEP (VARIABLEP X))))
           ((VARIABLEP NLISTP)))
```

The function **VARIABLE-LISTP** recognizes lists of variables. We use the standard mechanism for representing quantification over lists in the Boyer-Moore logic, namely, primitive recursion.

Definition *of* **VARIABLE-LISTP**

variable-listp(x) ≡ (∀ v ∈ x) variablep(v)

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

The next notion auxiliary to the notion of *term* is that of a *function symbol.* It is not important for the development that follows to know anything about the notion of a function symbol except that there is at least one 0-place function symbol (i. e. constant symbol), which we call **(FN)**. Below is the appropriate CONSTRAIN event, which introduces **FUNCTION-SYMBOLP** and **FN** and asserts that **(FN)** is a function symbol. Notice that the ‘‘witnessing alist’’ suggests that the prover check this axiom with **FUNCTION-SYMBOL-P** replaced by **LITATOM** and with **FN** replaced by the constant function that returns the literal atom **’ZERO**.

Introduction *of* **FUNCTION-SYMBOL-P.**

Let **(FN)** *be an arbitrary function symbol, where for example*
(FN) *could be* **’ZERO** *and* **FUNCTION-SYMBOLP** *could be* **LITATOM.**

```
(CONSTRAIN FUNCTION-SYMBOL-INTRO (REWRITE)
           (FUNCTION-SYMBOL-P (FN))
           ((FUNCTION-SYMBOL-P LITATOM)
            (FN (LAMBDA () ’ZERO))))
```

Now in order to define the notion of a term one has to define the notion of a list of terms as well. We will define these using mutual recursion, employing a standard trick for representing mutually recursive definitions in the Boyer-Moore logic: if **FLG** is not **F** then **(TERMP FLG X)** asserts that **X** is a term (informally, **termp(x)**), and otherwise **(TERMP FLG X)** asserts that **X** is a list of terms (informally, **termp-list(x)**).[5]

[5]Some Boyer-Moore prover users like to use ’LIST and T for the two explicitly-mentioned values of the **FLG** parameter in such situations. However, we found that a heuristic for defeating excessive backchaining was defeating some of our rewrite rules in that case.

Definition *of* **TERMP (***and* **termp-list)**

$\mathbf{termp(x)} \equiv$
$[\ \mathbf{variablep(x)} \vee$
$\quad \mathbf{termp(y_1)} \wedge \ldots \wedge \mathbf{termp(y_n)}$
$\qquad$ *where* $\mathbf{x}$ *is* $\mathbf{<f\ y_1 \ldots y_n>}$ *and* $\mathbf{function\text{-}symbolp(f)}\ ]$
$\mathbf{termp\text{-}list(<y_1,\ \ldots\ y_n>))} \equiv [\mathbf{termp(y_1)} \wedge \ldots \wedge \mathbf{termp(y_n)}]$

```
(DEFN TERMP (FLG X)
  (IF FLG
      (IF (VARIABLEP X)
          T
        (IF (LISTP X)
            (AND (FUNCTION-SYMBOL-P (CAR X))
                 (TERMP F (CDR X)))
          F))
    (IF (LISTP X)
        (AND (TERMP T (CAR X))
             (TERMP F (CDR X)))
      (EQUAL X NIL))))
```

The function **ALL-VARS** returns a list of all variables in **x**, where **x** is a term if **flg** is not **F** and a list of terms if **flg** is **F**). It does not bother to eliminate duplicates.

Definition *of* **ALL-VARS**

If x is a term, then
$\mathbf{all\text{-}vars(x) = \{x\}}$ *if x is a variable, else*
$\qquad\qquad \cup\ \{\mathbf{all\text{-}vars(y)}:\ \mathbf{y}$ *is an argument of* $\mathbf{x}\}$

$\mathbf{all\text{-}vars(<x_1,\ \ldots,\ x_n>)} = \cup\ \{\mathbf{all\text{-}vars(x_i)}:\ 1 \le i \le n\}$

```
(DEFN ALL-VARS (FLG X)
  (IF FLG
      (IF (VARIABLEP X)
          (LIST X)
        (IF (LISTP X)
            (ALL-VARS F (CDR X))
          NIL))
    (IF (LISTP X)
        (APPEND (ALL-VARS T (CAR X))
                (ALL-VARS F (CDR X)))
      NIL)))
```

We also need to implement some notion of *instantiation*. A *substitution* is essentially a function that maps terms to terms, represented as a list of term pairs. Of particular interest is the class of *variable substitutions*, where the domain consists of variables:

Definition *of* **VAR-SUBSTP**

$\mathbf{var\text{-}substp(s)} \equiv \mathbf{mapping(s)} \wedge \mathbf{variable\text{-}listp(domain(s))} \wedge \mathbf{termp\text{-}list(range(s))}$

```
(DEFN VAR-SUBSTP (S)
  (AND (MAPPING S)
       (VARIABLE-LISTP (DOMAIN S))
       (TERMP F (RANGE S))))
```

Given a substitution **s** (not necessarily a variable substitution), we define the *instantiation*

$$\mathbf{x\ /\ s}$$

of a term (or term list) **x** *under* the substitution **s** as follows. Notice that we follow the usual convention with respect to the parameter **flg**, namely if **flg** is **F** then **x** is a list of terms, and otherwise **x** is a single term.

Definition *of* **SUBST**

```
If  x  is a term, then:
x / s = s(x)   if  x ∈ domain(s); else,
        x      if  variablep(x); else,
        <f y_1/s ... y_n/s>  if  x  is  <f y_1 ... y_n>
If  x  is a list  <y_1 ... y_n> of terms, then
x / s = <y_1/s ... y_n/s>
```

```
(DEFN SUBST (FLG S X)
  (IF FLG
      (IF (MEMBER X (DOMAIN S))
          (VALUE X S)
        (IF (VARIABLEP X)
            X
          (IF (LISTP X)
              (CONS (CAR X)
                    (SUBST F S (CDR X)))
            ;; silly impossible value of F for non-termp
            F)))
    (IF (LISTP X)
        (CONS (SUBST T S (CAR X))
              (SUBST F S (CDR X)))
      NIL)))
```

The following simple fact is one of many obvious facts that need to be proved. It says that the property of being a term (or term list, if **FLG** is **F**) is preserved by the application of a substitution.

Lemma. **TERMP-SUBST**

```
[termp(x) ∧ termp-list(range(s))]  →  termp(x/s)
[termp-list(x) ∧ termp-list(range(s))]  →  termp-list(x/s)
```

```
(IMPLIES (AND (TERMP FLG X)
              (TERMP F (RANGE S)))
         (TERMP FLG (SUBST FLG S X)))
```

Just as **SUBST** is used to apply a substitution to a term, the function **APPLY-TO-SUBST** is used to apply one substitution to another substitution, i.e. to apply a substitution **s1** to each term in range of another substitution **s2**. We may informally write

s2 // s1

to denote the application of **s1** to **s2** in this sense. Formally, we have:

Definition *of* **APPLY-TO-SUBST**

```
s2 // s1 = {<x,y/s1>: <x,y> ∈ s2} .
```

```
(DEFN APPLY-TO-SUBST (S1 S2)
  (IF (LISTP S2)
      (IF (LISTP (CAR S2))
          (CONS (CONS (CAAR S2) (SUBST T S1 (CDAR S2)))
                (APPLY-TO-SUBST S1 (CDR S2)))
        (APPLY-TO-SUBST S1 (CDR S2)))
    NIL))
```

We may now define the *composition* of substitutions **s1** and **s2**, which we write as **(s1 • s2)**. This is the substitution that, when applied to a term, is the same as the result of first applying the substitution **s1** and then the substitution **s2**. Let us display the definition of composition first in informal notation and then in formal notation. (Here is a minor detail for those familiar with the Boyer-Moore logic or Lisp: the definition of **COMPOSE** in the Boyer-Moore logic may safely use **APPEND** rather than **UNION** because the function **VALUE** only looks for the first occurrence of the key for which the value is to be found.)

Definition *of* **COMPOSE (•)**

```
s1 • s2 = (s1 // s2) ∪ {<x,y> ∈ s2: x ∉ domain(s1)}.

(DEFN COMPOSE (S1 S2)
  (APPEND (APPLY-TO-SUBST S2 S1)
          S2))
```

The following lemma shows that **COMPOSE** behaves similarly to ordinary function composition. We write the lemma both in informal and in formal notation.

Lemma. **COMPOSE-PROPERTY**

```
variable-listp(domain(s2)) ∧ [termp(x) ∨ termp-list(x)]
→
(x / s1) / s2 = x / (s1 • s2)

(IMPLIES (AND (VARIABLE-LISTP (DOMAIN S2))
              (TERMP FLG X))
         (EQUAL (SUBST FLG S2 (SUBST FLG S1 X))
                (SUBST FLG (COMPOSE S1 S2) X)))
```

The next notion illustrates our first use of quantifiers in this development. An extension of the Boyer-Moore logic and prover by first-order quantification is reported in [12]. Briefly, the idea is that there is a new event DEFN-SK, where the suffix ‘‘-SK’’ refers to *Skolemization*, a well-known means for removing quantifiers that was invented by the logician Thoralf Skolem. Every DEFN-SK event in fact adds quantifier-free axioms that uniquely define the indicated function symbol in a conservative extension (cf. 2.3) of the existing history. The DEFN-SK event below asserts that **TERM1** is an *instance* of **TERM2**, with the usual convention that **FLG** indicates whether these are terms or term lists.

Definition *of* **INSTANCE**

```
instance(term1,term2) ≡ (∃ s) [var-substp(s) ∧ (term1 = term2/s)]

(DEFN-SK INSTANCE (FLG TERM1 TERM2)
  (EXISTS ONE-WAY-UNIFIER
          (AND (VAR-SUBSTP ONE-WAY-UNIFIER)
               (EQUAL TERM1 (SUBST FLG ONE-WAY-UNIFIER TERM2)))))
```

In fact the system adds the following axiom to ‘‘implement’’ this definition. The first conjunct gives a *sufficient* condition for **TERM1** to be an instance of **TERM2**: if **TERM1** is the result of substituting a variable substitution **ONE-WAY-UNIFIER** into **TERM2**, then **TERM1** is an instance of **TERM2**. The second conjunct gives a *necessary* condition for **TERM1** to be an instance of **TERM2** (i.e. gives a consequence of **instance(term1,term2)**): if **TERM1** is an instance of **TERM2** then **(ONE-WAY-UNIFIER FLG TERM1 TERM2)** is a variable substitution such that **TERM1** is the result of instantiating **TERM2** with this substitution. Let us state the axiom both in informal and in formal notation. In the informal version we will write the second conjunct in the natural order rather than the contraposed order of the formal version (which is stated that way for technical reasons related to rewriting). The function **ONE-WAY-UNIFIER** is what is generally called a *Skolem function*, in that its only given property is that it provides a witness (in this case, to the existence of an appropriate substitution).

Axiom *added for* **INSTANCE**

```
[(var-substp(s) ∧ term1 = term2/s)  →  instance(term1,term2)]
∧
[instance(term1,term2)  →  (var-substp(s0) ∧ term1 = term2/s0)]
where
s0 = one-way-unifier(term1,term2)
```

```
(AND (IMPLIES (AND (VAR-SUBSTP ONE-WAY-UNIFIER)
                   (EQUAL TERM1
                          (SUBST FLG ONE-WAY-UNIFIER TERM2)))
              (INSTANCE FLG TERM1 TERM2))
     (IMPLIES (NOT (AND (VAR-SUBSTP (ONE-WAY-UNIFIER FLG TERM1 TERM2))
                        (EQUAL TERM1
                               (SUBST FLG
                                      (ONE-WAY-UNIFIER FLG TERM1 TERM2)
                                      TERM2))))
              (NOT (INSTANCE FLG TERM1 TERM2))))
```

Our final definition from "terms.events" is rather idiosyncratic to the application at hand; it will be used to construct a substitution that is used in the proof of the main theorem. **nullify-subst(s)** is a substitution that maps the domain of **S** to the constant term **(FN)**.

Definition *of* **NULLIFY-SUBST**

```
nullify-subst(s) = {<x,c>:  <x,y> ∈ s}
                   where c is a fixed constant symbol
```

```
(DEFN NULLIFY-SUBST (S)
  (IF (LISTP S)
      (IF (LISTP (CAR S))
          (CONS (CONS (CAAR S) (LIST (FN)))
                (NULLIFY-SUBST (CDR S)))
        (NULLIFY-SUBST (CDR S)))
    NIL))
```

3. Some Basic Supporting Lemmas

In order to complete our mechanically-checked proof of the main theorem, we required a number of lemmas about the notions introduced above. We present some of those in this section, for two reasons. First, these lemmas give a flavor of the kinds of lemmas that appear in the libraries for this effort -- "sets.events", "alists.events", and "terms.events" -- and more generally, in other libraries as well. Second, we refer to these lemmas in some of the proofs that come later, but do not wish to clutter the exposition there with such trivial considerations. By the way, this is meant to be a representative list, not an exhaustive one.

The first lemma says that application of a substitution does not affect the domain.

Lemma. **DOMAIN-APPLY-TO-SUBST** *from* "terms.events"

```
domain(s2 // s1) = domain(s2)

(EQUAL (DOMAIN (APPLY-TO-SUBST S1 S2))
       (DOMAIN S2))
```

The next lemma says that a substitution has no effect when its domain contains no variables occuring in the term to which it is applied.

Lemma. **SUBST-NOT-OCCUR** *(from* "terms.events"*)*

```
(termp(x) ∨ termp-list(x)) ∧ variable-listp(domain(s)) ∧ domain(s) ∩ all-vars(x)
→  x/s = x

(IMPLIES (AND (TERMP FLG X)
              (VARIABLE-LISTP (DOMAIN S))
              (DISJOINT (DOMAIN S) (ALL-VARS FLG X)))
         (EQUAL (SUBST FLG S X) X))
```

The following lemmas say that there is no effect when restricting (respectively, co-restricting) a substitition **s** to a subset **x**, as long as all of the variables of the term **term** to which **s** is applied belong to (repectively, do not belong to) **x**. (In fact, they say that it is sufficient that none of those variables belong to **x ∩ domain(s)**.)

Lemma. **SUBST-RESTRICT** *(from* "terms.events"*)*

```
( domain(s) ∩ all-vars(term) ⊆ x ∧
  variable-listp(domain(s)) ∧
  [termp(term) ∨ termp-list(term)] )
→  term / (x | s) = term / s

(IMPLIES (AND (SUBSETP (INTERSECTION (DOMAIN S) (ALL-VARS FLG TERM))
                       X)
              (VARIABLE-LISTP (DOMAIN S))
              (TERMP FLG TERM))
         (EQUAL (SUBST FLG (RESTRICT S X) TERM)
                (SUBST FLG S TERM))))
```

Lemma. **SUBST-CO-RESTRICT** *(from* "terms.events"*)*

```
( x ∩ domain(s) ∩ all-vars(term) = ∅ ∧
  variable-listp(domain(s)) ∧
  [termp(term) ∨ termp-list(term)] )
→  term / (x |~ s) = term / s

(IMPLIES (AND (DISJOINT X
                        (INTERSECTION (DOMAIN S) (ALL-VARS FLG TERM)))
              (VARIABLE-LISTP (DOMAIN S))
              (TERMP FLG TERM))
         (EQUAL (SUBST FLG (CO-RESTRICT S X) TERM)
                (SUBST FLG S TERM)))
```

Two related lemmas say that one can drop a part of a subsitution whose domain does not intersect the term in question.

Lemma. **SUBST-APPEND-NOT-OCCUR-1** *(from* "terms.events"*)*

```
[ (termp(x) ∨ termp-list(x)) ∧
  variable-listp(domain(s1)) ∧
  all-vars(domain(s1)) ∩ all-vars(x) = ∅ ]
→
x / (s1 ∪ s2)  =  x / s2

(IMPLIES (AND (TERMP FLG X)
              (VARIABLE-LISTP (DOMAIN S1))
              (DISJOINT (ALL-VARS F (DOMAIN S1))
                        (ALL-VARS FLG X)))
         (EQUAL (SUBST FLG (APPEND S1 S2) X)
                (SUBST FLG S2 X)))
```

Lemma. **SUBST-APPEND-NOT-OCCUR-2** *(from* "terms.events"*)*

```
[ (termp(x) ∨ termp-list(x)) ∧
  variable-listp(domain(s2)) ∧
  all-vars(domain(s2)) ∩ all-vars(x) = ∅ ]
→
x / (s1 ∪ s2)  =  x / s1

(IMPLIES (AND (TERMP FLG X)
              (VARIABLE-LISTP (DOMAIN S2))
              (DISJOINT (ALL-VARS F (DOMAIN S2))
                        (ALL-VARS FLG X)))
         (EQUAL (SUBST FLG (APPEND S1 S2) X)
                (SUBST FLG S1 X)))
```

The following rewrite rule is kept in a disabled state, meaning that it is not used by the Boyer-Moore prover except when a hint is given to *enable* this rule. It is very useful when trying to prove that two lists do not intersect, because it reduces that problem to the problem of showing that nothing can belong to both lists. Functions such as **DISJOINT-WIT** are often called *definable Skolem functions* in that they provide witnesses to existential assertions (when they hold), in this case the assertion that **x** and **y** are not disjoint.[6]

Lemma **DISJOINT-WIT-WITNESSES** *(from* "sets.events"*)*

```
x ∩ y = ∅
≡  ¬ [ disjoint-wit(x,y) ∈ x ∧ disjoint-wit(x,y) ∈ y ]

(EQUAL (DISJOINT X Y)
       (NOT (AND (MEMBER (DISJOINT-WIT X Y) X)
                 (MEMBER (DISJOINT-WIT X Y) Y))))
```

The following lemma points out the obvious relationship between the domain of a restriction with the domain of the given substitution.

---

[6]The function **DISJOINT-WIT** is actually defined by recursion in "sets.events". The idea of using definable Skolem functions in the Boyer-Moore prover was brought to our attention by Ken Kunen.

Lemma **DOMAIN-RESTRICT** *(from* "alists.events"*)*

domain(s | dom) = domain(s) ∩ dom

```
(EQUAL (DOMAIN (RESTRICT S DOM))
       (INTERSECTION (DOMAIN S) DOM))
```

The remaining lemmas are also rather technical, so we prefer to liist them without comment here.

Lemma. **APPLY-TO-SUBST-IS-NO-OP-FOR-DISJOINT-DOMAIN** *(from* "terms.events"*)*

variable-listp(domain(s1)) ∧ domain(s1) ∩ all-vars(range(s2)) = ∅
→ s2 // s1 = s2

```
(IMPLIES (AND (VARIABLE-LISTP (DOMAIN S1))
              (ALISTP S2)
              (TERMP F (RANGE S2))
              (DISJOINT (DOMAIN S1)
                        (ALL-VARS F (RANGE S2))))
         (EQUAL (APPLY-TO-SUBST S1 S2) S2))
```

Lemma. **VALUE-INVERT-NOT-MEMBER-OF-DOMAIN** *(from* "alists.events"*)*

g ∈ range(sg) ∧ domain(s) ∩ domain(sg) = ∅
→ sg^{-1}(g) ∉ domain(s)

```
(IMPLIES (AND (MEMBER G (RANGE SG))
              (DISJOINT (DOMAIN S) (DOMAIN SG)))
         (NOT (MEMBER (VALUE G (INVERT SG))
                      (DOMAIN S))))
```

Lemma. **VALUE-APPLY-TO-SUBST** *(from* "terms.events"*)*

g ∈ domain(s) → (s // sg)(g) = s(g)/sg

```
(IMPLIES (MEMBER G (DOMAIN S))
         (EQUAL (VALUE G (APPLY-TO-SUBST SG S))
                (SUBST T SG (VALUE G S))))
```

The following obvious fact says that **NULLIFY-SUBST** does not alter the domain of a given substitution.

Lemma. **DOMAIN-NULLIFY-SUBST** *(from* "terms.events"*)*

domain(nullify-subst(s)) = domain(s)

```
(EQUAL (DOMAIN (NULLIFY-SUBST S))
       (DOMAIN S)))
```

Here is another important property of **NULLIFY-SUBST**.

Lemma. **DISJOINT-ALL-VARS-RANGE-APPLY-SUBST-NULLIFY-SUBST** *(from* "terms.events"*)*

termp-list(range(s))
→
domain(sg) ∩ all-vars(range(s // nullify-subst(sg))) = ∅

```
(IMPLIES (TERMP F (RANGE S))
         (DISJOINT (DOMAIN SG)
                   (ALL-VARS F
                             (RANGE (APPLY-TO-SUBST (NULLIFY-SUBST SG)
                                                    S)))))
```

4. Statement of the Main Theorem

In this section we state our main theorem, which should perhaps be called a ‘‘metatheorem’’, since it’s a theorem about formal theorems. The definitions in this section are all taken from the file "generalize.events", which is the last file in the Appendix. The events in that file have been numbered, and we give those numbers in the presentation below.

The first subsection below gives an outline of the high-level motivation for the definitions that follow. This is followed by a presentation of the definitions required for the statement of the main theorem. Some abbreviations are introduced in the third subsection. We conclude by stating the main theorem.

4.1 Motivation

In the introduction to this paper we discuss the original motivation for this work, which was to increase our confidence in the correctness of a particular algorithm for generalization in the presence of free variables. The following example is taken from the final section of [11]. It shows the necessity, for soundness, of having some restriction on how the GENERALIZE command interacts with the set of free variables of the proof state. Suppose that the history contains the rather silly (but correct) theorem that **[z+1 < z** $\rightarrow$ *C***]** for some contradiction *C*. Then to prove *C*, it suffices to prove **[z+1 < z]** for *some* **z**. In fact **z** here is what we call a *free variable* in PC-NQTHM; this designation has the effect of allowing us to instantiate **z** to be anything we like. Of course there is no value of **z** for which the statement **[z+1 < z]** is a theorem; there had better not be, or else *C* would be a theorem! But suppose we allow ourselves to generalize this goal by replacing **z+1** by some new variable, say **a**. The goal then is **[a < z]**. If **z** were still a *free variable*, then we could instantiate it to be **a+1**, which would leave us with the goal **[a < a+1]**. But this goal *is* a theorem, which is supposed to imply that the original goal *C* is a theorem -- yet, *C* was chosen to be a contradiction!

One way around such a problem is to enforce the following rule: when generalizing with a substitution that replaces terms $\mathbf{t_i}$ with corresponding new variables $\mathbf{v_i}$ (e.g. replaces **z+1** with **a** in the example above), the system removes from the list of free variables any variable that occurs in that substitution (e.g. **z** in that example). However, we can avoid removing quite that many free variables in general. The idea is that we must at least remove from the list of free variables those variables that occur both in the new current goal and in any of the terms being generalized away.

However, that set alone is not enough. Consider the theorem

$$[\mathtt{z+1 < w} \wedge \mathtt{w = z}] \rightarrow C$$

where as above, C is contradictory. The to prove C should be impossible, but we can do it if we can prove ‘‘appropriate’’ instances of the two goals **`[z+1 < w]`** and **`[w = z]`**. Here ‘‘appropriate’’ means ‘‘*via* some substitution whose domain is contained in the set of free variables of the new proof state’’; that set is **`{z, w}`**. After generalizing **`[z+1]`** as in the previous example, we have the two goals **`[a < w]`** and **`[w = z]`**. According to plan outlined just above, since **`z`** does not occur in the current goal **`[a < w]`** we may retain it on the list of free variables, and since **`w`** does not occur in the term **`[z+1]`** that was generalized away we may retain it on the list too. But now if we instantiate both **`w`** and **`z`** with **`a+1`** then we can prove the resulting goals, a contradiction.

Here is an informal statement of the main result; a precise statement is of course the topic of the rest of this Section. This material is adapted from Subsection 4.3 of [11]. We defer to the proof presented in Section 5 below further motivation behind choices made here.

- Fix a proof state **`state`**, i.e. a list of terms (*goals*) together with a list of *free variables*.
- Let **`sg`** be a variable substitution.
- Let **`state'`** be the result of applying the GENERALIZE command, with substitution **`sg`** mapping new variables to terms. Thus, the new current (top) goal is the result of substituting the inverse $\mathbf{sg^{-1}}$ of **`sg`** into the current goal of **`state`**, and the remaining goals are unchanged.
- Let **`FREE`** and **`FREE'`** be the respective sets of free variables of **`state`** and **`state'`**.
- Consider the symmetric binary relation $\mathbf{R_0}$ defined on **`FREE`** as follows: $\mathbf{R_0}$(**`v`**,**`w`**) if and only if **`v`** and **`w`** occur in a common goal of **`state'`**.
- Let **`R`** be the transitive closure of $\mathbf{R_0}$.
- Let **`C`** be the range of **`R`** on the intersection of **`FREE`** with the variables of the current goal in **`state'`**.
- Let **`V`** be the set of variables that occur in the range of **`sg`**.

In the second example presented above, **`C`** = **`{z, w}`** and **`V`** = **`{z}`**, so **`C ∩ V = {z}`**. With this example in mind, loosely speaking we want to remove from **`FREE`** the set **`(C ∩ V)`** consisting of all variables from **`FREE`** that both occur in somewhere in the terms being generalized away *and* also have ‘‘anything to do with’’ the new current goal (where ‘‘anything to do with’’ is defined in terms of the equivalence relation **`R`**). The precise relationship specified between **`FREE`** and **`FREE'`** is as follows.

```
FREE' = (FREE \ (C ∩ V)) \ (domain sg)
```

Here then, finally, is what we need to prove. It says that if the state is ‘‘valid’’ after generalization then it was already ‘‘valid’’, where ‘‘valid’’ is as explained in Subsection 1.3: some instance, where only free variables are instantiated, is a theorem.

GENERALIZE SOUNDNESS THEOREM. Let **`G`** be the current goal in proof state **`state`**; let **`P`** be the conjunction of the rest of the goals of **`state`**; let **`sg`** be a substitution mapping some variables not occurring in **`state`** to terms; let **`G'`** = **`G/sg`**$^{-1}$ be the current goal in the new proof state **`state'`**; and let **`FREE`** and **`FREE`**' be the free variables of **`state`** and **`state'`**, respectively. Suppose that for some substitution **`s'`** with domain contained in **`FREE'`**, **`|- (G' & P)/s'`**. Then for some substitution **`s`** with domain contained in **`FREE`**, we have **`|- (G & P)/s`**.

An informal sketch of a proof of this theorem is outlined in [11]. Let us proceed with a careful and rather formal, but (we hope) motivated treatment.

4.2 Definitions for main theorem

Some terms are *theorems* relative to a given history. Here is the axiom that we introduce to capture the essence of ‘‘theoremhood’’; in fact this is the only axiom we introduce about the notion of theorem. As in the introduction of the notions of *variable* and *function symbol* in Subsection 2.3, we use the CONSTRAIN mechanism to guarantee the consistency of these axioms. The first conjunct says that every theorem is a term. The second says that every instance of a theorem by a variable substitution is also a theorem.

1. Introduction *of* **THEOREM.**

```
[theorem(x) → termp(x)]
∧
[(theorem(x) ∧ var-substp(x)) → → theorem(x/s)]

(CONSTRAIN THEOREM-INTRO (REWRITE)
  (AND (IMPLIES (AND (THEOREM X)
                     FLG)
                (TERMP FLG X))
       (IMPLIES (AND (THEOREM X)
                     FLG
                     (VAR-SUBSTP S))
                (THEOREM (SUBST FLG S X))))
  ((THEOREM (LAMBDA (X) F))))
```

The corresponding notion of a *list of theorems* is obvious, and has properties (not listed here; see event #3 in the Appendix) analogous to those for **`THEOREM`** in the event above.

2. Definition *of* **THEOREM-LIST**

```
theorem-list(x) ≡ (∀ y ∈ x) theorem(y)

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

Next we wish to turn to the notion of a *proof state*, which is essentially a list of goals. We want to model a proof development methodology similar to the one in PC-NQTHM, as explained in the introduction, especially Subsection 1.3. That is, we model a proof state as an ordered pair (a **LISTP**) consisting of a term list (intuitively, a list of goals) together with a list of variables (intuitively, the free variables of that proof state).

4. Definition *of* **STATEP**

```
statep(<goals,free>) ≡ termp-list(goals) ∧ variable-listp(free)

(DEFN STATEP (STATE)
  (AND (LISTP STATE)
       (TERMP F (CAR STATE))
       (VARIABLE-LISTP (CDR STATE))))
```

In order to state our main theorem we need a notion of *valid* state. This definition captures the corresponding notion defined in [11], namely, a *valid* state is a state with the following property: for some variable substitution on a subset of the free variables of the state, if one substitutes that substitution into the goals of the state then the results are theorems. (The event type ‘‘DEFN-SK’’ is discussed above with the definition of **INSTANCE**.)

5. Definition *of* **VALID-STATE**

```
valid-state(<goals,free-vars>) ≡
(∃ s) (var-substp(s) ∧ domain(s) ⊆ free-vars ∧ theorem-list(goals/s))

(DEFN-SK VALID-STATE (STATE)
  (AND (STATEP STATE)
       (EXISTS WITNESSING-INSTANTIATION
          (AND (VAR-SUBSTP WITNESSING-INSTANTIATION)
               (SUBSETP (DOMAIN WITNESSING-INSTANTIATION) (CDR STATE))
               (THEOREM-LIST (SUBST F WITNESSING-INSTANTIATION (CAR STATE))))))
```

This definition adds a Skolem function **WITNESSING-INSTANTIATION**. This function may be thought of as picking out a substitution which, when applied to the goals of a given valid state, yields a list of theorems. Here is the axiom added by the system for the DEFN-SK event above. As with the previous DEFN-SK event introducing the function **INSTANCE**, this property of the witness is expressed by the second conjunct in the following axiom (which is stated in the contrapositive in the formal version, for technical reasons).

Axiom *added for* **VALID-STATE**

```
[ ( statep(state) ∧
    var-substp(s) ∧
    domain(s) ⊆ 2nd(state) ∧
    theorem-list(1st(state)/s) )
  →
  valid-state(state) ]
∧
[ valid-state(state)
  →
  ( statep(state) ∧
    var-substp(s0) ∧
    domain(s0) ⊆ 2nd(state) ∧
    theorem-list(1st(state)/s0) )
  where
  s0 = witnessing-instantiation(state) ]

(AND (IMPLIES (AND (STATEP STATE)
                   (VAR-SUBSTP WITNESSING-INSTANTIATION)
                   (SUBSETP (DOMAIN WITNESSING-INSTANTIATION)
                            (CDR STATE))
                   (THEOREM-LIST (SUBST F WITNESSING-INSTANTIATION
                                        (CAR STATE))))
              (VALID-STATE STATE))
     (IMPLIES (NOT (AND (STATEP STATE)
                        (VAR-SUBSTP (WITNESSING-INSTANTIATION STATE))
                        (SUBSETP (DOMAIN (WITNESSING-INSTANTIATION STATE))
                                 (CDR STATE))
                        (THEOREM-LIST (SUBST F
                                             (WITNESSING-INSTANTIATION STATE)
                                             (CAR STATE)))))
              (NOT (VALID-STATE STATE))))
```

The following definition is auxiliary to **GEN-CLOSURE**. Informally, we can say that given a list **free** of ‘‘free variables’’ along with a list **goals** of terms and a list **vars** of variables (intuitively, a list of variables that we've constructed so far in our process of forming the closure), then **new-gen-vars(goals,free,vars)** is a list of those members of **free** that occur in a goal in **goals** that contains an occurrence of a variable in **vars**.

6. Definition *of* **NEW-GEN-VARS**

```
new-gen-vars(goals,free,vars) =
∪ {free ∩ all-vars(g):  free ∩ all-vars(g) ∩ vars ≠ ∅}

(defn new-gen-vars (goals free vars)
  (if (listp goals)
      ;; see below for explanation of LET
      (let ((current-free-vars (intersection free (all-vars t (car goals)))))
        (if (disjoint current-free-vars vars)
            (new-gen-vars (cdr goals) free vars)
          (append current-free-vars
                  (new-gen-vars (cdr goals) free vars))))
    nil))
```

Notice the use of **LET** above. We use an extension to the syntax of the Boyer-Moore logic in which **LET** has the same meaning as it does in Common Lisp [17]; see the third appendix of the PC-NQTHM manual [10] for details. So for example, **CURRENT-FREE-VARS** in the definition above should be viewed as an abbreviation for **(INTERSECTION FREE (ALL-VARS T (CAR GOALS)))**, i.e. for **free ∩ all-vars(1st(goals))**.

Now we can define the closure referred to above. We may speak of **gen-closure(goals,free,free-vars-so-far)** as ‘‘the **GEN-CLOSURE** of **free-vars-so-far** with respect to **goals** and **free**.’’ The recursive nature of the definition of **GEN-CLOSURE** makes it a bit difficult to express informally; our apologies are probably in order for the rather obscure informal definition below.

10. Definition *of* **GEN-CLOSURE**

gen-closure(goals,free,free-vars-so-far) = x ∩ free, *where*
x *is the least fixed point of the function*
(λ x . [free-vars-so-far ∪ new-gen-vars(goals,free,x)])

```
(DEFN GEN-CLOSURE (GOALS FREE FREE-VARS-SO-FAR)
  (LET ((NEW-FREE-VARS (NEW-GEN-VARS GOALS FREE FREE-VARS-SO-FAR)))
    (IF (SUBSETP NEW-FREE-VARS FREE-VARS-SO-FAR)
        (INTERSECTION FREE-VARS-SO-FAR FREE)
      (GEN-CLOSURE GOALS FREE (APPEND NEW-FREE-VARS FREE-VARS-SO-FAR))))
  ;; the following hint is explained below
  ((LESSP (CARDINALITY (SET-DIFF FREE FREE-VARS-SO-FAR)))))
```

Notice that the definition above is recursive. The Boyer-Moore logic requires a proof in such cases; one might call this a ‘‘termination proof’’. The proof obligation is actually completely precise and need not be understood in the context of termination of some execution, though that’s a reasonable motivation. Informally speaking, the *hint* **(LESSP (CARDINALITY (SET-DIFF FREE FREE-VARS-SO-FAR)))** at the end of the ‘‘DEFN’’ event above instructs the system to prove that the cardinality of **(free \ free-vars-so-far)** decreases on each recursive call of the function **GEN-CLOSURE**. Formally, the proof obligation in this case is as follows.

¬ (new-free-vars ⊆ free-vars-so-far)
→
|free \ (new-free-vars ∪ free-vars-so-far)| < |free \ free-vars-so-far|
where **new-free-vars = new-gen-vars(goals,free,free-vars-so-far)**

```
(LET ((NEW-FREE-VARS (NEW-GEN-VARS GOALS FREE FREE-VARS-SO-FAR)))
     (IMPLIES (NOT (SUBSETP NEW-FREE-VARS FREE-VARS-SO-FAR))
              (LESSP (CARDINALITY (SET-DIFF FREE
                                            (APPEND NEW-FREE-VARS FREE-VARS-SO-FAR)))
                     (CARDINALITY (SET-DIFF FREE FREE-VARS-SO-FAR)))))
```

Inspection of the file "generalize.events" shows that a couple of lemmas were proved to help with the termination proof. In particular, the following lemma is easily proved by the system using induction. (A moment’s reflection will suggest its utility in the proof of the termination goal displayed just above.)

8. Lemma**. NEW-GEN-VARS-SUBSET**

new-gen-vars(goals,free,vars) ⊆ free

```
(SUBSETP (NEW-GEN-VARS GOALS FREE VARS)
         FREE))
```

Now let us formalize the hypothesis under which the **GENERALIZE** command (to be defined shortly) is allowed to be executed. The **GENERALIZE** command is intended to apply the inverse of some variable substituion **sg** to the top goal in the current proof state. Thus in the examples presented earlier in this section, the generalization obtained by replacing **z+1** by **a** is represented by the variable substitution **{<a, z+1>}**. As for the other parameters below: **state** is a proof state, the domain of **sg** is disjoint from the variables occurring in the goals of the **state**, there is at least one goal in the **state**, and the domain of **sg** is disjoint from the free variables of the **state**. We take liberties in the informal version below by writing **state** as **<goals,free>**.

11. Definition *of* **GENERALIZE-OKP**

```
generalize-okp(sg,<goals,free>) ≡
[var-substp(sg) ∧
 statep(<goals,free>) ∧
 domain(sg) ∩ all-vars(goals) = ∅ ∧
 goals ≠ ∅ ∧
 domain(sg) ∩ free = ∅]
```

```
(DEFN GENERALIZE-OKP (SG STATE)
  (AND (VAR-SUBSTP SG)
       (STATEP STATE)
       (DISJOINT (DOMAIN SG)
                 (ALL-VARS F (CAR STATE)))
       (LISTP (CAR STATE))
       (DISJOINT (DOMAIN SG) (CDR STATE))))
```

We define the function **GENERALIZE** to take a substitution **sg** and a proof state **state** and return a new proof state.[7] The goals of the new proof state are the same as the goals of **state** except that the first (i.e. *top*, *current*) goal has been modified by substituting the inverse of the variable substitution **sg** into the first goal of **state**, and the list of free variables has been (possibly) reduced.

12. Definition *of* **GENERALIZE**

```
generalize(sg, <{g} ∪ p, free>) =
<new-g,
 free \ (gen-closure({new-g} ∪ p, free, all-vars(new-g)) ∩ all-vars(range(sg)))>
where   new-g = g/sg^-1
```

```
(DEFN GENERALIZE (SG STATE)
  (LET ((G (CAAR STATE))       ;; the current goal
        (P (CDAR STATE))       ;; the rest of the goals
        (FREE (CDR STATE)))    ;; the free variables
    (LET ((NEW-G (SUBST T (INVERT SG) G)))  ;; the new current goal
      (LET ((DOMAIN-1     ;; potentially "bad" free variables
             (GEN-CLOSURE (CONS NEW-G P)
                          FREE
                          (ALL-VARS T NEW-G))))
        (LET ((NEW-FREE  ;; the new free variables
               (SET-DIFF FREE
                         (INTERSECTION DOMAIN-1 (ALL-VARS F (RANGE SG))))))
          (CONS (CONS NEW-G P)
                NEW-FREE))))))
```

[7]The set **DOMAIN-1** in the definition below is what is called **C** in 4.1 above; the name suggests (and is closely related to) the domain of a substitution **S1** that appears later, during the proof.

4.3 Some abbreviations

Before we state the main theorem, let us introduce some abbreviations for terms that occur repeatedly throughout the rest of this exposition. As usual, we'll use both informal notation and formal notation to introduce these abbreviations. Abbreviations will appear in *italics* font.

```
s = witnessing-instantiation(generalize(sg,state))
S = (WITNESSING-INSTANTIATION (GENERALIZE SG STATE))

goals = 1st(state)
GOALS = (CAR STATE)

g = 1st(goals)
G = (CAR GOALS)

p = 2nd(goals)
P = (CDR GOALS)

free = 2nd(state)
FREE = (CDR STATE)

new-g = g / sg^-1
NEW-G = (SUBST T (INVERT SG) G)

domain-1 = gen-closure(<new-g,p>, 2nd(state), all-vars(new-g))
DOMAIN-1 = (GEN-CLOSURE (CONS NEW-G P)
                        FREE
                        (ALL-VARS T NEW-G))

s1 = s | domain-1
S1 = (RESTRICT S DOMAIN-1)

s2 = (s |~ domain-1) // nullify-subst(sg)
S2 = (APPLY-TO-SUBST (NULLIFY-SUBST SG) (CO-RESTRICT S DOMAIN-1))

gen-inst = (s1 ∪ s2) // (sg // s2)
GEN-INST = (APPLY-TO-SUBST (APPLY-TO-SUBST S2 SG) (APPEND S1 S2))
```

Let us use the abbreviations introduced above to restate the definition of **GENERALIZE**.

12. Definition *of* **GENERALIZE**

```
generalize(sg, <{g} ∪ p, free>) =
<new-g,
 free \ (domain-1 ∩ all-vars(range(sg)))>

(DEFN GENERALIZE (SG STATE)
  (CONS (CONS NEW-G P)
        (SET-DIFF FREE
                  (INTERSECTION DOMAIN-1 (ALL-VARS F (RANGE SG))))))
```

4.4 Statement of main theorem

Finally we can state the main theorem. It says that if the preconditions of the **GENERALIZE** command are met and if the result of applying this command is a valid proof state, then the original proof state is valid.

62. Theorem. **GENERALIZE-IS-CORRECT**

```
generalize-okp(sg,state) ∧ valid-state(generalize(sg,state))
→ valid-state(state)
```

```
(IMPLIES (AND (GENERALIZE-OKP SG STATE)
              (VALID-STATE (GENERALIZE SG STATE)))
         (VALID-STATE STATE))
```

5. Proof of the Main Theorem

In this final section we outline the mechanically-checked proof of the main theorem GENERALIZE-IS-CORRECT displayed above. We actually break this proof into three parts. First we show how to reduce the main theorem to two lemmas. Then we devote the remaining two subsections to the respective proofs of those two lemmas.

5.1 Reducing the theorem to two lemmas

First of all, notice that by definition of **VALID-STATE** it suffices to find some substitution, call it **gen-inst(sg,state)**, for which we can prove the following fact.

61. Lemma. **MAIN-THEOREM-1**

```
generalize-okp(sg,<goals,free>) ∧ valid-state(generalize(sg,<goals,free>))
→
statep(<goals,free>) ∧
var-substp(wit) ∧
domain(wit) ⊆ free ∧
theorem-list(goals/wit)
where
wit = gen-inst(sg,state)
```

```
(LET ((WIT (GEN-INST SG STATE)))
  (IMPLIES (AND (GENERALIZE-OKP SG STATE)
                (VALID-STATE (GENERALIZE SG STATE)))
           (AND (STATEP STATE)
                (VAR-SUBSTP WIT)
                (SUBSETP (DOMAIN WIT) (CDR STATE))
                (THEOREM-LIST (SUBST F WIT (CAR STATE))))))
```

Such a variable substitution **wit = gen-inst(sg,state)** can be constructed as follows (see also Subsection 4.1 for motivation). Let **s** be a variable substitution that witnesses the validity of the state **generalize(sg,state)**. Let **domain-1** be the **GEN-CLOSURE** of the variables occurring in the new current goal (which is the result of applying the inverse of the generalizing substitution to the current goal), with respect to the new goals and the existing list of free variables. Then the desired substitution **gen-inst(sg,state)** may be defined as follows.

14. Definition *of* **GEN-INST**
(Recall that terms in *italics* are abbbreviations. See Subsection 4.3 for an explanation of the abbreviations.)

gen-inst(sg, state) =
(*s1* $\cup$ *s2***) // (sg //** *s2***)**

```
(DEFN GEN-INST (SG STATE)
  (APPLY-TO-SUBST (APPLY-TO-SUBST S2 SG)
                  (APPEND S1 S2)))
```

The first three conjuncts of the conclusion of MAIN-THEOREM-1 are now quite trivial; they correspond to events #15, #17, and #19 in the numbered list of events from "generalize.events" in the Appendix (and are named MAIN-THEOREM-1-CASE-1, MAIN-THEOREM-1-CASE-2, and MAIN-THEOREM-1-CASE-3). The first of these, **statep(state)**, is clear by definition of **GENERALIZE-OKP**. The second, **var-substp(gen-inst(sg,state))**, is clear from the way that **gen-inst(sg,state)** is built from variable substitutions. The third, **domain(wit)** $\subseteq$ **2**nd**(state)**, is also straightforward, though (like many simple results proved with the Boyer-Moore prover) it uses basic ‘‘library’’ facts such as the lemma DOMAIN-APPLY-TO-SUBST (see Section 3 above). A key observation for that case, which is specific to our notion of generalization, is the fact that the set of free variables of the state obtained by applying the **GENERALIZE** command is a subset of the set of free variables of the original state:

18. Lemma**. SUBSETP-CDR-GENERALIZE**

2nd**(generalize(sg,state))** $\subseteq$ **2**nd**(state)**

```
(SUBSETP (CDR (GENERALIZE SG STATE))
         (CDR STATE))
```

It remains then only to check the last of the four cases from the conclusion of MAIN-THEOREM-1, i.e. to prove the following (stated using abbreviations, in italics, from Subsection 4.3).

60. Lemma**. MAIN-THEOREM-1-CASE-4**

generalize-okp(sg,state) $\wedge$ **valid-state(generalize(sg,state))**
$\rightarrow$
theorem-list(*goals* **/** *gen-inst***)**

```
(IMPLIES (AND (GENERALIZE-OKP SG STATE)
              (VALID-STATE (GENERALIZE SG STATE)))
         (THEOREM-LIST (SUBST F GEN-INST GOALS)))
```

The idea now is to introduce a new predicate **MAIN-HYPS** that that abstracts the hypotheses that are needed, and then split the proof into two parts. First, we show that **MAIN-HYPS** implies the result for *arbitrary* substitutions and goals. Second, we show that **MAIN-HYPS** holds of the particular substitutions and goals in question. Thus the first part, MAIN-HYPS-SUFFICE below, states that the bizarre-looking substitution **((s1** $\cup$ **s2) // (sg // s2))** (which however is closely related to the definition of **GEN-INST**) serves to create a list of theorems, assuming that **MAIN-HYPS** holds of the relevant substitutions and goals. The other part, MAIN-HYPS-RELIEVED, shows that **MAIN-HYPS** holds of the necessary substitutions and goals.

Notice that we do not use abbreviations in the first of the following lemmas; as suggested above, it holds of arbitrary substitutions and goals. However, it *is* applied (by the theorem prover's rewriter) under the particular instantiation {**S1** := *S1*, **S2** := *S2*, **GOALS** := *GOALS*}.

27. Lemma. **MAIN-HYPS-SUFFICE**

```
main-hyps(s1,s2,sg,g,p)
→ theorem-list({g} ∪ p / ((s1 ∪ s2) // (sg // s2)))

(IMPLIES (AND (LISTP GOALS)
              (MAIN-HYPS S1 S2 SG (CAR GOALS) (CDR GOALS)))
         (THEOREM-LIST (SUBST F
                              (APPLY-TO-SUBST (APPLY-TO-SUBST S2 SG)
                                              (APPEND S1 S2))
                              GOALS)))
```

59. Lemma. **MAIN-HYPS-RELIEVED**

```
generalize-okp(sg,state) ∧ valid-state(generalize(sg,state))
→
main-hyps(s1,s2,sg,g,p)

(IMPLIES (AND (GENERALIZE-OKP SG STATE)
              (VALID-STATE (GENERALIZE SG STATE)))
         (MAIN-HYPS S1 S2 SG G P))
```

The proof now naturally splits into two parts, one for each of the two lemmas displayed immediately above. We close this subsection with the remaining definitions before turning to the proofs of these two remaining lemmas in the respective subsections below. First, here is the definition of **MAIN-HYPS**.

21. Definition *of* **MAIN-HYPS**

```
main-hyps(s1,s2,sg,g,p) ≡
[ termp(g) ∧
  all-vars(g) ∩ domain(sg) = ∅ ∧
  termp-list(p) ∧
  all-vars(p) ∩ domain(sg) = ∅ ∧
  gen-setting-substitutions(s1,s2,sg) ∧
  theorem-list(({g/sg^-1} ∪ p) / (s1 ∪ s2)) ]

(DEFN MAIN-HYPS (S1 S2 SG G P)
  (AND (TERMP T G)
       (DISJOINT (ALL-VARS T G) (DOMAIN SG))
       (TERMP F P)
       (DISJOINT (ALL-VARS F P) (DOMAIN SG))
       (GEN-SETTING-SUBSTITUTIONS S1 S2 SG)
       (THEOREM-LIST (SUBST F (APPEND S1 S2)
                            (CONS (SUBST T (INVERT SG) G) P)))))
```

The auxiliary function **GEN-SETTING-SUBSTITUTIONS** is defined as follows.

20. Definition *of* **GEN-SETTING-SUBSTITUTIONS**

```
gen-setting-substitutions(s1,s2,sg) ≡
[ var-substp(s1) ∧
  var-substp(s2) ∧
  var-substp(sg) ∧
  domain(s1) ∩ domain(sg) = ∅ ∧
  domain(s2) ∩ domain(sg) = ∅ ∧
  all-vars(range(sg)) ∩ domain(s1) = ∅ ∧
  all-vars(range(s2)) ∩ domain(sg) = ∅ ]
```

```
(DEFN GEN-SETTING-SUBSTITUTIONS (S1 S2 SG)
  (AND (VAR-SUBSTP S1)
       (VAR-SUBSTP S2)
       (VAR-SUBSTP SG)
       (DISJOINT (DOMAIN S1) (DOMAIN SG))
       (DISJOINT (DOMAIN S2) (DOMAIN SG))
       (DISJOINT (ALL-VARS F (RANGE SG))
                 (DOMAIN S1))
       (DISJOINT (ALL-VARS F (RANGE S2)) (DOMAIN SG))))
```

5.2 Proof of the lemma MAIN-HYPS-SUFFICE

Let us state the lemma once again.

27. Lemma. **MAIN-HYPS-SUFFICE**

```
main-hyps(s1,s2,sg,g,p)
→ theorem-list({g} ∪ p / ((s1 ∪ s2) // (sg // s2)))
```

```
(IMPLIES (AND (LISTP GOALS)
              (MAIN-HYPS S1 S2 SG (CAR GOALS) (CDR GOALS)))
         (THEOREM-LIST (SUBST F
                              (APPLY-TO-SUBST (APPLY-TO-SUBST S2 SG)
                                              (APPEND S1 S2))
                              GOALS)))
```

If we apply the definitions of **SUBST** and **THEOREM-LIST** in the expression above, then we see that it suffices to prove the following two properties. (Think of **g** as the current goal and of **p** as the rest of the goals.)

24. Lemma. **MAIN-HYPS-SUFFICE-FIRST**

```
main-hyps(s1,s2,sg,g,p)  →  theorem(g / ( (s1 ∪ s2) // (sg // s2) ))
```

```
(IMPLIES (MAIN-HYPS S1 S2 SG G P)
         (THEOREM (SUBST T
                         (APPLY-TO-SUBST (APPLY-TO-SUBST S2 SG)
                                         (APPEND S1 S2))
                         G)))
```

26. Lemma. **MAIN-HYPS-SUFFICE-REST**

```
main-hyps(s1,s2,sg,g,p)  →  theorem-list(p / ( (s1 ∪ s2) // (sg // s2) ))
```

```
(IMPLIES (MAIN-HYPS S1 S2 SG G P)
         (THEOREM-LIST (SUBST F
                              (APPLY-TO-SUBST (APPLY-TO-SUBST S2 SG)
                                              (APPEND S1 S2))
                              P)))
```

Let us consider these two cases separately.

5.2.1 Proof of the lemma MAIN-HYPS-SUFFICE-FIRST

Consider the first of these two lemmas, MAIN-HYPS-SUFFICE-FIRST. Let us begin by arguing informally for its correctness. The last conjunct of **MAIN-HYPS** implies, assuming the hypothesis of the lemma, that $(g/sg^{-1})/(s1 \cup s2)$ is a theorem. Now every instance of a theorem by a variable substitution is a theorem (by the CONSTRAIN event THEOREM-INTRO, event #1 in "generalize.events"). Then MAIN-HYPS-SUFFICE-FIRST above follows if we can show that the proposed theorem is an instance of $(g/sg^{-1})/(s1 \cup s2)$. The following lemma therefore implies MAIN-HYPS-SUFFICE-FIRST.

23. Lemma. **MAIN-HYPS-SUFFICE-FIRST-LEMMA**

$$\mathtt{termp(g)} \wedge \mathtt{all\text{-}vars(g)} \cap \mathtt{domain(sg)} = \emptyset \wedge \mathtt{gen\text{-}setting\text{-}substitutions(s1,s2,sg)}$$
$$\rightarrow$$
$$\mathtt{g} \,/\, (\,(\mathtt{s1} \cup \mathtt{s2}) \,//\, (\mathtt{sg} \,//\, \mathtt{s2})\,) = ((\mathtt{g} \,/\, \mathtt{sg}^{-1}) \,/\, (\mathtt{s1} \cup \mathtt{s2})) \,/\, (\mathtt{sg} \,//\, \mathtt{s2})$$

```
(IMPLIES (AND (TERMP T G)
              (DISJOINT (ALL-VARS T G) (DOMAIN SG))
              (GEN-SETTING-SUBSTITUTIONS S1 S2 SG))
         (EQUAL (SUBST T
                       (APPLY-TO-SUBST (APPLY-TO-SUBST S2 SG)
                                       (APPEND S1 S2))
                       G)
                (SUBST T (APPLY-TO-SUBST S2 SG)
                       (SUBST T (APPEND S1 S2)
                              (SUBST T (INVERT SG) G)))))
```

Let us see why this lemma holds, and in doing so, discover some of the motivation for the properties embodied in **MAIN-HYPS**. Assume the following hypotheses, the last of which is the inductive hypothesis. Note: we'll see during the proof what we need here about **GEN-SETTING-SUBSTITUTIONS**.

$\mathtt{termp(g)}$

$\mathtt{all\text{-}vars(g)} \cap \mathtt{domain(sg)} = \emptyset$

$\mathtt{gen\text{-}setting\text{-}substitutions(s1,s2,sg)}$

(IH) $\mathtt{g'} \,/\, (\,(\mathtt{s1} \cup \mathtt{s2}) \,//\, (\mathtt{sg} \,//\, \mathtt{s2})\,)$
$= ((\mathtt{g'} \,/\, \mathtt{sg}^{-1}) \,/\, (\mathtt{s1} \cup \mathtt{s2})) \,/\, (\mathtt{sg} \,//\, \mathtt{s2})$
for all subterms $\mathtt{g'}$ of $\mathtt{g}$

The proof now breaks into three cases. We omit a few details but give many others, just to show the kind of considerations required in the mechanically-checked proof.

Case 1: $\mathtt{g} \in \mathtt{range(sg)}$, say $\mathtt{g} = \mathtt{sg(v)}$. Then we have, working with the right side of the goal equation:

((g / sg^{-1}) / (s1 ∪ s2)) / (sg // s2)
= *{definition of v}*
(v / (s1 ∪ s2)) / (sg // s2)
= *{by Lemma* SUBST-NOT-OCCUR, *Section 3, since* **v ∉ domain(s1 ∪ s2)**
by definition of **GEN-SETTING-SUBSTITUTIONS}**
v / (sg // s2)
= *{since* **v ∈ domain(sg)***}*
(sg // s2)(v)
= *{definitions of* **v** *and* **//***}*
g / s2

On the other hand, reducing the left side of the goal equation we have

g / ((s1 ∪ s2) // (sg // s2))
= *{by* SUBST-APPEND-NOT-OCCUR-2 *(cf. Section 3), since by the*
GEN-SETTING-SUBSTITUTIONS *hypothesis we have*
all-vars(range(sg)) ∩ domain(s1) = ∅*}*
g / (s2 // (sg // s2))
= *{by the lemma* APPLY-TO-SUBST-IS-NO-OP-FOR-DISJOINT-DOMAIN *(cf. Section 3), since*
domain(sg) ∩ range(s2) = ∅*}*
g / s2

Case 2: **g ∉ range(sg)** and **variablep(g)**.

((g / sg^{-1}) / (s1 ∪ s2)) / (sg // s2)
= *{since* **g ∉ range(sg)***}*
(g / (s1 ∪ s2)) / (sg // s2)
= *{by the composition rule* COMPOSE-PROPERTY, *cf. Subsection 2.3}*
g / (((s1 ∪ s2) // (sg // s2)) ∪ (sg // s2))
= *{by the lemma* SUBST-APPEND-NOT-OCCUR-2 *(cf. Section 3)}*
g / ((s1 ∪ s2) // (sg // s2))

Case 3: otherwise. Then we may write **g** as **<f v_1 ... v_n>**, and we have:

((g / sg^{-1}) / (s1 ∪ s2)) / (sg // s2)
= *{definition of* **SUBST***, since by the case hypothesis,* **g** *is not in* **domain(sg^{-1})***}*
<f v_1/(s1 ∪ s2)
...
v_n/(s1 ∪ s2)> / (sg // s2)
= *{definition of* **SUBST** *again, since* **sg** *is a variable substitution}*
<f (v_1/(s1 ∪ s2)) / (sg // s2)
...
(v_n/(s1 ∪ s2)) / (sg // s2)>
= *{by the inductive hypothesis}*
<f v_1/((s1 ∪ s2) // (sg // s2))
...
v_n/((s1 ∪ s2) // (sg // s2))>
= *{definition of* **SUBST***}*
g / ((s1 ∪ s2) // (sg // s2))

Actually, a formalization of this proof in the Boyer-Moore logic tends to require one to prove the similar theorem about *lists* of terms by a simultaneous induction. The theorem prover essentially carries out the above argument in proving event #22 in "generalize.events", MAIN-HYPS-SUFFICE-FIRST-LEMMA-GENERAL, which is a generalization we provide of MAIN-HYPS-SUFFICE-FIRST-LEMMA to both terms and term lists. (That is, we leave **flg** uninstantiated.) The cases in the inductive argument correspond to the definition of **SUBST**, so we supply the hint **(INDUCT (SUBST FLG SG-1 G))** for this lemma. Notice that we also give **sg^{-1}** a name, **sg-1**, for the technical reason that such induction hints in the Boyer-Moore prover must have variables in the argument positions.

```
22.  Lemma.  MAIN-HYPS-SUFFICE-FIRST-LEMMA-GENERAL

[ (termp(g) ∨ term-list(g)) ∧
  all-vars(g) ∩ domain(sg) = ∅ ∧
  gen-setting-substitutions(s1,s2,sg) ∧
  ∧ sg-1 = sg^-1 ]
→
g / ( (s1 ∪ s2) // (sg // s2) ) = ((g / sg-1) / (s1 ∪ s2)) / (sg // s2)

(IMPLIES (AND (TERMP FLG G)
              (DISJOINT (ALL-VARS FLG G) (DOMAIN SG))
              (GEN-SETTING-SUBSTITUTIONS S1 S2 SG)
              (EQUAL SG-1 (INVERT SG)))
         (EQUAL (SUBST FLG
                       (APPLY-TO-SUBST (APPLY-TO-SUBST S2 SG)
                                       (APPEND S1 S2))
                       G)
                (SUBST FLG (APPLY-TO-SUBST S2 SG)
                       (SUBST FLG (APPEND S1 S2)
                              (SUBST FLG SG-1 G)))))
```

Finally, let us note that a number of trivial considerations that were ignored here must be dealt with in the mechanical proof. Consider again, for example, the second step in the proof of the first case above:

```
(v / (s1 ∪ s2)) / (sg // s2)
=   {by Lemma SUBST-NOT-OCCUR, Section 3, since  v ∉ domain(s1 ∪ s2)
    by definition of GEN-SETTING-SUBSTITUTIONS}
v / (sg // s2)
```

Why do we know that **v ∉ domain(s1 ∪ s2)**? The reason above is ‘‘by definition of **GEN-SETTING-SUBSTITUTIONS**’’. If we think carefully here then we realize that this use of the definition of **GEN-SETTING-SUBSTITUTIONS** guarantees that the domain of **sg** is disjoint from the domains of **s1** and **s2**; so at the very least we need to know that **v**, i.e. **sg^{-1}(g)**, is a member of the domain of **sg**. A lemma to this effect, VALUE-INVERT-NOT-MEMBER-OF-DOMAIN, has been included in Section 3. Another example where we glossed over small details is in the following step from Case 1:

```
(sg // s2)(v)
=   {definitions of v and //}
g / s2
```

In fact we proved a lemma to accomplish this bit of reasoning; see VALUE-APPLY-TO-SUBST in Section 3 (with the confusing instantiation **g** := **v**, **sg** := **s2**, **s** := **sg**).

5.2.2 Proof of the lemma MAIN-HYPS-SUFFICE-REST

Recall that the other half of proving MAIN-HYPS-SUFFICE is:

26. Lemma. **MAIN-HYPS-SUFFICE-REST**

```
goals ≠ ∅ ∧ main-hyps(s1,s2,sg,g,p)
→ theorem-list(p / (s1 ∪ s2) // (sg // s2))

(IMPLIES (MAIN-HYPS S1 S2 SG G P)
         (THEOREM-LIST (SUBST F
                              (APPLY-TO-SUBST (APPLY-TO-SUBST S2 SG)
                                              (APPEND S1 S2))
                              P)))
```

Again we may use the property that an instance of theorem (or theorem list) is a theorem (or theorem list, respectively). Therefore the property above follows from the following lemma, with **FLG** set to **F** and **S** set to **(APPEND S1 S2)** (informally, **s1 ∪ s2**), together with the definition of **MAIN-HYPS**.

25. Lemma. **MAIN-HYPS-SUFFICE-REST-LEMMA**

```
termp(p) ∧ variable-listp(domain(sg)) ∧ all-vars(p) ∩ domain(sg) = ∅
→
p / (s // (sg // s2)) = (p / s) / (sg // s2)

(IMPLIES (AND (TERMP FLG P)
              (VARIABLE-LISTP (DOMAIN SG))
              (DISJOINT (ALL-VARS FLG P) (DOMAIN SG)))
         (EQUAL (SUBST FLG
                       (APPLY-TO-SUBST (APPLY-TO-SUBST S2 SG)
                                       S)
                       P)
                (SUBST FLG
                       (APPLY-TO-SUBST S2 SG)
                       (SUBST FLG S P))))
```

This is actually quite a straightforward result, using the rewrite rule COMPOSE-PROPERTY displayed in Subsection 2.3 above. Here is an informal sketch of the proof (but note that the theorem prover proves this automatically from the previously proved rules).

(p / s) / (sg // s2)
= *{by* COMPOSE-PROPERTY*}*
p / (s • (sg // s2))
= *{by definition of* **COMPOSE***}*
p / (s // (sg // s2)) ∪ {<x,y> ∈ (sg // s2): x ∉ domain(s)}.
= *{by* SUBST-APPEND-NOT-OCCUR-2 *(cf. Section 3), since by hypothesis no variable occurring in* **p** *is in* **domain(sg // s2)***, i.e. in* **domain(sg)** *(see* DOMAIN-APPLY-TO-SUBST *in Section 3)}*
p / (s // (sg // s2))

5.3 Proof of the lemma MAIN-HYPS-RELIEVED

The only thing left to prove is the lemma MAIN-HYPS-RELIEVED. Let us repeat the statement of that lemma, but opening up the definition of **MAIN-HYPS**. We will continue to use the abbreviations introduced in Subsection 4.3 above.

59. Lemma**.** **MAIN-HYPS-RELIEVED** *(with* **MAIN-HYPS** *opened up)*

generalize-okp(sg,state) $\wedge$ **valid-state(generalize(sg,state))**
$\rightarrow$
termp(g**)** $\wedge$
all-vars(g**)** $\cap$ **domain(sg) =** $\varnothing$ $\wedge$
termp-list(p**)** $\wedge$
all-vars(p**)** $\cap$ **domain(sg) =** $\varnothing$ $\wedge$
gen-setting-substitutions($s1$**,**$s2$**,sg)** $\wedge$
theorem-list(({$new\text{-}g$**}** $\cup$ p**) / (**$s1$ $\cup$ $s2$**))**

```
(IMPLIES (AND (GENERALIZE-OKP SG STATE)
              (VALID-STATE (GENERALIZE SG STATE)))
         (AND (TERMP T G)
              (DISJOINT (ALL-VARS T G) (DOMAIN SG))
              (TERMP F P)
              (DISJOINT (ALL-VARS F P) (DOMAIN SG))
              (GEN-SETTING-SUBSTITUTIONS S1 S2 SG)
              (THEOREM-LIST (SUBST F (APPEND S1 S2)
                                     (CONS NEW-G P))))))
```

We thus have six cases to deal with. However, the first four are quite easy; the lemmas MAIN-HYPS-RELIEVED-**n** for **n** = 1, 2, 3, and 4 are events #30 through #33 in the file "generalize.events" (cf. Appendix A). It remains only to prove the other two cases, MAIN-HYPS-RELIEVED-5 and MAIN-HYPS-RELIEVED-6.

5.3.1 Proof of the lemma MAIN-HYPS-RELIEVED-5

Let us first state the lemma MAIN-HYPS-RELIEVED-5.

41. Lemma**.** **MAIN-HYPS-RELIEVED-5**

generalize-okp(sg,state) $\wedge$ **valid-state(generalize(sg,state))**
$\rightarrow$ **gen-setting-substitutions(**$s1$**,**$s2$**,sg)**

```
(IMPLIES (AND (GENERALIZE-OKP SG STATE)
              (VALID-STATE (GENERALIZE SG STATE)))
         (GEN-SETTING-SUBSTITUTIONS S1 S2 SG))
```

Opening up the function **GEN-SETTING-SUBSTITUTIONS** gives us a number of subgoals. The lemmas which follow cover all of these subgoals. Many of them are more general than the corresponding cases of the lemma MAIN-HYPS-RELIEVED-5. For example, in the first lemma below notice that **domain-1** is arbitrary in place of the substitution denoted by the abbreviation *domain-1*. Generality often makes the theorem prover's job easier.

34. Lemma. **MAIN-HYPS-RELIEVED-5-LEMMA-1**

```
valid-state(generalize(sg,state))
→ var-substp(s1) ∧ var-subsp(s2)
where
s1 = s | domain-1
s2 = (s |~ domain-1) // nullify-subst(sg)
```

```
(LET ((S1 (RESTRICT S DOMAIN-1))
      (S2 (APPLY-TO-SUBST (NULLIFY-SUBST SG)
                          (CO-RESTRICT S DOMAIN-1))))
     (IMPLIES (VALID-STATE (GENERALIZE SG STATE))
              (AND (VAR-SUBSTP S1)
                   (VAR-SUBSTP S2))))
```

35. Lemma **MAIN-HYPS-RELIEVED-5-LEMMA-2**

```
generalize-okp(sg,state)  →  var-substp(sg)
```

```
(IMPLIES (GENERALIZE-OKP SG STATE)
         (VAR-SUBSTP SG))
```

For the next two cases from the definition of **GEN-SETTING-SUBSTITUTIONS**, we first observe that the domain of the witnessing substitution *s* is disjoint from the domain of **sg**. This is an easy consequence of the definitions, which imply that **domain(s)** $\subseteq$ **2**$^{\text{nd}}$**(generalize(sg,state))** $\subseteq$ *free* and that *free* is disjoint from the domain of **sg**.

36. Lemma **WITNESSING-INSTANTIATION-IS-DISJOINT-FROM-GENERALIZING-SUBSTITUTION**

```
generalize-okp(sg,state) ∧ valid-state(generalize(sg,state))
→  domain(s) ∩ domain(sg) = ∅
```

```
(IMPLIES (AND (GENERALIZE-OKP SG STATE)
              (VALID-STATE (GENERALIZE SG STATE)))
         (DISJOINT (DOMAIN S) (DOMAIN SG)))
```

Since the domain of any restriction or co-restriction of a substitution is a subset of the original domain, and since the application of a substitution **s** to a substitution **s'** has the same domain as does **s'** (see DOMAIN-APPLY-TO-SUBST in Section 3), the next two cases follow from the lemma displayed just above.

37. Lemma **MAIN-HYPS-RELIEVED-5-LEMMA-3**

```
generalize-okp(sg,state) ∧ valid-state(generalize(sg,state))
→
domain(s1) ∩ domain(sg) = ∅ ∧ domain(s2) ∩ domain(sg) = ∅
where
s1 = s | domain-1
s2 = (s |~ domain-1) // nullify-subst(sg)
```

```
(LET ((S1 (RESTRICT S DOMAIN-1))
      (S2 (APPLY-TO-SUBST (NULLIFY-SUBST SG)
                          (CO-RESTRICT S DOMAIN-1))))
  (IMPLIES (AND (GENERALIZE-OKP SG STATE)
                (VALID-STATE (GENERALIZE SG STATE)))
           (AND (DISJOINT (DOMAIN S1) (DOMAIN SG))
                (DISJOINT (DOMAIN S2) (DOMAIN SG)))))
```

The following lemma MAIN-HYPS-RELIEVED-5-LEMMA-4 handles the next case.

39. Lemma. **MAIN-HYPS-RELIEVED-5-LEMMA-4**

generalize-okp(sg,state) $\wedge$ **valid-state(generalize(sg,state))**
$\rightarrow$ **all-vars(range(sg))** $\cap$ **domain(***s1***) =** $\emptyset$

```
(IMPLIES (AND (GENERALIZE-OKP SG STATE)
              (VALID-STATE (GENERALIZE SG STATE)))
         (DISJOINT (ALL-VARS F (RANGE SG))
                   (DOMAIN S1)))
```

Let us argue informally for the correctness of this lemma. Assuming its hypotheses, we have:

all-vars(range(sg)) $\cap$ **domain(***s1***)**
= *{by definition}*
all-vars(range(sg)) $\cap$ **domain(***s* **|** *domain-1***)**
= *{by* DOMAIN-RESTRICT *in Section 3}*
all-vars(range(sg)) $\cap$ **domain(***s***)** $\cap$ *domain-1*
$\subseteq$ *{by axiom introduced for* **VALID-STATE***}*
all-vars(range(sg)) $\cap$ **2**$^{\text{nd}}$**(generalize(sg,state))** $\cap$ *domain-1*
= *{by definition of* **GENERALIZE***}*
all-vars(range(sg))
  $\cap$ **[***free* **\ (***domain-1* $\cap$ **all-vars(range(sg)))]**
  $\cap$ *domain-1*
= *{trivial set-theoretic reasoning; see below}*
$\emptyset$

How would a person reason in the last step? A natural course would be to consider an arbitrary **x** and show that it if it belongs to **all-vars(range(sg))** and also to **[***free* **\ (***domain-1* $\cap$ **all-vars(range(sg)))]**, then it does not belong to *domain-1*. In fact the analogous fact is proved as a lemma for the intersection displayed two steps earlier in the informal proof above.

38. Lemma. **MAIN-HYPS-RELIEVED-5-LEMMA-4-WIT**

(generalize-okp(sg,state) $\wedge$
  valid-state(generalize(sg,state)) $\wedge$
  wit $\in$ **all-vars(range(sg))** $\wedge$
  wit $\in$ **domain(***s***))**
$\rightarrow$ **wit** $\notin$ *domain-1*

```
(IMPLIES (AND (GENERALIZE-OKP SG STATE)
              (VALID-STATE (GENERALIZE SG STATE))
              (MEMBER WIT (ALL-VARS F (RANGE SG)))
              (MEMBER WIT (DOMAIN S)))
         (NOT (MEMBER WIT DOMAIN-1)))
```

We have one final technical comment on the proof of MAIN-HYPS-RELIEVED-5-LEMMA-4. In addition to proving the lemma MAIN-HYPS-RELIEVED-5-LEMMA-4-WIT first (as a rewrite rule), a hint is also given to enable the lemma DISJOINT-WIT-WITNESSES. That lemma has the effect of reducing the statement that **all-vars(range(sg))** is disjoint from the domain of *s1* to the question issue of whether a particular value could belong to both of them. For a description of that lemma, see Section 3.

The final case goes through automatically, though here it is crucial that *s2* is built using

NULLIFY-SUBST; see the lemma DISJOINT-ALL-VARS-RANGE-APPLY-SUBST-NULLIFY-SUBST in Section 3.

40. Lemma **MAIN-HYPS-RELIEVED-5-LEMMA-5**

generalize-okp(sg,state) ∧ **valid-state(generalize(sg,state))**
→ **all-vars(range(***s2***))** ∩ **domain(sg) =** ∅

```
(IMPLIES (AND (GENERALIZE-OKP SG STATE)
              (VALID-STATE (GENERALIZE SG STATE)))
         (DISJOINT (ALL-VARS F (RANGE S2))
                   (DOMAIN SG)))
```

5.3.2 Proof of the lemma MAIN-HYPS-RELIEVED-6

Now all that is left is the proof of the lemma MAIN-HYPS-RELIEVED-6. Here is its statement.

58. Lemma. **MAIN-HYPS-RELIEVED-6**

generalize-okp(sg,state) ∧ **valid-state(generalize(sg,state))**
→ **theorem-list(({***new-g***}** ∪ *p***) / (***s1* ∪ *s2***))**

```
(IMPLIES (AND (GENERALIZE-OKP SG STATE)
              (VALID-STATE (GENERALIZE SG STATE)))
         (THEOREM-LIST (SUBST F (APPEND S1 S2)
                                (CONS NEW-G P))))
```

Here is a very high-level view of the proof, which incidentally should show why we chose to bring in the notion of ‘‘gen-closure’’ Because of the way that the function **GEN-CLOSURE** is defined (event #10 above), the set *domain-1* has the following property: for every goal **x** in the new state **generalize(sg,state)**, the set of free variables in that state that occur in **x** are either contained in *domain-1* or are disjoint from it. In the former case, which includes the case **x** = *new-g*, no variable occurring in **x** is in the domain of *s2*, and it follows that that **x/(***s1* ∪ *s2***)** = **x/***s1* = **x/***s*. In the latter case we similarly have **x/(***s1* ∪ *s2***)** = **x/***s2*. Since we have already dealt with the case **x** = *new-g*, we may assume that **x** ∈ **p**, and by a little additional technical argument we can show that **x/***s2* is an instance of **x/***s*. So we have that **x/(***s1* ∪ *s2***)** is an instance of **x/***s*, and since **x/***s* is a theorem (by definition of *s* and the **VALID-STATE** hypothesis), so is **x/(***s1* ∪ *s2***)**.

Let us proceed now along the lines of the mechanically-checked proof. By opening up **SUBST** and **THEOREM-LIST** we can break MAIN-HYPS-RELIEVED-6 into the following two goals.

45. Lemma. **MAIN-HYPS-RELIEVED-6-FIRST**

generalize-okp(sg,state) ∧ **valid-state(generalize(sg,state))**
→ **theorem(***new-g* **/ (***s1* ∪ *s2***))**

```
(IMPLIES (AND (GENERALIZE-OKP SG STATE)
              (VALID-STATE (GENERALIZE SG STATE)))
         (THEOREM (SUBST T (APPEND S1 S2) NEW-G)))
```

57. Lemma. **MAIN-HYPS-RELIEVED-6-REST**

generalize-okp(sg,state) $\wedge$ **valid-state(generalize(sg,state))**
$\rightarrow$ **theorem-list(***p* **/ (***s1* $\cup$ *s2***))**

```
(IMPLIES (AND (GENERALIZE-OKP SG STATE)
              (VALID-STATE (GENERALIZE SG STATE)))
         (THEOREM-LIST (SUBST F (APPEND S1 S2) P)))
```

Let us consider these in turn.

5.3.2(1) Proof of the lemma MAIN-HYPS-RELIEVED-6-FIRST. Here is an informal proof of the first of these two lemmas. We begin with a key observation, which we will both prove and use presently.

(*) **domain(***s***)** $\cap$ **all-vars(***new-g***)** $\subseteq$ *domain-1*

Now recall the lemma SUBST-APPEND-NOT-OCCUR-2 (stated in Subsection 5.2 above) which says if no variable of a term **x** belongs to the domain of a substitution **s2** then **x / (s1** $\cup$ **s2) = x / s1**. The domain of *s2* is equal to **domain(s) ** *domain-1*; hence the requirement for SUBST-APPEND-NOT-OCCUR-2 that **domain(***s2***)** be disjoint from **all-vars(***new-g***)** holds by (*).[8] So assuming our hypotheses of **generalize-okp(sg,state)** and **valid-state(generalize(sg,state))**, we may summarize the argument as follows.

new-g **/ (***s1* $\cup$ *s2***)**
= *{by the lemma* SUBST-APPEND-NOT-OCCUR-2 *(cf. Section 3) and* (*)*}*
new-g **/** *s1*
= *{by the lemma* SUBST-RESTRICT *(cf. Section 3) and* (*)*}*
new-g **/** *s*

It remains to check that (*) holds. Consider the following lemma.

44. Lemma **GEN-CLOSURE-CONTAINS-THIRD-ARG**

x $\subseteq$ **(free** $\cap$ **vars)**
$\rightarrow$ **x** $\subseteq$ **gen-closure(goals,free,vars)**

```
(IMPLIES (SUBSETP X (INTERSECTION FREE VARS))
         (SUBSETP X (GEN-CLOSURE GOALS FREE VARS)))
```

If we apply this lemma with **goals** := **{***new-g***}** $\cup$ *p*, **free** = *free*, **vars** = **all-vars(***new-g***)**, and **x** := **domain(***s***)** $\cap$ **all-vars(***new-g***)**, the resulting instance can be expressed using our abbreviations as follows.

[8] The lemmas used in this argument are DOMAIN-CO-RESTRICT from "alists.events" and DISJOINT-SET-DIFF-GENERAL from "sets.events"

```
domain(s) ∩ all-vars(new-g) ⊆ (free ∩ all-vars(new-g))    →    (*)

(IMPLIES (SUBSETP (INTERSECTION (DOMAIN S) (ALL-VARS NEW-G))
                  (INTERSECTION FREE (ALL-VARS NEW-G)))
         (*))
```

So in order to prove (*), it suffices to prove the hypothesis of this implication, which in turn follows from

domain(*s*) $\subseteq$ **2nd(generalize(sg,state))** $\subseteq$ *free*

The first inclusion follows from the fact that the domain of *s* is contained in the free variables of the new (generalized) state, which is part of the **VALID-STATE** hypothesis. The second inclusion is just the lemma SUBSETP-CDR-GENERALIZE from Subsection 5.1 above. This concludes the proof of MAIN-HYPS-RELIEVED-6-FIRST.

In fact we close with one technical comment. The lemma CAR-GENERALIZE is proved before the lemma GEN-CLOSURE-CONTAINS-THIRD-ARG above so as to speed up the proofs. The idea is that we only want to invoke the rather hairy definition of **GENERALIZE** when we are looking at goals, not when we are simply asking about the witnessing substitution.

42. Lemma **CAR-GENERALIZE**

```
1st(generalize(sg, <{g} ∪ p, free>)) =
{g/sg-1} ∪ p

(EQUAL (CAR (GENERALIZE SG STATE))
       (CONS (SUBST T (INVERT SG) (CAAR STATE))
             (CDAR STATE)))
```

5.3.2(2) Proof of the lemma MAIN-HYPS-RELIEVED-6-REST. We now move to the proof of our final goal, which once again is:

57. Lemma. **MAIN-HYPS-RELIEVED-6-REST**

```
generalize-okp(sg,state) ∧ valid-state(generalize(sg,state))
→  theorem-list(p / (s1 ∪ s2))

(IMPLIES (AND (GENERALIZE-OKP SG STATE)
              (VALID-STATE (GENERALIZE SG STATE)))
         (THEOREM-LIST (SUBST F (APPEND S1 S2) P)))
```

Let us begin with the following key notion suggested by the informal proof given above. It asserts that every goals's free variables are either contained in the set **X** or are disjoint from **X**.

46. Definition *of* **ALL-VARS-DISJOINT-OR-SUBSETP**

all-vars-disjoint-or-subsetp(goals,free,x) $\equiv$
($\forall$ **g in goals) [free** $\cap$ **all-vars(g)** $\subseteq$ **x** $\vee$ **free** $\cap$ **all-vars(g)** $\cap$ **x =** $\emptyset$ **]**

```
(DEFN ALL-VARS-DISJOINT-OR-SUBSETP (GOALS FREE X)
  (IF (LISTP GOALS)
      (AND (OR (SUBSETP (INTERSECTION FREE (ALL-VARS T (CAR GOALS)))
                        X)
               (DISJOINT (INTERSECTION FREE (ALL-VARS T (CAR GOALS)))
                         X))
           (ALL-VARS-DISJOINT-OR-SUBSETP (CDR GOALS) FREE X))
    T))
```

Observe that the set of goals *p* has the above property with respect to the free variables of the generalized state and the appropriate ‘‘gen-closure’’, *domain-1*:

52. Lemma. **MAIN-HYPS-RELIEVED-6-REST-LEMMA-2**

generalize-okp(sg,state) $\wedge$ **valid-state(generalize(sg,state))**
$\rightarrow$ **all-vars-disjoint-or-subsetp(***p***, 2**nd**(generalize(sg,state)),** *domain-1***)**

```
(IMPLIES (AND (GENERALIZE-OKP SG STATE)
              (VALID-STATE (GENERALIZE SG STATE)))
         (ALL-VARS-DISJOINT-OR-SUBSETP P (CDR (GENERALIZE SG STATE)) DOMAIN-1))
```

This follows from the definition of *domain-1* and the following observation. Actually, the following lemma relevant in the special case (instance) where **new-free** is **2**nd**(generalize(sg,state))**, i.e. the set of free variables of the new (generalized) state; **free** is *free*; **goals** is *p*; **g** is *g*; and **vars** is *domain-1*.

51. Lemma **ALL-VARS-DISJOINT-OR-SUBSETP-GEN-CLOSURE**

new-free $\subseteq$ **free**
$\rightarrow$
all-vars-disjoint-or-subsetp(p, new-free, gen-closure({g} $\cup$ **p, free, vars))**

```
(IMPLIES (SUBSETP NEW-FREE FREE)
         (ALL-VARS-DISJOINT-OR-SUBSETP
          P NEW-FREE
          (GEN-CLOSURE (CONS G P) FREE VARS)))
```

Let us attempt to finish the proof of our remaining goal MAIN-HYPS-RELIEVED-6-REST with informal reasoning. Assume its hypotheses. Let **x** be any goal in *p*; then by the **VALID-STATE** hypothesis, we have **theorem(x)**. Moreover, the lemma MAIN-HYPS-RELIEVED-6-REST-LEMMA-2 above implies says that the set of free variables occurring in **x** is contained in or disjoint from *domain-1*. Hence there are two cases. We follow the outline given at the start of this subsection 5.3.2 (just below the statement of the lemma MAIN-HYPS-RELIEVED-6).

Case 1: **all-vars(x)** $\subseteq$ *domain-1*. Then since **domain(***s2***)** = **domain(***s***)** **** *domain-1* (by definition of *s2* and the lemmas DOMAIN-CORESTRICT and DOMAIN-APPLY-TO-SUBST in Section 3), it follows from the case hypothesis that

(1) **all-vars(x)** $\cap$ **domain(***s2***)** **=** $\emptyset$.

Therefore

> **x/(***s1* $\cup$ *s2***)**
> **=** *{by* (1) *together with the lemma* SUBST-APPEND-NOT-OCCUR-2 *(cf. Section 3)}*
> **x/***s1*
> **=** *{by* (1) *together with the lemma* SUBST-RESTRICT *(cf. Section 3)}*
> **x/***s*

Case 2: **all-vars(x)** $\cap$ *domain-1* **=** $\emptyset$. The argument is a little more involved in this case, because *s2* is not simply a co-restriction of *s*. Recall that *s2* is defined as **(***s* **|~** *domain-1***) //** **nullify-subst(sg)**. This time we argue as follows.

> **x/(***s1* $\cup$ *s2***)**
> **=** *{by the lemma* SUBST-APPEND-NOT-OCCUR-1 *(cf. Section 3)}*
> **x/***s2*
> **=** *{by definition}*
> **x/((***s* **|~** *domain-1***) // nullify-subst(sg))**
> **=** *{as we will show below}*
> **(x/(***s* **|~** *domain-1***)) / nullify-subst(sg)**
> **=** *{by the lemma* SUBST-CO-RESTRICT *(cf. Section 3)}*
> **(x/***s***) / nullify-subst(sg)**

Therefore **x/(***s1* $\cup$ *s2***)** is a theorem (cf. event #54 in "generalize.events" in the Appendix, which we omit here), since it is an instance of **x/***s* (which is a theorem by the **VALID-STATE** hypothesis). But it remains to explain the reason ‘‘as we will show below’’ for the penultimate step above. The following lemma is the key. It is applied automatically by the theorem prover’s rewriter using the substitution **{sg := nullify-subst(sg), s := (***s* **|~** *domain-1***)}**.

53. Lemma **SUBST-APPLY-TO-SUBST-ELIMINATOR**

```
[ variable-listp(domain(sg)) ∧
  variable-listp(domain(s)) ∧
  termp(x) ∧
  domain(sg) ∩ all-vars(x) = ∅ ]
→
x / (s // sg) = (x / s) / sg

(IMPLIES (AND (VARIABLE-LISTP (DOMAIN SG))
              (VARIABLE-LISTP (DOMAIN S))
              (TERMP T X)
              (DISJOINT (DOMAIN SG) (ALL-VARS T X)))
         (EQUAL (SUBST T (APPLY-TO-SUBST SG S) X)
                (SUBST T SG
                       (SUBST T S X))))
```

Notice that by the lemma DOMAIN-NULLIFY-SUBST (cf. Section 3), we can safely equate the domains of **sg** and **nullify-subst(sg)**. But why can we assume that the domain of **sg** is disjoint from the variables of **x**? Recall that **x** is an arbitrary member of the set of goals from *p* that (by the Case 2 hypothesis) do not intersect *domain-1*. That is, the following lemma should suffice to conclude the proof. (The definition of the function **GOALS-DISJOINT-FROM-VARS** will follow.)

50. Lemma **MAIN-HYPS-RELIEVED-6-REST-LEMMA-1**

```
[ generalize-okp(sg,state) ∧ valid-state(generalize(sg,state)) ]
→
[ domain(sg) ∩
  all-vars(goals-disjoint-from-vars(p, 2nd(generalize(sg,state)), domain-1))
  = ∅]

(IMPLIES (AND (GENERALIZE-OKP SG STATE)
              (VALID-STATE (GENERALIZE SG STATE)))
         (DISJOINT (DOMAIN SG)
                   (ALL-VARS F (GOALS-DISJOINT-FROM-VARS
                                P (CDR (GENERALIZE SG STATE))
                                DOMAIN-1))))
```

And here is the obvious definition of **GOALS-DISJOINT-FROM-VARS**, followed by an important property of this function.

47. Definition *of* **GOALS-DISJOINT-FROM-VARS**

```
goals-disjoint-from-vars(goals,free,vars)  =
{g ∈ goals:  free ∩ all-vars(g) = ∅}

(defn goals-disjoint-from-vars (goals free vars)
  (if (listp goals)
      (let ((current-free-vars (intersection free (all-vars t (car goals)))))
        (if (disjoint current-free-vars vars)
            (cons (car goals)
                  (goals-disjoint-from-vars (cdr goals) free vars))
          (goals-disjoint-from-vars (cdr goals) free vars)))
    nil))
```

48. Lemma **GOALS-DISJOINT-FROM-VARS-SUBSETP**

```
goals-disjoint-from-vars(goals,free,vars) ⊆ goals

(SUBSETP (GOALS-DISJOINT-FROM-VARS GOALS FREE VARS)
         GOALS)
```

Event #49, DISJOINT-ALL-VARS-GOALS-DISJOINT-FROM-VARS, is merely a technical lemma that is necessary because of the theorem prover's difficulty in relieving hypotheses of rewrite rules that contain variables not bound in the conclusion. We omit its statement here.

We conclude by summarizing the top-level structure of the proof of the lemma MAIN-HYPS-RELIEVED-6-REST, which is motivated by the discussion above. This lemma is an immediate consequence of the following lemma, in conjunction with the lemma MAIN-HYPS-RELIEVED-6-REST-LEMMA-1 (event #50) and MAIN-HYPS-RELIEVED-6-REST-LEMMA-2 (event #52), already explained above, which are used to relieve its last two hypotheses. Notice that this lemma is somewhat more abstract than those two, in that it refers to arbitrary values of **p**, **s**, **domain-1**, and **new-free**.

56. Lemma **MAIN-HYPS-RELIEVED-6-REST-GENERALIZATION**

```
[ var-substp(sg) ∧
  var-substp(s) ∧
  domain(s) ⊆ new-free ∧
  termp-list(p) ∧
  theorem-list(p/s) ∧
  domain(sg) ∩ all-vars(goals-disjoint-from-vars(p,new-free,domain-1)) = ∅ ∧
  all-vars-disjoint-or-subsetp(p,new-free,domain-1) ]
→
theorem-list(p/(s1 ∪ s2))
where
s1 = s | domain-1
s2 = (s |~ domain-1) // nullify-subst(sg)

(LET ((S1 (RESTRICT S DOMAIN-1))
      (S2 (APPLY-TO-SUBST (NULLIFY-SUBST SG)
                          (CO-RESTRICT S DOMAIN-1))))
  (IMPLIES (AND (VAR-SUBSTP SG)
                (VAR-SUBSTP S)
                (SUBSETP (DOMAIN S) NEW-FREE)
                (TERMP F P)
                (THEOREM-LIST (SUBST F S P))
                (DISJOINT (DOMAIN SG)
                          (ALL-VARS F (GOALS-DISJOINT-FROM-VARS
                                       P NEW-FREE DOMAIN-1)))
                (ALL-VARS-DISJOINT-OR-SUBSETP P NEW-FREE DOMAIN-1))
           (THEOREM-LIST (SUBST F (APPEND S1 S2) P))))
```

The theorem prover implements informal arguments presented above when proving this theorem by induction on the length of **p**. However, we encountered difficulties at first in finding the right argument, at least during our second proof effort (see Subsection 1.2). The remainder of this section contains an edited version of the comments made during that proof, just after completion of MAIN-HYPS-RELIEVED-6-REST-LEMMA-2 (so that all that was left was the proof of MAIN-HYPS-RELIEVED-6-REST-GENERALIZATION). All of this below may be safely omitted; it's there simply for those familiar with the Boyer-Moore theorem prover who want to dig a little deeper into the details of the proof effort.

5.3.2(3) Some comments on the proof of the lemma MAIN-HYPS-RELIEVED-6-REST-GENERALIZATION. Finally, all that's left is MAIN-HYPS-RELIEVED-6-REST-GENERALIZATION. An attempted proof by induction of that theorem results in 11 goals, all but one of which goes through automatically. The remaining one is as follows.

```
(IMPLIES
 (AND
  (DISJOINT NEW-FREE
            (INTERSECTION DOMAIN-1
                          (ALL-VARS T X)))
  (THEOREM-LIST
   (SUBST F
          (APPEND (RESTRICT S DOMAIN-1)
                  (APPLY-TO-SUBST (NULLIFY-SUBST SG)
                                  (CO-RESTRICT S DOMAIN-1)))
          Z))
  (MAPPING SG)
  (VARIABLE-LISTP (DOMAIN SG))
  (TERMP F (RANGE SG))
  (MAPPING S)
  (VARIABLE-LISTP (DOMAIN S))
  (TERMP F (RANGE S))
  (SUBSETP (DOMAIN S) NEW-FREE)
  (TERMP T X)
  (TERMP F Z)
  (THEOREM (SUBST T S X))
  (THEOREM-LIST (SUBST F S Z))
  (DISJOINT (DOMAIN SG) (ALL-VARS T X))
  (DISJOINT (DOMAIN SG)
            (ALL-VARS F
                      (GOALS-DISJOINT-FROM-VARS Z NEW-FREE DOMAIN-1)))
  (ALL-VARS-DISJOINT-OR-SUBSETP Z NEW-FREE DOMAIN-1))
 (THEOREM (SUBST T
                 (APPEND (RESTRICT S DOMAIN-1)
                         (APPLY-TO-SUBST (NULLIFY-SUBST SG)
                                         (CO-RESTRICT S DOMAIN-1)))
                 X)))
```

Let us attempt to prove this goal with PC-NQTHM, thus seeing why the rewriter can't handle it automatically. With the aid of PC-NQTHM's SHOW-REWRITES command, we see that we would like to rewrite with the lemma SUBST-APPEND-NOT-OCCUR-1 (see Section 3) to replace the conclusion with:

```
(THEOREM (SUBST T
                (APPLY-TO-SUBST (NULLIFY-SUBST SG)
                                (CO-RESTRICT S DOMAIN-1))
                X))
```

However, in order to do that we see (using PC-NQTHM's REWRITE command) that we need to know that under the hypotheses, the following holds.

```
(DISJOINT (ALL-VARS F
                    (DOMAIN (RESTRICT S DOMAIN-1)))
          (ALL-VARS T X))
```

One would think that this follows quite clearly from just two of the hypotheses:

```
(DISJOINT NEW-FREE
          (INTERSECTION DOMAIN-1
                        (ALL-VARS T X)))

(SUBSETP (DOMAIN S) NEW-FREE)
```

This is one of those cases of a problem with free variables in hypotheses that are so annoying. The lemma

DOMAIN-RESTRICT has been proved in "alists.events" (see also Section 3) to help with this. But then we lose the effect of an existing lemma which applied directly to simplify the term **(ALL-VARS F (DOMAIN (RESTRICT S DOMAIN-1)))** (a familiar phenomenon for those familiar with Knuth-Bendix completion). The lemma VARIABLE-LISTP-INTERSECTION has since been proved in "terms.events" to take care of that problem.

Now it looks like the rewrite using SUBST-APPEND-NOT-OCCUR-1 should succeed, since all hypotheses are relieved by rewriting alone. Just to make sure, we back up in PC-NQTHM and see if the BASH command (which calls the Boyer-Moore prover's simplifier) uses this rule on our original goal. Sure enough, it does.

Having successfully applied PC-NQTHM's REWRITE command and relieved the resulting hypothesis, we now have a conclusion that is the one displayed above, i.e.

```
(THEOREM (SUBST T
                (APPLY-TO-SUBST (NULLIFY-SUBST SG)
                                (CO-RESTRICT S DOMAIN-1))
                X))
```

Since (as we already know) **(NULLIFY-SUBST SG)** has the same domain as does SG, and since the hypotheses imply that **(DOMAIN SG)** is disjoint from the variables of **X**, the **SUBST** expression in this conclusion should simplify to:

```
(SUBST T (NULLIFY-SUBST SG)
       (SUBST T (CO-RESTRICT S DOMAIN-1)
              X))
```

We therefore need the lemma SUBST-APPLY-TO-SUBST-ELIMINATOR below (which is used under the substitution where **S** gets **(CO-RESTRICT S DOMAIN-1)** and **SG** gets **(NULLIFY-SUBST SG)**). [And so on......]

Appendix A
Events Files: sets, alists, terms, and generalize

THE FILE "sets.events"

```
;; Requires deftheory enhancement.
;; Requires only ground-zero theory, nqthm mode.

(setq events '(

;; Sets; Matt Kaufmann, Dec. 1989, revised March 1990.  The first few
;; events are some basic events about lists.  I'll take the approach
;; that all these basic functions will be disabled once enough
;; algebraic properties have been proved.  The first two lemmas,
;; LENGTH-CONS and LENGTH-NLISTP, reflect this decision.  I suspect
;; that it's a win in big proofs to keep basic functions disabled.

;; Theories:

;; (deftheory set-defns
;;  (length properp fix-properp member append subsetp delete
;;          disjoint intersection set-diff setp))

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

(prove-lemma length-nlistp (rewrite)
  (implies (nlistp x)
           (equal (length x) 0)))

(prove-lemma length-cons (rewrite)
  (equal (length (cons a x))
         (add1 (length x))))

(prove-lemma length-append (rewrite)
  (equal (length (append x y))
         (plus (length x) (length y))))

(disable length)

(prove-lemma append-assoc (rewrite)
  (equal (append (append x y) z)
         (append x (append y z))))

(prove-lemma member-cons (rewrite)
  (equal (member a (cons x l))
         (or (equal a x)
             (member a l))))

(prove-lemma member-nlistp (rewrite)
  (implies (nlistp l)
           (not (member a l))))

(disable member)

(defn subsetp (x y)
  (if (nlistp x)
      t
    (and (member (car x) y)
         (subsetp (cdr x) y))))

(defn subsetp-wit (x y)
```

```
  (if (nlistp x)
      t
      (if (member (car x) y)
          (subsetp-wit (cdr x) y)
        (car x))))

(prove-lemma subsetp-wit-witnesses (rewrite)
  ;; for occasional use in messy proofs; it and its lemma are kept disabled
  (equal (subsetp x y)
         (not (and (member (subsetp-wit x y) x)
                   (not (member (subsetp-wit x y) y))))))

(prove-lemma subsetp-wit-witnesses-general-1 (rewrite)
  (implies (and (not (member (subsetp-wit x y) x))
                (member a x))
           (member a y)))

(prove-lemma subsetp-wit-witnesses-general-2 (rewrite)
  (implies (and (member (subsetp-wit x y) y)
                (member a x))
           (member a y)))

(disable subsetp-wit-witnesses)
(disable subsetp-wit-witnesses-general-1)
(disable subsetp-wit-witnesses-general-2)

(prove-lemma subsetp-cons-1 (rewrite)
  (equal (subsetp (cons a x) y)
         (and (member a y) (subsetp x y))))

(prove-lemma subsetp-cons-2
             (rewrite)
             (implies (subsetp l m)
                      (subsetp l (cons a m))))

(prove-lemma subsetp-reflexivity
             (rewrite)
             (subsetp x x))

(prove-lemma cdr-subsetp
             (rewrite)
             (subsetp (cdr x) x))

(prove-lemma member-subsetp
             (rewrite)
             (implies (and (member x y) (subsetp y z))
                      (member x z)))

(prove-lemma subsetp-is-transitive
             (rewrite)
             (implies (and (subsetp x y) (subsetp y z))
                      (subsetp x z)))

(prove-lemma member-append (rewrite)
  (equal (member a (append x y))
         (or (member a x) (member a y))))

(prove-lemma subsetp-append (rewrite)
  (equal (subsetp (append x y) z)
         (and (subsetp x z) (subsetp y z))))

(prove-lemma subsetp-of-append-sufficiency (rewrite)
  (implies (or (subsetp a b) (subsetp a c))
           (subsetp a (append b c))))

(prove-lemma subsetp-nlistp (rewrite)
  (implies (nlistp x)
           (and (subsetp x y)
                (equal (subsetp y x)
```

```
                       (nlistp y)))))

(prove-lemma subsetp-cons-not-member (rewrite)
  (implies (not (member z x))
           (equal (subsetp x (cons z v))
                  (subsetp x v))))

(disable subsetp)

(defn properp (x)
  (if (listp x)
      (properp (cdr x))
    (equal x nil)))

(defn fix-properp (x)
  (if (listp x)
      (cons (car x)
            (fix-properp (cdr x)))
    nil))

(prove-lemma properp-fix-properp (rewrite)
  (properp (fix-properp x)))

(prove-lemma fix-properp-properp (rewrite)
  (implies (properp x)
           (equal (fix-properp x) x)))

(prove-lemma properp-cons (rewrite)
  (equal (properp (cons x y))
         (properp y)))

(prove-lemma properp-nlistp (rewrite)
  (implies (nlistp x)
           (equal (properp x)
                  (equal x nil))))

(prove-lemma fix-properp-cons (rewrite)
  (equal (fix-properp (cons x y))
         (cons x (fix-properp y))))

(prove-lemma fix-properp-nlistp (rewrite)
  (implies (nlistp x)
           (equal (fix-properp x)
                  nil)))

(prove-lemma properp-append (rewrite)
  (equal (properp (append x y))
         (properp y)))

(prove-lemma fix-properp-append (rewrite)
  (equal (fix-properp (append x y))
         (append x (fix-properp y))))

(prove-lemma append-nil (rewrite)
  (equal (append x nil)
         (fix-properp x)))

(defn delete (x l)
      (if (listp l)
          (if (equal x (car l))
              (cdr l)
              (cons (car l) (delete x (cdr l))))
          l))

(prove-lemma properp-delete (rewrite)
  (equal (properp (delete x l))
         (properp l)))

(defn disjoint (x y)
```

```
  (if (listp x)
      (and (not (member (car x) y))
           (disjoint (cdr x) y))
    t))

(defn disjoint-wit (x y)
  ;; for occasional use in messy proofs; it and the following lemma are kept disabled
  (if (listp x)
      (if (member (car x) y)
          (car x)
        (disjoint-wit (cdr x) y))
      t))

(prove-lemma disjoint-wit-witnesses (rewrite)
  (equal (disjoint x y)
         (not (and (member (disjoint-wit x y) x)
                   (member (disjoint-wit x y) y)))))

(disable disjoint-wit)

(disable disjoint-wit-witnesses)

(defn intersection (x y)
  (if (listp x)
      (if (member (car x) y)
          (cons (car x)
                (intersection (cdr x) y))
        (intersection (cdr x) y))
    nil))

(prove-lemma properp-intersection (rewrite)
  (properp (intersection x y)))

(defn set-diff (x y)
  (if (listp x)
      (if (member (car x) y)
          (set-diff (cdr x) y)
        (cons (car x) (set-diff (cdr x) y)))
    nil))

(prove-lemma properp-set-diff (rewrite)
  (properp (set-diff x y)))

(defn setp (x)
  (if (not (listp x))
      (equal x nil)
      (and (not (member (car x) (cdr x)))
           (setp (cdr x)))))

(prove-lemma setp-implies-properp (rewrite)
  (implies (setp x)
           (properp x)))

(disable properp)

(deftheory set-defns
  (length properp fix-properp member append subsetp delete
          disjoint intersection set-diff setp properp))

;; Set theory lemmas

(prove-lemma delete-cons (rewrite)
  (equal (delete a (cons b x))
         (if (equal a b)
             x
           (cons b (delete a x)))))

(prove-lemma delete-nlistp (rewrite)
  (implies (nlistp x)
```

```
           (equal (delete a x) x)))

(prove-lemma listp-delete (rewrite)
  (equal (listp (delete x l))
         (if (listp l)
             (or (not (equal x (car l)))
                 (listp (cdr l)))
             f)))

(prove-lemma delete-non-member (rewrite)
  (implies (not (member x y))
           (equal (delete x y) y)))

(prove-lemma delete-delete (rewrite)
  (equal (delete y (delete x z))
         (delete x (delete y z))))

(prove-lemma member-delete (rewrite)
  (implies (setp x)
           (equal (member a (delete b x))
                  (and (not (equal a b))
                       (member a x)))))

(prove-lemma setp-delete (rewrite)
  (implies (setp x)
           (setp (delete a x))))

(disable delete)

(prove-lemma disjoint-cons-1 (rewrite)
  (equal (disjoint (cons a x) y)
         (and (not (member a y))
              (disjoint x y))))

(prove-lemma disjoint-cons-2 (rewrite)
  (equal (disjoint x (cons a y))
         (and (not (member a x))
              (disjoint x y))))

(prove-lemma disjoint-nlistp (rewrite)
  (implies (or (nlistp x) (nlistp y))
           (disjoint x y)))

(prove-lemma disjoint-symmetry (rewrite)
  (equal (disjoint x y)
         (disjoint y x)))

(prove-lemma disjoint-append-right (rewrite)
  (equal (disjoint u (append y z))
         (and (disjoint u y)
              (disjoint u z))))

(prove-lemma disjoint-append-left (rewrite)
  (equal (disjoint (append y z) u)
         (and (disjoint y u)
              (disjoint z u))))

(prove-lemma disjoint-non-member (rewrite)
  (implies (and (member a x)
                (member a y))
           (not (disjoint x y))))

(prove-lemma disjoint-subsetp-monotone-second (rewrite)
  (implies (and (subsetp y z)
                (disjoint x z))
           (disjoint x y)))

(prove-lemma subsetp-disjoint-2 (rewrite)
  (implies (and (subsetp x y)
```

```
                (disjoint y z))
           (disjoint z x)))

(prove-lemma subsetp-disjoint-1 (rewrite)
  (implies (and (subsetp x y)
                (disjoint y z))
           (disjoint x z))
  ((use (disjoint-symmetry (x x) (y z)))
   (disable disjoint-symmetry)))

(prove-lemma subsetp-disjoint-3 (rewrite)
  (implies (and (subsetp x y)
                (disjoint z y))
           (disjoint x z)))

(disable disjoint)

(prove-lemma intersection-disjoint (rewrite)
  (equal (equal (intersection x y) nil)
         (disjoint x y)))

(prove-lemma intersection-nlistp (rewrite)
  (implies (or (nlistp x) (nlistp y))
           (equal (intersection x y) nil)))

(prove-lemma member-intersection (rewrite)
  (equal (member a (intersection x y))
         (and (member a x) (member a y))))

(prove-lemma subsetp-intersection (rewrite)
  (equal (subsetp x (intersection y z))
         (and (subsetp x y) (subsetp x z)))
  ((induct (subsetp x y))))

(prove-lemma intersection-symmetry (rewrite)
  (subsetp (intersection x y)
           (intersection y x)))

(prove-lemma intersection-cons-1 (rewrite)
  (equal (intersection (cons a x) y)
         (if (member a y)
             (cons a (intersection x y))
           (intersection x y))))

(prove-lemma intersection-cons-2 (rewrite)
  (implies (not (member a y))
           (equal (intersection y (cons a x))
                  (intersection y x))))

;; The following is needed because DISJOINT-INTERSECTION-COMMUTER,
;; added during polishing, caused the proof of
;; DISJOINT-DOMAIN-CO-RESTRICT (in "alists.events") to fail.

(prove-lemma intersection-cons-3 (rewrite)
  (implies (member w x)
           (equal (subsetp (intersection y (cons w z))
                           x)
                  (subsetp (intersection y z)
                           x)))
  ((enable intersection)))

(prove-lemma intersection-cons-subsetp (rewrite)
  (subsetp (intersection x y)
           (intersection x (cons a y))))

(prove-lemma subsetp-intersection-left-1 (rewrite)
  (subsetp (intersection x y) x)
  ((enable intersection)))
```

```
(prove-lemma subsetp-intersection-left-2 (rewrite)
  (subsetp (intersection x y) y)
  ((enable intersection)))

(prove-lemma subsetp-intersection-sufficiency-1 (rewrite)
  (implies (subsetp y z)
           (subsetp (intersection x y) z))
  ((enable intersection)))

(prove-lemma subsetp-intersection-sufficiency-2 (rewrite)
  (implies (subsetp y z)
           (subsetp (intersection y x) z))
  ((enable intersection)))

(prove-lemma intersection-associative (rewrite)
  (equal (intersection (intersection x y) z)
         (intersection x (intersection y z)))
  ((enable intersection)))

(prove-lemma intersection-elimination (rewrite)
  (implies (subsetp x y)
           (equal (intersection x y)
                  (fix-properp x))))

(prove-lemma length-intersection (rewrite)
  (not (lessp (length x)
              (length (intersection x y)))))

(prove-lemma subsetp-intersection-member (rewrite)
  (implies (and (subsetp (intersection x y) z)
                (not (member a z)))
           (and (implies (member a x)
                         (not (member a y)))
                (implies (member a y)
                         (not (member a x))))))

;; The following wasn't needed in the proof about generalization, but is a nice rule.
(prove-lemma intersection-append (rewrite)
  (equal (intersection (append x y) z)
         (append (intersection x z) (intersection y z))))

;; I'd rather just prove that intersection distributes over append on
;; the right but that isn't true.  Congruence relations would probably
;; help a lot with that problem.  In the meantime, I content myself
;; with the following.
(prove-lemma disjoint-intersection-append (rewrite)
  (equal (disjoint x (intersection y (append z1 z2)))
         (and (disjoint x (intersection y z1))
              (disjoint x (intersection y z2))))
  ((enable intersection)))

;; See comment just above DISJOINT-INTERSECTION-APPEND
(prove-lemma subsetp-intersection-append (rewrite)
  (equal (subsetp (intersection u (append x y))
                  z)
         (and (subsetp (intersection u x) z)
              (subsetp (intersection u y) z))))

(prove-lemma subsetp-intersection-elimination-lemma (rewrite)
  (implies (and (subsetp y x)
                (not (subsetp y z)))
           (not (subsetp (intersection x y) z)))
  ((use (subsetp-is-transitive (x y) (y (intersection x y)) (z z)))
   (disable intersection)))

(prove-lemma subsetp-intersection-elimination (rewrite)
  ;; Interestingly, the prover failed to prove this when I used EQUAL.
  ;; Apparently the IFF causes a necessary case split.
  (implies (subsetp y x)
```

```
            (iff (subsetp (intersection x y) z)
                 (subsetp y z)))
  ((disable intersection)))

(prove-lemma disjoint-intersection (rewrite)
  (equal (disjoint (intersection x y) z)
         (disjoint x (intersection y z)))
  ((enable intersection)
   (disable disjoint)))

(prove-lemma subsetp-intersection-monotone-1 (rewrite)
  (implies (and (subsetp (intersection x y) z)
                (subsetp x1 x))
           (subsetp (intersection x1 y)
                    z))
  ((enable disjoint subsetp)))

;; The lemma SUBSETP-INTERSECTION-MONOTONE-2 below was added during
;; polishing of the final proof in "generalize.events", since the
;; lemma immediately above wasn't enough at that point.  Actually
;; I realized at this point that intersection commutes from the point
;; of view of subsetp:

(prove-lemma subsetp-intersection-commuter (rewrite)
  (equal (subsetp (intersection x y) z)
         (subsetp (intersection y x) z))
  ((use (subsetp-wit-witnesses (x (intersection y x)) (y z))
        (subsetp-wit-witnesses (x (intersection x y)) (y z)))))

(prove-lemma subsetp-intersection-monotone-2 (rewrite)
  (implies (and (subsetp (intersection y x) z)
                (subsetp x1 x))
           (subsetp (intersection x1 y)
                    z)))

(prove-lemma disjoint-intersection-commuter (rewrite)
  (equal (disjoint x (intersection y z))
         (disjoint x (intersection z y)))
  ((use (disjoint-wit-witnesses (x x) (y (intersection y z)))
        (disjoint-wit-witnesses (x x) (y (intersection z y))))
   (disable intersection)))

(prove-lemma disjoint-intersection3 (rewrite)
  (implies (disjoint free
                     (intersection vars x))
           (equal (intersection x (intersection vars free))
                  nil))
  ((use (disjoint-wit-witnesses
         (x x)
         (y (intersection vars free))))))

(disable intersection)

(prove-lemma member-set-diff (rewrite)
  (equal (member a (set-diff y z))
         (and (member a y)
              (not (member a z)))))

(prove-lemma subsetp-set-diff-1 (rewrite)
  (subsetp (set-diff x y) x))

(prove-lemma disjointp-set-diff (rewrite)
  (disjoint (set-diff x y) y))

(prove-lemma subsetp-set-diff-2 (rewrite)
  (equal (subsetp x (set-diff y z))
         (and (subsetp x y)
              (disjoint x z)))
  ((enable-theory set-defns)))
```

```
(prove-lemma set-diff-cons (rewrite)
  (equal (set-diff (cons a x) y)
         (if (member a y)
             (set-diff x y)
           (cons a (set-diff x y)))))

(prove-lemma set-diff-nlistp (rewrite)
  (implies (nlistp x)
           (equal (set-diff x y) nil)))

;; The following was discovered during final polishing, for the
;; proof of MAIN-HYPS-RELIEVED-6-FIRST.
(prove-lemma disjoint-set-diff-general (rewrite)
  (equal (disjoint x (set-diff y z))
         (subsetp (intersection x y) z))
  ((induct (intersection x y))))

(prove-lemma intersection-subsetp-identity (rewrite)
  (implies (and (properp x)
                (subsetp x y))
           (equal (intersection x y) x))
  ((enable subsetp)))

(prove-lemma intersection-x-x (rewrite)
  (implies (properp x)
           (equal (intersection x x) x)))

(prove-lemma subsetp-set-diff-mononone-2 (rewrite)
  (subsetp (set-diff x (append y z))
           (set-diff x z))
  ((disable set-diff)))

(prove-lemma subsetp-set-diff-monotone-second (rewrite)
  (equal (subsetp (set-diff x y) (set-diff x z))
         (subsetp (intersection x z) y))
  ((enable intersection)))

(prove-lemma set-diff-nil (rewrite)
  (equal (set-diff x nil)
         (fix-properp x)))

(prove-lemma set-diff-cons-non-member-1 (rewrite)
  (implies (not (member a x))
           (equal (set-diff x (cons a y))
                  (set-diff x y))))

(prove-lemma length-intersection-set-diff ()
  (equal (length x)
         (plus (length (set-diff x y))
               (length (intersection x y))))
  ((enable set-diff intersection length)))

(prove-lemma length-set-diff-opener (rewrite)
  (equal (length (set-diff x y))
         (difference (length x)
                     (length (intersection x y))))
  ((use (length-intersection-set-diff))))

(prove-lemma listp-set-diff (rewrite)
  (equal (listp (set-diff x y))
         (not (subsetp x y)))
  ((enable set-diff)))

;; Here is a messy lemma about disjoint and such
(prove-lemma disjoint-intersection-set-diff-intersection (rewrite)
  (disjoint x (intersection y (set-diff z (intersection y x))))
  ((enable disjoint-wit-witnesses)
   (disable set-diff)))
```

```
(disable set-diff)

(prove-lemma member-fix-properp (rewrite)
  (equal (member a (fix-properp x))
         (member a x)))

(prove-lemma setp-append (rewrite)
  (equal (setp (append x y))
         (and (disjoint x y)
              (setp (fix-properp x))
              (setp y))))

(prove-lemma setp-cons (rewrite)
  (equal (setp (cons a x))
         (and (not (member a x))
              (setp x))))

(prove-lemma setp-nlistp (rewrite)
  (implies (nlistp x)
           (equal (setp x)
                  (equal x nil))))

(defn make-set
      (l)
      (if (not (listp l))
          nil
          (if (member (car l) (cdr l))
              (make-set (cdr l))
              (cons (car l) (make-set (cdr l))))))

(prove-lemma make-set-preserves-member
             (rewrite)
             (equal (member x (make-set l))
                    (member x l)))

(prove-lemma make-set-preserves-subsetp-1
             (rewrite)
             (equal (subsetp (make-set x) (make-set y))
                    (subsetp x y)))

(prove-lemma make-set-preserves-subsetp-2
             (rewrite)
             (equal (subsetp x (make-set y))
                    (subsetp x y))
             ((enable subsetp)))

(prove-lemma make-set-preserves-subsetp-3
             (rewrite)
             (equal (subsetp (make-set x) y)
                    (subsetp x y)))

(prove-lemma make-set-gives-setp
             (rewrite)
             (setp (make-set x)))

(prove-lemma make-set-set-diff (rewrite)
  (equal (make-set (set-diff x y))
         (set-diff (make-set x) (make-set y))))

(prove-lemma set-diff-make-set (rewrite)
  (equal (set-diff x (make-set y))
         (set-diff x y))
  ((enable set-diff)))

(prove-lemma listp-make-set (rewrite)
  (equal (listp (make-set x))
         (listp x)))

(disable setp)
```

```
;;;;;; The following were proved in the course of the final run
;;;;;; through the generalization proof.  There are a couple or
;;;;;; so noted above here, too.

(prove-lemma set-diff-append (rewrite)
  (equal (set-diff x (append y z))
         (set-diff (set-diff x z) y))
  ((induct (set-diff x z))))

(prove-lemma length-set-diff-leq (rewrite)
  (not (lessp (length x)
              (length (set-diff x y)))))

(prove-lemma lessp-length (rewrite)
  (implies (listp x)
           (lessp 0 (length x)))
  ((enable length)))

(prove-lemma listp-intersection (rewrite)
  (equal (listp (intersection x y))
         (not (disjoint x y)))
  ((enable intersection)))

(prove-lemma length-set-diff-lessp (rewrite)
  (implies (not (disjoint x new))
           (lessp (length (set-diff x new))
                  (length x))))

(prove-lemma disjoint-implies-empty-intersection (rewrite)
  (implies (disjoint x y)
           (equal (intersection x y) nil)))

;; The following lemma DISJOINT-INTERSECTION3-MIDDLE is needed for the
;; proof of ALL-VARS-DISJOINT-OR-SUBSETP-GEN-CLOSURE in
;; generalize.events.  I think I could avoid lemmas like this one if
;; INTERSECTION were actually commutative-associative (in which case
;; I'd get rid of disjoint and rely on normalization).
;; Maybe I should redo the notion of disjoint sometime, perhaps using
;; the fact that intersection is commutative and associative when it's
;; equated with nil.

(prove-lemma disjoint-intersection3-middle (rewrite)
  (implies (disjoint y (intersection x z))
           (equal (intersection x (intersection y z))
                  nil))
  ((use (disjoint-wit-witnesses
         (x x) (y (intersection y z))))))

(prove-lemma disjoint-subsetp-hack (rewrite)
  (implies (and (disjoint x
                          (intersection u v))
                (subsetp w x))
           (disjoint u
                     (intersection w v)))
  ((use (disjoint-wit-witnesses
         (x u)
         (y (intersection w v)))
        (disjoint-non-member
         (a (disjoint-wit u (intersection w v)))
         (x x)
         (y (intersection u v)))
        (member-subsetp
         (x (disjoint-wit u
                          (intersection w v)))
         (y w)
         (z x)))
   (disable disjoint-non-member member-subsetp)))

(prove-lemma subsetp-set-diff-sufficiency (rewrite)
```

```
  (implies (subsetp x y)
           (subsetp (set-diff x z) y))
  ((enable set-diff)))

;; The following lemma SETP-INTERSECTION-SUFFICIENCY is needed for
;; MAPPING-RESTRICT from "alists.events", because (I believe)
;; DOMAIN-RESTRICT, which was added during polishing, changed the
;; course of the previous proof.  Similarly for
;; SETP-SET-DIFF-SUFFICIENCY and the lemma MAPPING-CO-RESTRICT.

(prove-lemma setp-intersection-sufficiency (rewrite)
  (implies (setp x)
           (setp (intersection x y)))
  ((enable intersection)))

(prove-lemma setp-set-diff-sufficiency (rewrite)
  (implies (setp x)
           (setp (set-diff x y)))
  ((enable set-diff)))

;; The definition of FIX-PROPERP was also added in polishing because
;; of a problem with the proof of GEN-CLOSURE-ACCEPT in
;; "generalize.events".  Here are a couple of lemmas about it that
;; might or might not be useful; all other lemmas about it above, and
;; the definition, were added during polishing.

(disable fix-properp)

(prove-lemma subsetp-fix-properp-1 (rewrite)
  (equal (subsetp (fix-properp x) y)
         (subsetp x y))
  ((enable subsetp)))

(prove-lemma subsetp-fix-properp-2 (rewrite)
  (equal (subsetp x (fix-properp y))
         (subsetp x y))
  ((enable subsetp)))

))
```

THE FILE "alists.events"

```
;; Requires deftheory enhancement.
;; Requires sets.

(setq events '(

;; Alists, March 1990.  Most of the definitions and some of the lemmas
;; were contributed by Bill Bevier; the rest are by Matt Kaufmann.

;; Functions defined here:

;; (deftheory alist-defns
;;   (alistp domain range value bind rembind invert mapping
;;           restrict co-restrict))

(defn alistp (x)
  (if (listp x)
      (and (listp (car x))
           (alistp (cdr x)))
    (equal x nil)))

(prove-lemma alistp-implies-properp (rewrite)
  (implies (alistp x)
           (properp x)))

(prove-lemma alistp-nlistp (rewrite)
  (implies (nlistp x)
           (equal (alistp x)
                  (equal x nil))))

(prove-lemma alistp-cons (rewrite)
  (equal (alistp (cons a x))
         (and (listp a)
              (alistp x))))

(disable alistp)

(prove-lemma alistp-append (rewrite)
  (equal (alistp (append x y))
         (and (alistp (fix-properp x)) (alistp y))))

(defn domain (map)
  (if (listp map)
      (if (listp (car map))
          (cons (car (car map)) (domain (cdr map)))
        (domain (cdr map)))
    nil))

(prove-lemma properp-domain (rewrite)
  (properp (domain map)))

(prove-lemma domain-append (rewrite)
  (equal (domain (append x y))
         (append (domain x) (domain y))))

(prove-lemma domain-nlistp (rewrite)
  (implies (nlistp map)
           (equal (domain map) nil)))

(prove-lemma domain-cons (rewrite)
  (equal (domain (cons a map))
         (if (listp a)
             (cons (car a) (domain map))
           (domain map))))

(prove-lemma member-domain-sufficiency (rewrite)
  (implies (member (cons a x) y)
```

```
           (member a (domain y))))

(prove-lemma subsetp-domain (rewrite)
  (implies (subsetp x y)
           (subsetp (domain x) (domain y))))

(disable domain)

(defn range (map)
  (if (listp map)
      (if (listp (car map))
          (cons (cdr (car map)) (range (cdr map)))
        (range (cdr map)))
    nil))

(prove-lemma properp-range (rewrite)
  (properp (range map)))

(prove-lemma range-append (rewrite)
  (equal (range (append s1 s2))
         (append (range s1) (range s2))))

(prove-lemma range-nlistp (rewrite)
  (implies (nlistp map)
           (equal (range map) nil)))

(prove-lemma range-cons (rewrite)
  (equal (range (cons a map))
         (if (listp a)
             (cons (cdr a) (range map))
           (range map))))

(disable range)

;; BOUNDP has been eliminated in favor of membership in domain.
;; Notice that I have to talk about things like disjointness of
;; domains anyhow.  New definition body would be (member x (domain map)).

;(defn boundp (x map)
;  (if (listp map)
;      (if (listp (car map))
;          (if (equal x (caar map))
;              t
;            (boundp x (cdr map)))
;        (boundp x (cdr map)))
;    f))

(defn value (x map)
  (if (listp map)
      (if (and (listp (car map))
               (equal x (caar map)))
          (cdar map)
        (value x (cdr map)))
    0))

(prove-lemma value-nlistp (rewrite)
  (implies (nlistp map)
           (equal (value x map) 0)))

(prove-lemma value-cons (rewrite)
  (equal (value x (cons pair map))
         (if (and (listp pair)
                  (equal x (car pair)))
             (cdr pair)
           (value x map))))

(disable value)

(defn bind (x v map)
```

```
  (if (listp map)
      (if (listp (car map))
          (if (equal x (caar map))
              (cons (cons x v) (cdr map))
            (cons (car map) (bind x v (cdr map))))
        (cons (car map) (bind x v (cdr map))))
    (cons (cons x v) nil)))

(defn rembind (x map)
  (if (listp map)
      (if (listp (car map))
          (if (equal x (caar map))
              (cdr map)
            (cons (car map) (rembind x (cdr map))))
        (cons (car map) (rembind x (cdr map))))
    nil))

(defn invert (map)
  (if (listp map)
      (if (listp (car map))
          (cons (cons (cdr (car map))
                      (car (car map)))
                (invert (cdr map)))
        (invert (cdr map)))
    nil))

(prove-lemma properp-invert (rewrite)
  (properp (invert map)))

(prove-lemma invert-nlistp (rewrite)
  (implies (nlistp map)
           (equal (invert map) nil)))

(prove-lemma invert-cons (rewrite)
  (equal (invert (cons pair map))
         (if (listp pair)
             (cons (cons (cdr pair) (car pair))
                   (invert map))
           (invert map))))

(prove-lemma value-invert-not-member-of-domain (rewrite)
  (implies (and (member g (range sg))
                (disjoint (domain s) (domain sg)))
           (not (member (value g (invert sg)) (domain s)))))

(disable invert)

(defn mapping (map)
  ;; an alist with no duplicate keys
  (and (alistp map)
       (setp (domain map))))

;; For when we disable mapping:
(prove-lemma mapping-implies-alistp (rewrite)
  (implies (mapping map)
           (alistp map)))

(prove-lemma mapping-implies-setp-domain (rewrite)
  (implies (mapping map)
           (setp (domain map))))

(defn restrict (s new-domain)
  (if (listp s)
      (if (and (listp (car s))
               (member (caar s) new-domain))
          (cons (car s)
                (restrict (cdr s) new-domain))
        (restrict (cdr s) new-domain))
    nil))
```

```
(defn co-restrict (s new-domain)
  (if (listp s)
      (if (and (listp (car s))
               (not (member (caar s) new-domain)))
          (cons (car s)
                (co-restrict (cdr s) new-domain))
        (co-restrict (cdr s) new-domain))
    nil))

(deftheory alist-defns
  (alistp domain range value bind rembind invert mapping
          restrict co-restrict))

;;;;; alist lemmas

; DOMAIN

;; The following was proved in the course of the final run through
;; the generalization proof.  Actually now I see that
;; some other lemmas are now obsolete, so I'll put these both
;; early in the file and delete the others.

(prove-lemma domain-restrict (rewrite)
  (equal (domain (restrict s dom))
         (intersection (domain s) dom))
  ((enable restrict)))

(prove-lemma domain-co-restrict (rewrite)
  (equal (domain (co-restrict s dom))
         (set-diff (domain s) dom))
  ((enable co-restrict)))

(prove-lemma domain-bind (rewrite)
  (equal (domain (bind x v map))
         (if (member x (domain map))
             (domain map)
           (append (domain map) (list x)))))

(prove-lemma domain-rembind (rewrite)
  (equal (domain (rembind x map))
         (delete x (domain map))))

(prove-lemma domain-invert (rewrite)
  (equal (domain (invert map))
         (range map))
  ((enable-theory alist-defns)))

; RANGE

(prove-lemma range-invert (rewrite)
  (equal (range (invert map))
         (domain map))
  ((enable-theory alist-defns)))

; BOUNDP

(prove-lemma boundp-bind (rewrite)
  (equal (member x (domain (bind y v map)))
         (or (equal x y)
             (member x (domain map)))))

(prove-lemma boundp-rembind (rewrite)
  (implies (mapping map)
           (equal (member x (domain (rembind y map)))
                  (if (equal x y)
                      f
                    (member x (domain map))))))

(prove-lemma boundp-subsetp ()
```

```
       (implies (and (subsetp map1 map2)
                     (member name (domain map1)))
                (member name (domain map2))))

(prove-lemma disjoint-domain-singleton (rewrite)
  (and (equal (disjoint (domain s) (list x))
              (not (member x (domain s))))
       (equal (disjoint (list x) (domain s))
              (not (member x (domain s))))))

(prove-lemma boundp-value-invert (rewrite)
  (implies (member x (range map))
           (member (value x (invert map)) (domain map)))
  ((induct (domain map))))

; VALUE

(prove-lemma value-when-not-bound (rewrite)
  (implies (not (member name (domain map)))
           (equal (value name map)
                  0))
  ((induct (domain map))))

(prove-lemma value-bind (rewrite)
  (equal (value x (bind y v map))
         (if (equal x y)
             v
           (value x map))))

(prove-lemma value-rembind (rewrite)
  (implies (mapping map)
           (equal (value x (rembind y map))
                  (if (equal x y)
                      0
                    (value x map)))))

(prove-lemma value-append (rewrite)
  (equal (value x (append s1 s2))
         (if (member x (domain s1))
             (value x s1)
           (value x s2))))

(prove-lemma value-value-invert (rewrite)
  (implies (and (member x (range s))
                (mapping s))
           (equal (value (value x (invert s))
                         s)
                  x))
  ((enable-theory alist-defns)))

; MAPPING

(prove-lemma mapping-append (rewrite)
  (equal (mapping (append s1 s2))
         (and (disjoint (domain s1) (domain s2))
              (mapping (fix-properp s1))
              (mapping s2))))

(disable mapping)

;; RESTRICT and CO-RESTRICT

(prove-lemma alistp-restrict (rewrite)
  (alistp (restrict s r)))

(prove-lemma alistp-co-restrict (rewrite)
  (alistp (co-restrict s r)))

(prove-lemma value-restrict (rewrite)
```

```
  (implies (and (member a r)
                (member a (domain s)))
           (equal (value a (restrict s r))
                  (value a s))))

(prove-lemma value-co-restrict (rewrite)
  (implies (and (not (member a r))
                (member a (domain s)))
           (equal (value a (co-restrict s r))
                  (value a s))))

(prove-lemma mapping-restrict (rewrite)
  (implies (mapping s)
           (mapping (restrict s x)))
  ((enable mapping)))

(prove-lemma mapping-co-restrict (rewrite)
  (implies (mapping s)
           (mapping (co-restrict s x)))
  ((enable mapping)))

(disable restrict)
(disable co-restrict)

))
```

THE FILE "terms.events"

```
;; Requires deftheory, defn-sk, and constrain enhancments.
;; Requires sets and alists libraries.

(setq events '(

;; This is a library of events about terms, including substitutions.
;; A TERMP is either a variable or the application of a function
;; symbol to a "proper list" of terms.  Variables and function symbols
;; are introduced with CONSTRAIN.

;; NOTE:  In functions like TERMP that have a flag, it seems to be
;; important to use T and F rather than, say, T and 'LIST.  That's
;; because otherwise, the "worse-than" heuristic will otherwise
;; prevent some necessary backchaining in cases where the hypothesis
;; to be relieved is of the form (TERMP 'LIST ...) and an "ancestor"
;; is of the form (TERMP T ...).

;; Definitions:

;; (deftheory term-defns
;;  (variablep-intro variable-listp termp function-symbol-intro all-vars))

;; (deftheory substitution-defns
;;   (instance var-substp compose apply-to-subst subst
;;    nullify-subst ;; returns a substitution whose range has no variables
;;    ))

(constrain variablep-intro (rewrite)
           (and (implies (listp x)
                         (not (variablep x)))
                (or (truep (variablep x))
                    (falsep (variablep x))))
           ((variablep nlistp)))

(defn variable-listp (x)
  (if (listp x)
      (and (variablep (car x))
           (variable-listp (cdr x)))
    (equal x nil)))

(prove-lemma variable-listp-implies-properp (rewrite)
  (implies (variable-listp x)
           (properp x)))

(prove-lemma variable-listp-cons (rewrite)
  (equal (variable-listp (cons a x))
         (and (variablep a)
              (variable-listp x))))

(prove-lemma variable-nlistp (rewrite)
  (implies (nlistp x)
           (equal (variable-listp x)
                  (equal x nil))))

(disable variable-listp)

(constrain function-symbol-intro (rewrite)
           ;; We designate ZERO as a function symbol
           (function-symbol-p (fn))
           ((function-symbol-p litatom)
            (fn (lambda () 'zero))))

(defn termp (flg x)
  (if flg
      (if (variablep x)
          t
```

```
        (if (listp x)
            (and (function-symbol-p (car x))
                 (termp f (cdr x)))
          f))
    (if (listp x)
        (and (termp t (car x))
             (termp f (cdr x)))
      (equal x nil))))

(prove-lemma termp-list-cons (rewrite)
  (equal (termp f (cons a x))
         (and (termp t a)
              (termp f x))))

(prove-lemma termp-list-nlistp (rewrite)
  (implies (nlistp x)
           (equal (termp f x)
                  (equal x nil))))

(prove-lemma termp-t-cons (rewrite)
  (implies flg
           (equal (termp flg (cons a x))
                  (and (function-symbol-p a)
                       (termp f x)))))

(prove-lemma termp-t-nlistp (rewrite)
  (implies (and flg
                (not (listp x)))
           (equal (termp flg x)
                  (variablep x))))

(disable termp)

(prove-lemma termp-list-implies-properp (rewrite)
  (implies (termp f x)
           (properp x))
  ((induct (properp x))))

(defn all-vars (flg x)
  ;; duplicates are ok
  (if flg
      (if (variablep x)
          (list x)
        (if (listp x)
            (all-vars f (cdr x))
          nil))
    (if (listp x)
        (append (all-vars t (car x))
                (all-vars f (cdr x)))
      nil)))

(prove-lemma properp-all-vars (rewrite)
  (properp (all-vars flg x)))

(prove-lemma all-vars-list-cons (rewrite)
  (equal (all-vars f (cons a x))
         (append (all-vars t a)
                 (all-vars f x)))
  ((enable all-vars)))

(prove-lemma all-vars-t-cons (rewrite)
  (implies flg
           (equal (all-vars flg (cons a x))
                  (all-vars f x))))

;; Here is a hack to deal with the flags.

(prove-lemma all-vars-subsetp-append-hack (rewrite)
  (implies (and flg1 flg2)
```

```
           (and (subsetp (all-vars flg1 x)
                         (append (all-vars flg2 x) y))
                (subsetp (all-vars flg1 x)
                         (append y (all-vars flg2 x))))))

;; The following is used later in the proof of MEMBER-PRESERVES-DISJOINT-ALL-VARS
;; and could conceivably be of use elsewhere.
(prove-lemma all-vars-flg-boolean nil
             (implies flg
                      (equal (all-vars flg x)
                             (all-vars t x)))
             ((enable-theory t)))

(disable all-vars)

(deftheory term-defns
  (variablep-intro variable-listp termp function-symbol-intro all-vars))

;;;;; lemmas about termps

(prove-lemma variable-listp-set-diff (rewrite)
  (implies (variable-listp x)
           (variable-listp (set-diff x y)))
  ((enable-theory term-defns)))

(prove-lemma all-vars-variablep (rewrite)
  (implies (and flg (variablep x))
           (equal (all-vars flg x) (list x)))
  ((enable-theory t)))

(prove-lemma member-variable-listp-implies-variablep (rewrite)
  (implies (and (member a x) (variable-listp x))
           (variablep a))
  ((enable-theory t)))

;; The following was proved in the course of the final run through the
;; generalization proof.

(prove-lemma variable-listp-intersection (rewrite)
  (implies (or (variable-listp x) (variable-listp y))
           (variable-listp (intersection x y)))
  ((enable intersection)))

(prove-lemma termp-range-restrict (rewrite)
  (implies (termp f (range s))
           (termp f (range (restrict s x))))
  ((enable restrict)))

(prove-lemma termp-range-co-restrict (rewrite)
  (implies (termp f (range s))
           (termp f (range (co-restrict s x))))
  ((enable co-restrict)))

(prove-lemma member-preserves-disjoint-all-vars-lemma nil
  (implies (and (disjoint y (all-vars f x))
                (member g x))
           (disjoint y (all-vars t g)))
  ((induct (member g x))))

(prove-lemma member-preserves-disjoint-all-vars (rewrite)
  (implies (and flg
                (disjoint y (all-vars f x))
                (member g x))
           (disjoint y (all-vars flg g)))
  ((use (member-preserves-disjoint-all-vars-lemma)
        (all-vars-flg-boolean (x g)))))

(prove-lemma member-all-vars-subsetp (rewrite)
  (implies (and flg
```

```
                (member a x))
           (subsetp (all-vars flg a)
                    (all-vars f x)))
  ((enable member)))

(prove-lemma all-vars-f-monotone (rewrite)
  (implies (subsetp x y)
           (subsetp (all-vars f x) (all-vars f y)))
  ((enable subsetp all-vars)))

;;;;; substitutions:  definitions

(defn var-substp (s)
  (and (mapping s)
       (variable-listp (domain s))
       (termp f (range s))))

(defn subst (flg s x)
  ;; works for other than var-substp's
  (if flg
      (if (member x (domain s))
          (value x s)
        (if (variablep x)
            x
          (if (listp x)
              (cons (car x)
                    (subst f s (cdr x)))
            ;; impossible value of f for non-termp
            f)))
    (if (listp x)
        (cons (subst t s (car x))
              (subst f s (cdr x)))
      nil)))

(defn apply-to-subst (s1 s2)
  ;; apply s1 to each term in range of s2
  (if (listp s2)
      (if (listp (car s2))
          (cons (cons (caar s2) (subst t s1 (cdar s2)))
                (apply-to-subst s1 (cdr s2)))
        (apply-to-subst s1 (cdr s2)))
    nil))

(defn compose (s1 s2)
  ;; represents the result of applying s1 and then s2
  (append (apply-to-subst s2 s1)
          s2))

;; Later we may wish to prove correctness of one-way-unify
(defn-sk instance (flg term1 term2)
  ;; term1 is an instance of term2
  (exists one-way-unifier
          (and (var-substp one-way-unifier)
               (equal term1 (subst flg one-way-unifier term2)))))

;;;;; substitution lemmas

(prove-lemma subst-list-cons (rewrite)
  (equal (subst f s (cons a x))
         (cons (subst t s a)
               (subst f s x))))

(prove-lemma subst-list-nlistp (rewrite)
  (implies (nlistp x)
           (equal (subst f s x) nil)))

(prove-lemma subst-t-variablep (rewrite)
  (implies (and flg
                (variablep x))
```

```
          (equal (subst flg s x)
                 (if (member x (domain s))
                     (value x s)
                   x))))

(prove-lemma subst-t-non-variablep (rewrite)
  (implies flg
           (equal (subst flg s (cons fn x))
                  (if (member (cons fn x) (domain s))
                      (value (cons fn x) s)
                    (cons fn (subst f s x)))))
  ((enable subst)))

(prove-lemma all-vars-subst-lemma (rewrite)
  (implies (and flg
                (member x (domain s)))
           (subsetp (all-vars flg (value x s))
                    (all-vars f (range s))))
  ;; hint needed for induction
  ((enable range)))

(prove-lemma all-vars-subst (rewrite)
  (implies (termp flg x)
           (subsetp (all-vars flg (subst flg s x))
                    (append (all-vars flg x)
                            (all-vars f (range s)))))
  ((enable termp)))

(prove-lemma subst-occur (rewrite)
  (implies (and flg
                (member x (domain s)))
           (equal (subst flg s x)
                  (value x s))))

(prove-lemma boundp-in-var-substp-implies-variablep (rewrite)
  (implies (and (variable-listp (domain s))
                (not (variablep a)))
           (not (member a (domain s))))
  ((induct (domain s))))

(prove-lemma variablep-value-invert (rewrite)
  (implies (and (variable-listp (domain s))
                (member x (range s)))
           (variablep (value x (invert s))))
  ((induct (range s))))

(prove-lemma subst-invert (rewrite)
  (implies (and (termp flg x)
                (disjoint (domain s) (all-vars flg x))
                (var-substp s))
           (equal (subst flg s (subst flg (invert s) x))
                  x))
  ((enable termp)))

(prove-lemma domain-apply-to-subst (rewrite)
  (equal (domain (apply-to-subst s1 s2))
         (domain s2)))

(prove-lemma alistp-apply-to-subst (rewrite)
  (alistp (apply-to-subst s1 s2)))

(prove-lemma mapping-apply-to-subst (rewrite)
  (implies (mapping s)
           (mapping (apply-to-subst s1 s)))
  ((enable mapping)))

;; Lemmas like the following shouldn't be necessary if congruence
;; relations (as suggested by Bishop Brock) are implemented.
```

```
(prove-lemma subst-flg-not-list (rewrite)
  (implies flg
           (and (equal (equal (subst flg s x)
                              (subst t s x))
                       t)
                (equal (equal (subst t s x)
                              (subst flg s x))
                       t))))

(prove-lemma subst-co-restrict (rewrite)
  (implies (and (disjoint x
                          (intersection (domain s) (all-vars flg term)))
                (variable-listp (domain s))
                (termp flg term))
           (equal (subst flg (co-restrict s x) term)
                  (subst flg s term))))

(prove-lemma subst-restrict (rewrite)
  (implies (and (subsetp (intersection (domain s) (all-vars flg term))
                         x)
                (variable-listp (domain s))
                (termp flg term))
           (equal (subst flg (restrict s x) term)
                  (subst flg s term))))

(prove-lemma termp-value (rewrite)
  (implies (and flg
                (member x (domain s))
                (termp f (range s)))
           (termp flg (value x s)))
  ((enable termp)
   (induct (value x s))))

(prove-lemma termp-subst (rewrite)
  (implies (and (termp flg x)
                (termp f (range s)))
           (termp flg (subst flg s x)))
  ((enable termp)))

(prove-lemma termp-domain (rewrite)
  (implies (variable-listp (domain s))
           (termp f (domain s)))
  ((enable termp)
   (induct (domain s))))

(prove-lemma var-substp-apply-to-subst (rewrite)
  (implies (and (termp f (range s))
                (termp f (range sg)))
           (termp f (range (apply-to-subst sg s))))
  ((enable termp)))

(prove-lemma value-apply-to-subst (rewrite)
  (implies (member g (domain s))
           (equal (value g (apply-to-subst sg s))
                  (subst t sg (value g s)))))

(prove-lemma non-variablep-not-member-of-variable-listp (rewrite)
  (implies (and (variable-listp d)
                (not (variablep term)))
           (not (member term d)))
  ((induct (member term d))))

(prove-lemma compose-property-reversed (rewrite)
  (implies (and (variable-listp (domain s2))
                (termp flg x))
           (equal (subst flg (compose s1 s2) x)
                  (subst flg s2 (subst flg s1 x))))
  ((enable termp)))
```

```
(prove-lemma compose-property
             (rewrite)
             (implies (and (variable-listp (domain s2))
                           (termp flg x))
                      (equal (subst flg s2 (subst flg s1 x))
                             (subst flg (compose s1 s2) x)))
             ((disable compose)))

(disable compose-property-reversed)

(prove-lemma subst-not-occur (rewrite)
  (implies (and (termp flg x)
                (variable-listp (domain s))
                (disjoint (domain s) (all-vars flg x)))
           (equal (subst flg s x) x))
  ((enable termp)))

(prove-lemma disjoint-range-implies-disjoint-value (rewrite)
  (implies (and (member x (domain s))
                flg
                (disjoint z (all-vars f (range s))))
           (disjoint z (all-vars flg (value x s))))
  ((use (subsetp-disjoint-2
         (x (all-vars flg (value x s)))
         (y (all-vars f (range s)))
         (z z)))))

(prove-lemma disjoint-all-vars-subst (rewrite)
  (implies (and (termp flg x)
                (disjoint z (all-vars flg x))
                (disjoint z (all-vars f (range s))))
           (disjoint z (all-vars flg (subst flg s x))))
  ((enable termp)))

(prove-lemma all-vars-variable-listp (rewrite)
  (implies (variable-listp x)
           (equal (all-vars f x)
                  x))
  ((induct (variable-listp x))))

(prove-lemma variable-listp-append (rewrite)
  (equal (variable-listp (append x y))
         (and (variable-listp (fix-properp x))
              (variable-listp y)))
  ((induct (domain s))))

(prove-lemma termp-list-append (rewrite)
  (equal (termp f (append x y))
         (and (termp f (fix-properp x))
              (termp f y)))
  ((induct (range s))))

(prove-lemma apply-to-subst-append (rewrite)
  (equal (apply-to-subst sg (append s1 s2))
         (append (apply-to-subst sg s1)
                 (apply-to-subst sg s2))))

(prove-lemma subst-apply-to-subst (rewrite)
  (implies (and flg
                (member g (domain s)))
           (equal (subst flg (apply-to-subst sg s) g)
                  (subst flg sg (value g s)))))

(prove-lemma subst-append-not-occur-1 (rewrite)
  (implies (and (termp flg x)
                (variable-listp (domain s1))
                (disjoint (all-vars f (domain s1))
                          (all-vars flg x)))
           (equal (subst flg (append s1 s2) x)
```

```
                  (subst flg s2 x)))
  ((induct (subst flg s2 x))))

(prove-lemma subst-append-not-occur-2 (rewrite)
  (implies (and (termp flg x)
                (variable-listp (domain s2))
                (disjoint (all-vars f (domain s2))
                          (all-vars flg x)))
           (equal (subst flg (append s1 s2) x)
                  (subst flg s1 x)))
  ((induct (subst flg s2 x))))

(prove-lemma apply-to-subst-is-no-op-for-disjoint-domain (rewrite)
  (implies (and (variable-listp (domain s1))
                (alistp s2)
                (termp f (range s2))
                (disjoint (domain s1) (all-vars f (range s2))))
           (equal (apply-to-subst s1 s2)
                  s2)))

(prove-lemma member-subst (rewrite)
  (implies (and flg (member a x))
           (member (subst flg s a)
                   (subst f s x)))
  ((enable member)))

(prove-lemma subsetp-subst (rewrite)
  (implies (subsetp x y)
           (subsetp (subst f s x)
                    (subst f s y)))
  ((enable subsetp)))

(disable instance)
;;;(disable compose) -- COMPOSE is left enabled for use with COMPOSE-PROPERTY
(disable apply-to-subst)
(disable subst)
(disable rembind)
(disable bind)

;;;;; nullify-subst:  a substitution that has a range containing
;;                    no variables

(defn nullify-subst (s)
  (if (listp s)
      (if (listp (car s))
          (cons (cons (caar s) (list (fn)))
                (nullify-subst (cdr s)))
        (nullify-subst (cdr s)))
    nil))

(prove-lemma properp-nullify-subst (rewrite)
  (properp (nullify-subst s)))

(prove-lemma all-vars-f-range-nullify-subst (rewrite)
  (equal (all-vars f (range (nullify-subst s)))
         nil))

(prove-lemma termp-range-nullify-subst (rewrite)
  (termp f (range (nullify-subst s))))

(prove-lemma domain-nullify-subst (rewrite)
  (equal (domain (nullify-subst s))
         (domain s)))

(prove-lemma mapping-nullify-subst (rewrite)
  (implies (alistp s)
           (equal (mapping (nullify-subst s))
                  (mapping s)))
  ((enable mapping)))
```

```
(prove-lemma disjoint-all-vars-subst-nullify-subst (rewrite)
  (implies (termp flg term)
           (disjoint (domain sg)
                     (all-vars flg
                               (subst flg (nullify-subst sg) term))))
  ((enable subst)
   (disable nullify-subst)))

(prove-lemma disjoint-all-vars-range-apply-subst-nullify-subst (rewrite)
  (implies (termp f (range s))
           (disjoint (domain sg)
                     (all-vars f
                               (range (apply-to-subst (nullify-subst sg) s)))))
  ((enable apply-to-subst)
   (disable nullify-subst)))

(disable nullify-subst)

(deftheory substitution-defns
  (instance var-substp compose apply-to-subst subst nullify-subst))

))
```

THE FILE "generalize.events"

```
(setq events '(

;; Requires sets, alists, and terms, which however currently contain a
;; number of rules that aren't really needed here, even indirectly.

;; This is a proof soundness of a slight abstraction of the GENERALIZE
;; command of PC-NQTHM.

;;; Here's what I want to prove.

;;; (implies (and (generalize-okp sg state)
;;;               (valid-state (generalize sg state)))
;;;          (valid-state state))

;;; I also prove the much simpler fact, GENERALIZE-STATEP:

;;; (implies (generalize-okp sg state)
;;;          (statep (generalize sg state)))

;; << 1 >>
(constrain theorem-intro (rewrite)
  (and (implies (and (theorem x)
                     flg)
                (termp flg x))
       (implies (and (theorem x)
                     flg
                     (var-substp s))
                (theorem (subst flg s x))))
  ((theorem (lambda (x) f))))

;; << 2 >>
(defn theorem-list (x)
  (if (listp x)
      (and (theorem (car x))
           (theorem-list (cdr x)))
    (equal x nil)))

;; << 3 >>
(prove-lemma theorem-list-properties (rewrite)
  (and (implies (theorem-list x)
                (termp f x))
       (implies (and (theorem-list x)
                     (var-substp s))
                (theorem-list (subst f s x)))))

;; << 4 >>
(defn statep (state)
  (and (listp state)
       (termp f (car state))
       (variable-listp (cdr state))))

;; << 5 >>
(defn-sk valid-state (state)
  (and (statep state)
       (exists witnessing-instantiation
               (and (var-substp witnessing-instantiation)
                    (subsetp (domain witnessing-instantiation) (cdr state))
                    (theorem-list (subst f witnessing-instantiation (car state)))))))

;; << 6 >>
(defn new-gen-vars (goals free vars)
  (if (listp goals)
      (let ((current-free-vars (intersection free (all-vars t (car goals)))))
        (if (disjoint current-free-vars vars)
            (new-gen-vars (cdr goals) free vars)
          (append current-free-vars
```

```
                  (new-gen-vars (cdr goals) free vars))))
    nil))

;; << 7 >>
(defn cardinality (x)
  (length (make-set x)))

;; Next goal:  get the definition of GEN-CLOSURE accepted.  In fact,
;; the lemma GEN-CLOSURE-ACCEPT below suffices, taking NEW to be
;; (NEW-GEN-VARS GOALS FREE FREE-VARS-SO-FAR), as long as we prove the
;; following lemma, NEW-GEN-VARS-SUBSET.

;; << 8 >>
(prove-lemma new-gen-vars-subset (rewrite)
  (subsetp (new-gen-vars goals free vars)
           free))

;; It is interesting to note that the exact form of the following
;; lemma changed while polishing the proof, since rewrite rules
;; applied to the old version so as to make it irrelevant.

;; << 9 >>
(prove-lemma gen-closure-accept (rewrite)
  (implies (and (not (subsetp new free-vars-so-far))
                (subsetp new free))
           (lessp (difference (difference (length (make-set free))
                                          (length (intersection (make-set free)
                                                                free-vars-so-far)))
                              (length (intersection (set-diff (make-set free)
                                                              free-vars-so-far)
                                                    new)))
                  (difference (length (make-set free))
                              (length (intersection (make-set free)
                                                    free-vars-so-far))))))

;; Here I have a choice:  I could intersect the accumulator with free
;; at the end, or I could assume that it's intersected with free
;; before it's input.  I'll choose the former approach, so that I'll
;; have a simpler rewrite rule and so that I can call gen-closure more
;; simply.  I may wish to commute the arguments to intersection in the
;; exit below, but probably that won't matter because I'll only be
;; talking about membership.

;; << 10 >>
(defn gen-closure (goals free free-vars-so-far)
  ;; Returns the goals with variables among the closure of the vars of
  ;; goals-so-far under the ``occurs in the same goal as'' relation,
  ;; restricted to free.
  (let ((new-free-vars (new-gen-vars goals free free-vars-so-far)))
    (if (subsetp new-free-vars free-vars-so-far)
        (intersection free-vars-so-far free)
      (gen-closure goals free (append new-free-vars free-vars-so-far))))
  ((lessp (cardinality (set-diff free free-vars-so-far)))))

;; << 11 >>
(defn generalize-okp (sg state)
  (and (var-substp sg)
       (statep state)
       (disjoint (domain sg)
                 (all-vars f (car state)))
       (listp (car state))
       (disjoint (domain sg) (cdr state))))

;; << 12 >>
(defn generalize (sg state)
  (let ((g (caar state))
        (p (cdar state))
        (free (cdr state)))
    (let ((new-g (subst t (invert sg) g)))
```

```
      (let ((domain-1
             (gen-closure (cons new-g p)
                          free
                          (all-vars t new-g))))
        (let ((new-free
               (set-diff free
                         (intersection domain-1 (all-vars f (range sg))))))
          (cons (cons new-g p)
                new-free)))))

;; Here is a fact, not needed elsewhere, that is worth noticing, in
;; case we wish to extend the main theorem to a sequence of
;; PC-NQTHM-like commands.

;; << 13 >>
(prove-lemma generalize-statep nil
  (implies (generalize-okp sg state)
           (statep (generalize sg state))))

;; << 14 >>
(defn gen-inst (sg state)
  (let ((s (witnessing-instantiation (generalize sg state)))
        (g (caar state))
        (p (cdar state))
        (free (cdr state)))
    (let ((new-g (subst t (invert sg) g)))
      (let ((domain-1 (gen-closure (cons new-g p)
                                   (cdr state)
                                   (all-vars t new-g))))
        (let ((s1 (restrict s domain-1))
              (s2 (apply-to-subst
                   (nullify-subst sg)
                   (co-restrict s domain-1))))
          (apply-to-subst
           (apply-to-subst s2 sg)
           (append s1 s2)))))))

;; Let’s see that it suffices to prove the result of opening up the
;; conclusion of the main theorem with a particular witness.


;;; (add-axiom main-theorem-1 (rewrite)
;;;   (let ((wit (gen-inst sg state)))
;;;     (implies (and (generalize-okp sg state)
;;;                   (valid-state (generalize sg state)))
;;;              (and (statep state)
;;;                   (var-substp wit)
;;;                   (subsetp (domain wit) (cdr state))
;;;                   (theorem-list (subst f wit (car state)))))))
;;;
;;; (prove-lemma generalize-is-correct (rewrite)
;;;   (implies (and (generalize-okp sg state)
;;;                 (valid-state (generalize sg state)))
;;;            (valid-state state))
;;;   ((disable-theory t)
;;;    (enable-theory ground-zero)
;;;    (enable main-theorem-1)
;;;    (use (valid-state
;;;          (witnessing-instantiation (gen-inst sg state))))))


;; So, it suffices to prove main-theorem-1.  The first three conjuncts
;; of the conclusion are quite trivial.

;; << 15 >>
(prove-lemma main-theorem-1-case-1 (rewrite)
  (implies (generalize-okp sg state)
           (statep state)))
```

```
;; We put one direction of the definition of valid-state here, for
;; efficiency in proofs.

;; << 16 >>
(prove-lemma valid-state-opener (rewrite)
  (equal (valid-state state)
         (and (statep state)
              (let ((witnessing-instantiation (witnessing-instantiation state)))
                (and (var-substp witnessing-instantiation)
                     (subsetp (domain witnessing-instantiation) (cdr state))
                     (theorem-list (subst f witnessing-instantiation (car state)))))))
  ((disable-theory t)
   (enable-theory ground-zero)
   (use (valid-state (witnessing-instantiation (witnessing-instantiation state))))))

;; << 17 >>
(prove-lemma main-theorem-1-case-2 (rewrite)
  (let ((wit (gen-inst sg state)))
    (implies (and (generalize-okp sg state)
                  (valid-state (generalize sg state)))
             (var-substp wit)))
  ((disable generalize)))

;; << 18 >>
(prove-lemma subsetp-cdr-generalize (rewrite)
  (subsetp (cdr (generalize sg state)) (cdr state)))

;; At this point I had to prove SUBSETP-SET-DIFF-SUFFICIENCY because
;; of some lemma that was created during the polishing process
;; (perhaps DOMAIN-RESTRICT).

;; << 19 >>
(prove-lemma main-theorem-1-case-3 (rewrite)
  (let ((wit (gen-inst sg state)))
    (implies (valid-state (generalize sg state))
             (subsetp (domain wit) (cdr state))))
  ((disable generalize)))

;; So now we only have to prove MAIN-THEOREM-1-CASE-4 (written here
;; without use of LET):


;;; (add-axiom main-theorem-1-case-4 (rewrite)
;;;   (implies (and (generalize-okp sg state)
;;;                 (valid-state (generalize sg state)))
;;;            (theorem-list (subst f (gen-inst sg state) (car state)))))
;;;
;;; (prove-lemma main-theorem-1 (rewrite)
;;;   (let ((wit (gen-inst sg state)))
;;;     (implies (and (generalize-okp sg state)
;;;                   (valid-state (generalize sg state)))
;;;              (and (statep state)
;;;                   (var-substp wit)
;;;                   (subsetp (domain wit) (cdr state))
;;;                   (theorem-list (subst f wit (car state))))))
;;;   ((disable-theory t)
;;;    (enable-theory ground-zero)
;;;    (enable main-theorem-1-case-1 main-theorem-1-case-2
;;;            main-theorem-1-case-3 main-theorem-1-case-4)))


;; << 20 >>
(defn gen-setting-substitutions (s1 s2 sg)
  (and (var-substp s1)
       (var-substp s2)
       (var-substp sg)
       (disjoint (domain s1) (domain sg))
       (disjoint (domain s2) (domain sg))
       (disjoint (all-vars f (range sg))
```

```
                (domain s1))
          (disjoint (all-vars f (range s2)) (domain sg))))

;; << 21 >>
(defn main-hyps (s1 s2 sg g p)
  (and (termp t g)
       (disjoint (all-vars t g) (domain sg))
       (termp f p)
       (disjoint (all-vars f p) (domain sg))
       (gen-setting-substitutions s1 s2 sg)
       (theorem-list (subst f (append s1 s2)
                              (cons (subst t (invert sg) g) p)))))

;; The goal above, MAIN-THEOREM-1-CASE-4, should follow from the
;; following two lemmas.


;;; (add-axiom main-hyps-suffice (rewrite)
;;;   (implies (and (listp goals)
;;;                 (main-hyps s1 s2 sg (car goals) (cdr goals)))
;;;            (theorem-list (subst f
;;;                                 (apply-to-subst (apply-to-subst s2 sg)
;;;                                                 (append s1 s2))
;;;                                 goals))))
;;;
;;; (add-axiom main-hyps-relieved (rewrite)
;;;   (let ((g (caar state))
;;;         (p (cdar state))
;;;         (free (cdr state))
;;;         (s (witnessing-instantiation (generalize sg state))))
;;;     (let ((new-g (subst t (invert sg) g)))
;;;       (let ((domain-1
;;;              (gen-closure (cons new-g p) free (all-vars t new-g))))
;;;         (let ((s1 (restrict s domain-1))
;;;               (s2 (apply-to-subst (nullify-subst sg)
;;;                                   (co-restrict s domain-1))))
;;;           (implies (and (generalize-okp sg state)
;;;                         (valid-state (generalize sg state)))
;;;                    (main-hyps s1 s2 sg g p)))))))
;;;
;;; (prove-lemma main-theorem-1-case-4 (rewrite)
;;;   (implies (and (generalize-okp sg state)
;;;                 (valid-state (generalize sg state)))
;;;            (theorem-list (subst f (gen-inst sg state) (car state))))
;;;   ((disable-theory t)
;;;    (enable-theory ground-zero)
;;;    (enable gen-inst main-hyps-suffice generalize-okp main-hyps-relieved)))


;; So, now let us start with MAIN-HYPS-SUFFICE.  It should follow from
;; two subgoals, as shown:


;;; (add-axiom main-hyps-suffice-first (rewrite)
;;;   (implies (main-hyps s1 s2 sg g p)
;;;            (theorem (subst t
;;;                            (apply-to-subst (apply-to-subst s2 sg)
;;;                                            (append s1 s2))
;;;                            g))))
;;;
;;; (add-axiom main-hyps-suffice-rest (rewrite)
;;;   (implies (main-hyps s1 s2 sg g p)
;;;            (theorem-list (subst f
;;;                                 (apply-to-subst (apply-to-subst s2 sg)
;;;                                                 (append s1 s2))
;;;                                 p))))
;;;
;;; (prove-lemma main-hyps-suffice (rewrite)
;;;   (implies (and (listp goals)
```

```
;;;                 (main-hyps s1 s2 sg (car goals) (cdr goals)))
;;;            (theorem-list (subst f
;;;                                 (apply-to-subst (apply-to-subst s2 sg)
;;;                                                 (append s1 s2))
;;;                                 goals)))
;;;   ((disable-theory t)
;;;    (enable-theory ground-zero)
;;;    (enable theorem-list subst main-hyps-suffice-first main-hyps-suffice-rest)))


;; Consider the first of these.  Although COMPOSE-PROPERTY is
;; used in the proof (because it's enabled), it's actually not
;; necessary.  A proof took slightly over 10 minutes with the rule
;; enabled, and roughly 9 minutes without; at least this was the
;; case at one point during the proof development.

;; << 22 >>
(prove-lemma main-hyps-suffice-first-lemma-general nil
  (implies (and (termp flg g)
                (disjoint (all-vars flg g) (domain sg))
                (gen-setting-substitutions s1 s2 sg)
                (equal sg-1 (invert sg)))
           (equal (subst flg
                         (apply-to-subst (apply-to-subst s2 sg)
                                         (append s1 s2))
                         g)
                  (subst flg (apply-to-subst s2 sg)
                         (subst flg (append s1 s2)
                                (subst flg sg-1 g))))))
  ((induct (subst flg sg-1 g))))

;; << 23 >>
(prove-lemma main-hyps-suffice-first-lemma (rewrite)
  (implies (and (termp t g)
                (disjoint (all-vars t g) (domain sg))
                (gen-setting-substitutions s1 s2 sg))
           (equal (subst t
                         (apply-to-subst (apply-to-subst s2 sg)
                                         (append s1 s2))
                         g)
                  (subst t (apply-to-subst s2 sg)
                         (subst t (append s1 s2)
                                (subst t (invert sg) g)))))
  ((use (main-hyps-suffice-first-lemma-general (flg t) (sg-1 (invert sg))))
   (disable-theory t)
   (enable-theory ground-zero)))

;; << 24 >>
(prove-lemma main-hyps-suffice-first (rewrite)
  (implies (main-hyps s1 s2 sg g p)
           (theorem (subst t
                           (apply-to-subst (apply-to-subst s2 sg)
                                           (append s1 s2))
                           g)))
  ;; Disabling compose-property is necessary so that the fact
  ;; that theoremhood is inherited upon substitution is used.  Disabling
  ;; APPLY-TO-SUBST-APPEND is necessary so that
  ;; MAIN-HYPS-SUFFICE-FIRST-LEMMA is used (a Knuth-Bendix sort of
  ;; problem).
  ((disable compose-property apply-to-subst-append)))

;; The following is useful with s = (append s1 s2).

;; << 25 >>
(prove-lemma main-hyps-suffice-rest-lemma (rewrite)
  (implies (and (termp flg p)
                (variable-listp (domain sg))
                (disjoint (all-vars flg p) (domain sg)))
           (equal (subst flg
```

```
                         (apply-to-subst (apply-to-subst s2 sg)
                                         s)
                         p)
                  (subst flg
                         (apply-to-subst s2 sg)
                         (subst flg s p)))))

;; << 26 >>
(prove-lemma main-hyps-suffice-rest (rewrite)
  (implies (main-hyps s1 s2 sg g p)
           (theorem-list (subst f
                                (apply-to-subst (apply-to-subst s2 sg)
                                                (append s1 s2))
                                p)))
  ;; If I don't disable compose-property I get an infinite loop
  ;; in the rewriter, it seems.
  ((disable apply-to-subst-append compose-property)))

;; << 27 >>
(prove-lemma main-hyps-suffice (rewrite)
  (implies (and (listp goals)
                (main-hyps s1 s2 sg (car goals) (cdr goals)))
           (theorem-list (subst f
                                (apply-to-subst (apply-to-subst s2 sg)
                                                (append s1 s2))
                                goals)))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable theorem-list subst main-hyps-suffice-first main-hyps-suffice-rest)))

;; I'll disable the two lemmas used above so that I avoid the possibility
;; of looping with compose-property.

;; << 28 >>
(disable main-hyps-suffice-first-lemma)
;; << 29 >>
(disable main-hyps-suffice-rest-lemma)

;; All that remains now is to prove MAIN-HYPS-RELIEVED.  If we open up
;; MAIN-HYPS we see what the necessary subgoals are.  Recall the
;; definition of MAIN-HYPS:


;;; (defn main-hyps (s1 s2 sg g p)
;;;   (and (termp t g)
;;;        (disjoint (all-vars t g) (domain sg))
;;;        (termp f p)
;;;        (disjoint (all-vars f p) (domain sg))
;;;        (gen-setting-substitutions s1 s2 sg)
;;;        (theorem-list (subst f (append s1 s2)
;;;                             (cons (subst t (invert sg) g) p)))))


;; << 30 >>
(prove-lemma main-hyps-relieved-1 (rewrite)
  (let ((g (caar state)))
    (implies (generalize-okp sg state)
             (termp t g))))

;; << 31 >>
(prove-lemma main-hyps-relieved-2 (rewrite)
  (let ((g (caar state)))
    (implies (generalize-okp sg state)
             (disjoint (all-vars t g) (domain sg)))))

;; << 32 >>
(prove-lemma main-hyps-relieved-3 (rewrite)
  (let ((p (cdar state)))
    (implies (generalize-okp sg state)
```

```
             (termp f p))))

;; << 33 >>
(prove-lemma main-hyps-relieved-4 (rewrite)
  (let ((p (cdar state)))
    (implies (generalize-okp sg state)
             (disjoint (all-vars f p) (domain sg)))))


;;; (add-axiom main-hyps-relieved-5 (rewrite)
;;;   (let ((g (caar state))
;;;         (p (cdar state))
;;;         (free (cdr state))
;;;         (s (witnessing-instantiation (generalize sg state))))
;;;     (let ((new-g (subst t (invert sg) g)))
;;;       (let ((domain-1
;;;              (gen-closure (cons new-g p) free (all-vars t new-g))))
;;;         (let ((s1 (restrict s domain-1))
;;;               (s2 (apply-to-subst (nullify-subst sg)
;;;                                   (co-restrict s domain-1))))
;;;           (implies (and (generalize-okp sg state)
;;;                         (valid-state (generalize sg state)))
;;;                    (gen-setting-substitutions s1 s2 sg)))))))
;;;
;;; (add-axiom main-hyps-relieved-6 (rewrite)
;;;   (let ((g (caar state))
;;;         (p (cdar state))
;;;         (free (cdr state))
;;;         (s (witnessing-instantiation (generalize sg state))))
;;;     (let ((new-g (subst t (invert sg) g)))
;;;       (let ((domain-1
;;;              (gen-closure (cons new-g p) free (all-vars t new-g))))
;;;         (let ((s1 (restrict s domain-1))
;;;               (s2 (apply-to-subst (nullify-subst sg)
;;;                                   (co-restrict s domain-1))))
;;;           (implies (and (generalize-okp sg state)
;;;                         (valid-state (generalize sg state)))
;;;                    (theorem-list (subst f (append s1 s2)
;;;                                         (cons (subst t (invert sg) g) p)))))))))
;;;
;;; (prove-lemma main-hyps-relieved (rewrite)
;;;   (let ((g (caar state))
;;;         (p (cdar state))
;;;         (free (cdr state))
;;;         (s (witnessing-instantiation (generalize sg state))))
;;;     (let ((new-g (subst t (invert sg) g)))
;;;       (let ((domain-1
;;;              (gen-closure (cons new-g p) free (all-vars t new-g))))
;;;         (let ((s1 (restrict s domain-1))
;;;               (s2 (apply-to-subst (nullify-subst sg)
;;;                                   (co-restrict s domain-1))))
;;;           (implies (and (generalize-okp sg state)
;;;                         (valid-state (generalize sg state)))
;;;                    (main-hyps s1 s2 sg g p))))))
;;;   ((disable-theory t)
;;;    (enable-theory ground-zero)
;;;    (enable main-hyps main-hyps-relieved-1 main-hyps-relieved-2
;;;            main-hyps-relieved-3 main-hyps-relieved-4
;;;            main-hyps-relieved-5 main-hyps-relieved-6)))


;; So, it remains to prove the goals MAIN-HYPS-RELIEVED-5 and
;; MAIN-HYPS-RELIEVED-6.  Let us start with the first.  Opening up
;; GEN-SETTING-SUBSTITUTIONS gives us a number of subgoals.

;; The case for the first two conjuncts of GEN-SETTING-SUBSTITUTIONS
;; do not require knowledge about DOMAIN-1 (or G, P, FREE, or NEW-G),
;; but simply follow from the validity of the state (GENERALIZE SG
;; STATE).  Disabling GENERALIZE is very useful for the first of these
```

```
;; (probably not necessary, though I didn't let the prover run long
;; enough to find out for sure).

;; << 34 >>
(prove-lemma main-hyps-relieved-5-lemma-1 (rewrite)
  (let ((s (witnessing-instantiation (generalize sg state))))
    (let ((s1 (restrict s domain-1))
          (s2 (apply-to-subst (nullify-subst sg)
                              (co-restrict s domain-1))))
      (implies (valid-state (generalize sg state))
               (and (var-substp s1)
                    (var-substp s2)))))
  ((disable generalize)))

;; The next case is trivial.

;; << 35 >>
(prove-lemma main-hyps-relieved-5-lemma-2 (rewrite)
  (implies (generalize-okp sg state)
           (var-substp sg)))

;; For the next two conjuncts of GEN-SETTING-SUBSTITUTIONS we first
;; observe that (DOMAIN S) is disjoint from (DOMAIN SG), and then we
;; use SUBSETP-DISJOINT-3 where X is the domain of S1 or S2, Y is the
;; domain of S, and Z is the domain of SG:
;;             (IMPLIES (AND (SUBSETP X Y) (DISJOINT Z Y))
;;                      (DISJOINT X Z))

;; << 36 >>
(prove-lemma witnessing-instantiation-is-disjoint-from-generalizing-substitution nil
  (let ((s (witnessing-instantiation (generalize sg state))))
    (implies (and (generalize-okp sg state)
                  (valid-state (generalize sg state)))
             (disjoint (domain s) (domain sg)))))

;; Here we abstract away DOMAIN-1 (and hence G, P, FREE, and NEW-G).
;; Incidentally, a similar phenomenon occurred here to the one
;; reported just above the statement above of MAIN-THEOREM-1-CASE-3:
;; final polishing resulted in the need for another lemma.  That extra
;; lemma is DISJOINT-SET-DIFF-SUFFICIENCY in this case, to be found in
;; "sets.events".

;; << 37 >>
(prove-lemma main-hyps-relieved-5-lemma-3 (rewrite)
  (let ((s (witnessing-instantiation (generalize sg state))))
    (let ((s1 (restrict s domain-1))
          (s2 (apply-to-subst (nullify-subst sg)
                              (co-restrict s domain-1))))
      (implies (and (generalize-okp sg state)
                    (valid-state (generalize sg state)))
               (and (disjoint (domain s1) (domain sg))
                    (disjoint (domain s2) (domain sg))))))
  ((use (witnessing-instantiation-is-disjoint-from-generalizing-substitution))
   (disable generalize-okp valid-state-opener generalize)))

;; The lemma MAIN-HYPS-RELIEVED-5-LEMMA-4-WIT is true because the
;; domain of s is contained in the free variables of the generalized
;; state (by choice, i.e. definition, of WITNESSING-INSTANTIATION),
;; which is disjoint from the intersection of the indicated
;; GEN-CLOSURE with the variables in the range of sg.  I'll use a
;; trick that I learned from Ken Kunen (definable Skolem function is
;; all, really) to reduce disjointness considerations to membership
;; considerations.

;; << 38 >>
(prove-lemma main-hyps-relieved-5-lemma-4-wit (rewrite)
  (let ((g (caar state))
        (p (cdar state))
        (free (cdr state))
```

```
        (s (witnessing-instantiation (generalize sg state))))
    (let ((new-g (subst t (invert sg) g)))
      (let ((domain-1
             (gen-closure (cons new-g p) free (all-vars t new-g))))
        (let ((s1 (restrict s domain-1)))
          (implies (and (generalize-okp sg state)
                        (valid-state (generalize sg state))
                        (member wit (all-vars f (range sg)))
                        (member wit (domain s)))
                   (not (member wit domain-1)))))))
  ((disable gen-closure subst invert all-vars restrict)))

;; << 39 >>
(prove-lemma main-hyps-relieved-5-lemma-4 (rewrite)
  (let ((g (caar state))
        (p (cdar state))
        (free (cdr state))
        (s (witnessing-instantiation (generalize sg state))))
    (let ((new-g (subst t (invert sg) g)))
      (let ((domain-1
             (gen-closure (cons new-g p) free (all-vars t new-g))))
        (let ((s1 (restrict s domain-1)))
          (implies (and (generalize-okp sg state)
                        (valid-state (generalize sg state)))
                   (disjoint (all-vars f (range sg))
                             (domain s1)))))))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable domain-restrict member-intersection
           disjoint-wit-witnesses main-hyps-relieved-5-lemma-4-wit)))

;; << 40 >>
(prove-lemma main-hyps-relieved-5-lemma-5 (rewrite)
  (let ((g (caar state))
        (p (cdar state))
        (free (cdr state))
        (s (witnessing-instantiation (generalize sg state))))
    (let ((new-g (subst t (invert sg) g)))
      (let ((domain-1
             (gen-closure (cons new-g p) free (all-vars t new-g))))
        (let ((s2 (apply-to-subst (nullify-subst sg)
                                  (co-restrict s domain-1))))
          (implies (and (generalize-okp sg state)
                        (valid-state (generalize sg state)))
                   (disjoint (all-vars f (range s2))
                             (domain sg))))))))

;; << 41 >>
(prove-lemma main-hyps-relieved-5 (rewrite)
  (let ((g (caar state))
        (p (cdar state))
        (free (cdr state))
        (s (witnessing-instantiation (generalize sg state))))
    (let ((new-g (subst t (invert sg) g)))
      (let ((domain-1
             (gen-closure (cons new-g p) free (all-vars t new-g))))
        (let ((s1 (restrict s domain-1))
              (s2 (apply-to-subst (nullify-subst sg)
                                  (co-restrict s domain-1))))
          (implies (and (generalize-okp sg state)
                        (valid-state (generalize sg state)))
                   (gen-setting-substitutions s1 s2 sg))))))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable gen-setting-substitutions
           main-hyps-relieved-5-lemma-1 main-hyps-relieved-5-lemma-2
           main-hyps-relieved-5-lemma-3
           main-hyps-relieved-5-lemma-4 main-hyps-relieved-5-lemma-5)))
```

```
;; Now we begin the remaining goal, MAIN-HYPS-RELIEVED-6.  The idea is
;; to show that the appropriate goal list is a theorem-list by showing
;; separately that the first and the rest are theorems, since the
;; reasons are slightly different.  The FIRST is a theorem because its
;; free vars are all in domain-1, hence in the domain of s1; so, s2
;; can be dropped from the APPEND.  The REST all have the property
;; that their free vars are contained in or disjoint from domain-1,
;; and for those disjoint from it, they do not contain variables from
;; the domain of sg.  Notice that the new current (FIRST) goal may
;; violate the latter requirement, since it may have no free vars at
;; all but contain vars from the domain of sg.  That's why we have to
;; make a special case out of it.


;;; (add-axiom main-hyps-relieved-6-first (rewrite)
;;;   (let ((g (caar state))
;;;         (p (cdar state))
;;;         (free (cdr state))
;;;         (s (witnessing-instantiation (generalize sg state))))
;;;     (let ((new-g (subst t (invert sg) g)))
;;;       (let ((domain-1
;;;              (gen-closure (cons new-g p) free (all-vars t new-g))))
;;;         (let ((s1 (restrict s domain-1))
;;;               (s2 (apply-to-subst (nullify-subst sg)
;;;                                   (co-restrict s domain-1))))
;;;           (implies (and (generalize-okp sg state)
;;;                         (valid-state (generalize sg state)))
;;;                    (theorem (subst t (append s1 s2)
;;;                                    new-g))))))))
;;;
;;; (add-axiom main-hyps-relieved-6-rest (rewrite)
;;;   (let ((g (caar state))
;;;         (p (cdar state))
;;;         (free (cdr state))
;;;         (s (witnessing-instantiation (generalize sg state))))
;;;     (let ((new-g (subst t (invert sg) g)))
;;;       (let ((domain-1
;;;              (gen-closure (cons new-g p) free (all-vars t new-g))))
;;;         (let ((s1 (restrict s domain-1))
;;;               (s2 (apply-to-subst (nullify-subst sg)
;;;                                   (co-restrict s domain-1))))
;;;           (implies (and (generalize-okp sg state)
;;;                         (valid-state (generalize sg state)))
;;;                    (theorem-list (subst f (append s1 s2) p))))))))
;;;
;;; (prove-lemma main-hyps-relieved-6 (rewrite)
;;;   (let ((g (caar state))
;;;         (p (cdar state))
;;;         (free (cdr state))
;;;         (s (witnessing-instantiation (generalize sg state))))
;;;     (let ((new-g (subst t (invert sg) g)))
;;;       (let ((domain-1
;;;              (gen-closure (cons new-g p) free (all-vars t new-g))))
;;;         (let ((s1 (restrict s domain-1))
;;;               (s2 (apply-to-subst (nullify-subst sg)
;;;                                   (co-restrict s domain-1))))
;;;           (implies (and (generalize-okp sg state)
;;;                         (valid-state (generalize sg state)))
;;;                    (theorem-list (subst f (append s1 s2)
;;;                                         (cons (subst t (invert sg) g) p))))))))
;;;   ((disable-theory t)
;;;    (enable-theory ground-zero)
;;;    (enable main-hyps-relieved-6-first main-hyps-relieved-6-rest
;;;            subst theorem-list)))


;; The first is true because the free vars in new-g are all in the
;; domain of s1, since they are all in domain-1.  By the way, the
;; proof-checker was useful here; I dove to the subst term (after
```

```
;; adding abbreviations and promoting hypotheses) and saw that I
;; wanted to rewrite with SUBST-APPEND-NOT-OCCUR-2.  I also notice the
;; need for GEN-CLOSURE-CONTAINS-THIRD-ARG during the attempt to prove
;; a goal.

;; First, we only want to open up GENERALIZE when we are looking at
;; goals, not when we are simply asking about the witnessing
;; substitution.  I believe that this speeds up the proofs
;; considerably.

;; << 42 >>
(prove-lemma car-generalize (rewrite)
  (equal (car (generalize sg state))
         (cons (subst t (invert sg) (caar state))
               (cdar state))))

;; << 43 >>
(disable generalize)

;; Inspection of the proof of a subgoal of MAIN-HYPS-RELIEVED-6-FIRST
;; suggests that we need the following lemma.  Actually, before the
;; final polishing it was the case that the following version sufficed.
;; But final polishing led me to prove a ''better'' version, as well
;; as the lemma DISJOINT-SET-DIFF-GENERAL in "sets.events".


;;; (prove-lemma gen-closure-contains-third-arg (rewrite)
;;;   (implies (subsetp domain free)
;;;            (subsetp (intersection domain vars)
;;;                     (gen-closure goals free vars))))


;; << 44 >>
(prove-lemma gen-closure-contains-third-arg (rewrite)
  (implies (subsetp x (intersection free vars))
           (subsetp x
                    (gen-closure goals free vars))))

;; << 45 >>
(prove-lemma main-hyps-relieved-6-first (rewrite)
  (let ((g (caar state))
        (p (cdar state))
        (free (cdr state))
        (s (witnessing-instantiation (generalize sg state))))
    (let ((new-g (subst t (invert sg) g)))
      (let ((domain-1
             (gen-closure (cons new-g p) free (all-vars t new-g))))
        (let ((s1 (restrict s domain-1))
              (s2 (apply-to-subst (nullify-subst sg)
                                  (co-restrict s domain-1))))
          (implies (and (generalize-okp sg state)
                        (valid-state (generalize sg state)))
                   (theorem (subst t (append s1 s2) new-g)))))))))

;; Now we embark on the final goal, MAIN-HYPS-RELIEVED-6-REST.  The
;; idea is that one splits the witnessing substitution s into two
;; appropriate parts, s1 and s2.  These parts are the respective
;; restriction and (approximately) co-restriction of the original
;; witnessing substitution s to some set that is ''closed'' in the
;; appropriate sense.  Actually, the co-restriction is allowed to have
;; a substitution applied to it, whose domain is disjoint from the
;; variables occurring in goals ''outside'' that closure.  Below we
;; give the lemmas and the proof of MAIN-HYPS-RELIEVED-6 from those
;; lemmas.  But first let us introduce the necessary notions.

;; << 46 >>
(defn all-vars-disjoint-or-subsetp (p free x)
  ;; says that every goals's free variables are either contained
  ;; in x or are disjoint from x
```

```
  (if (listp p)
      (and (or (subsetp (intersection free (all-vars t (car p)))
                        x)
               (disjoint (intersection free (all-vars t (car p)))
                         x))
           (all-vars-disjoint-or-subsetp (cdr p) free x))
    t))

;; Our plan will be to show that (CDAR STATE), i.e. p, has the above
;; property with respect to the free variables of the generalized
;; state and the appropriate gen-closure.  In cases where one applies
;; a substitution of the form (append s1 s2) to such a list of goals,
;; where the domain of s1 is contained in the intersection of those
;; free variables with that closure and the domain of s2 is disjoint
;; from that intersection, we expect that the result is a theorem-list
;; if each of the following are theorem-lists:  apply s1 to the goals
;; whose vars intersect its domain, and apply s2 to the rest.
;; Reduction rules about applying restrictions etc. will then finish
;; the job.

;; Notice the similarity of the following definition with new-gen-vars.
;; Think of vars as the closure variables, and free as the free variable
;; set within which this all ‘‘takes place’’.

;; << 47 >>
(defn goals-disjoint-from-vars (goals free vars)
  (if (listp goals)
      (let ((current-free-vars (intersection free (all-vars t (car goals)))))
        (if (disjoint current-free-vars vars)
            (cons (car goals)
                  (goals-disjoint-from-vars (cdr goals) free vars))
          (goals-disjoint-from-vars (cdr goals) free vars)))
    nil))

;; Now all that remains is MAIN-HYPS-RELIEVED-6-REST.  I originally
;; forgot the (TERMP F P) hypothesis of
;; MAIN-HYPS-RELIEVED-6-REST-GENERALIZATION below, but it wasn’t very
;; hard to back up and fix this.


;;; (add-axiom main-hyps-relieved-6-rest-generalization (rewrite)
;;;   (let ((s1 (restrict s domain-1))
;;;         (s2 (apply-to-subst (nullify-subst sg)
;;;                             (co-restrict s domain-1))))
;;;     (implies (and (var-substp sg)
;;;                   (var-substp s)
;;;                   (subsetp (domain s) new-free)
;;;                   (termp f p)
;;;                   (theorem-list (subst f s p))
;;;                   (disjoint (domain sg)
;;;                             (all-vars f (goals-disjoint-from-vars
;;;                                          p new-free domain-1)))
;;;                   (all-vars-disjoint-or-subsetp p new-free domain-1))
;;;              (theorem-list (subst f (append s1 s2) p)))))
;;;
;;; (add-axiom main-hyps-relieved-6-rest-lemma-1 (rewrite)
;;;   (let ((g (caar state))
;;;         (p (cdar state))
;;;         (free (cdr state))
;;;         (s (witnessing-instantiation (generalize sg state))))
;;;     (let ((new-g (subst t (invert sg) g)))
;;;       (let ((domain-1
;;;              (gen-closure (cons new-g p) free (all-vars t new-g))))
;;;         (let ((s1 (restrict s domain-1))
;;;               (s2 (apply-to-subst (nullify-subst sg)
;;;                                   (co-restrict s domain-1))))
;;;           (implies (and (generalize-okp sg state)
;;;                         (valid-state (generalize sg state)))
;;;                    (disjoint (domain sg)
```

```
;;;                              (all-vars f (goals-disjoint-from-vars
;;;                                           p (cdr (generalize sg state))
;;;                                           domain-1)))))))))
;;;
;;; ;; Minor note:  I used the BREAK-LEMMA feature of NQTHM to realize
;;; ;; that I needed the following lemma.
;;;
;;; (add-axiom main-hyps-relieved-6-rest-lemma-2 (rewrite)
;;;   (let ((g (caar state))
;;;         (p (cdar state))
;;;         (free (cdr state))
;;;         (s (witnessing-instantiation (generalize sg state))))
;;;     (let ((new-g (subst t (invert sg) g)))
;;;       (let ((domain-1
;;;              (gen-closure (cons new-g p) free (all-vars t new-g))))
;;;         (let ((s1 (restrict s domain-1))
;;;               (s2 (apply-to-subst (nullify-subst sg)
;;;                                   (co-restrict s domain-1))))
;;;           (implies (and (generalize-okp sg state)
;;;                         (valid-state (generalize sg state)))
;;;                    (all-vars-disjoint-or-subsetp p (cdr (generalize sg state))
;;;                                                  domain-1)))))))
;;;
;;; (prove-lemma main-hyps-relieved-6-rest (rewrite)
;;;   (let ((g (caar state))
;;;         (p (cdar state))
;;;         (free (cdr state))
;;;         (s (witnessing-instantiation (generalize sg state))))
;;;     (let ((new-g (subst t (invert sg) g)))
;;;       (let ((domain-1
;;;              (gen-closure (cons new-g p) free (all-vars t new-g))))
;;;         (let ((s1 (restrict s domain-1))
;;;               (s2 (apply-to-subst (nullify-subst sg)
;;;                                   (co-restrict s domain-1))))
;;;           (implies (and (generalize-okp sg state)
;;;                         (valid-state (generalize sg state)))
;;;                    (theorem-list (subst f (append s1 s2) p)))))))
;;;   ((disable-theory t)
;;;    (enable-theory ground-zero)
;;;    (enable ;; so that we can get at p from (car state):
;;;            theorem-list subst car-generalize
;;;            ;; relieving hyps of main-hyps-relieved-6-rest-generalization:
;;;            main-hyps-relieved-6-rest-lemma-1 main-hyps-relieved-6-rest-lemma-2
;;;            ;; to relieve the (termp f p) hypothesis in
;;;            ;; main-hyps-relieved-6-rest-generalization:
;;;            statep termp-list-cons
;;;            generalize-okp valid-state-opener
;;;            main-hyps-relieved-6-rest-generalization)))


;; At this point I did a sanity check and sure enough, the pushed
;; lemmas all go through at this point:  MAIN-HYPS-RELIEVED-6,
;; MAIN-HYPS-RELIEVED, MAIN-THEOREM-1-CASE-4, MAIN-THEOREM-1, and
;; GENERALIZE-IS-CORRECT.


;; It remains to prove MAIN-HYPS-RELIEVED-6-REST-LEMMA-1,
;; MAIN-HYPS-RELIEVED-6-REST-LEMMA-2, and
;; MAIN-HYPS-RELIEVED-6-REST-GENERALIZATION.

;; For the first of these we need the following trivial observation.

;; << 48 >>
(prove-lemma goals-disjoint-from-vars-subsetp (rewrite)
  (subsetp (goals-disjoint-from-vars goals free vars)
           goals))

;; Unfortunately the observation above doesn't quite suffice, because
;; of a technical problem with free variables in hypotheses.  The
```

```
;; following consequence does, though.

;; << 49 >>
(prove-lemma disjoint-all-vars-goals-disjoint-from-vars (rewrite)
  (implies (disjoint x (all-vars f goals))
           (disjoint x (all-vars f (goals-disjoint-from-vars goals free vars))))
  ((use (all-vars-f-monotone (x (goals-disjoint-from-vars goals free vars))
                             (y goals)))
   (disable all-vars-f-monotone)))

;; << 50 >>
(prove-lemma main-hyps-relieved-6-rest-lemma-1 (rewrite)
  (let ((g (caar state))
        (p (cdar state))
        (free (cdr state))
        (s (witnessing-instantiation (generalize sg state))))
    (let ((new-g (subst t (invert sg) g)))
      (let ((domain-1
             (gen-closure (cons new-g p) free (all-vars t new-g))))
        (let ((s1 (restrict s domain-1))
              (s2 (apply-to-subst (nullify-subst sg)
                                  (co-restrict s domain-1))))
          (implies (and (generalize-okp sg state)
                        (valid-state (generalize sg state)))
                   (disjoint (domain sg)
                             (all-vars f (goals-disjoint-from-vars
                                          p (cdr (generalize sg state))
                                          domain-1)))))))))

;; The next goal, MAIN-HYPS-RELIEVED-6-REST-LEMMA-2, needs the lemma
;; ALL-VARS-DISJOINT-OR-SUBSETP-GEN-CLOSURE below.  That lemma's
;; mechanical proof depends on the trivial observation
;; DISJOINT-INTERSECTION3-MIDDLE in file sets.events.

;; << 51 >>
(prove-lemma all-vars-disjoint-or-subsetp-gen-closure
         (rewrite)
         (implies (subsetp new-free free)
                  (all-vars-disjoint-or-subsetp
                   goals new-free (gen-closure (cons g goals) free vars))))

;; << 52 >>
(prove-lemma main-hyps-relieved-6-rest-lemma-2 (rewrite)
  (let ((g (caar state))
        (p (cdar state))
        (free (cdr state))
        (s (witnessing-instantiation (generalize sg state))))
    (let ((new-g (subst t (invert sg) g)))
      (let ((domain-1
             (gen-closure (cons new-g p) free (all-vars t new-g))))
        (let ((s1 (restrict s domain-1))
              (s2 (apply-to-subst (nullify-subst sg)
                                  (co-restrict s domain-1))))
          (implies (and (generalize-okp sg state)
                        (valid-state (generalize sg state)))
                   (all-vars-disjoint-or-subsetp
                    p (cdr (generalize sg state)) domain-1)))))))

;; Finally, all that's left is
;; MAIN-HYPS-RELIEVED-6-REST-GENERALIZATION.  An attempted proof by
;; induction of that theorem results in 11 goals, all but one of which
;; goes through automatically.  The tech. report shows how I used
;; PC-NQTHM to figure things out.  In particular, our problems
;; are now reduced to the following goal.

;;; (SUBST T (NULLIFY-SUBST SG)
;;;        (SUBST T (CO-RESTRICT S DOMAIN-1)
;;;               X))
```

```
;; We need the lemma SUBST-APPLY-TO-SUBST-ELIMINATOR below (which is
;; used under the substitution where S gets (CO-RESTRICT S DOMAIN-1)
;; and SG gets (NULLIFY-SUBST SG)).  However, we'll immediately derive
;; the desired consequence and then disable this lemma, since it
;; appears that it would loop with COMPOSE-PROPERTY.

;; << 53 >>
(prove-lemma subst-apply-to-subst-eliminator (rewrite)
  (implies (and (variable-listp (domain sg))
                (variable-listp (domain s))
                (termp t x)
                (disjoint (domain sg) (all-vars t x)))
           (equal (subst t (apply-to-subst sg s) x)
                  (subst t sg
                         (subst t s x)))))

;; << 54 >>
(prove-lemma theorem-subst-apply-to-subst-with-disjoint-domain (rewrite)
  (implies (and (var-substp sg)
                (var-substp s)
                (termp t x)
                (disjoint (domain sg) (all-vars t x))
                (theorem (subst t s x)))
           (theorem (subst t (apply-to-subst sg s) x)))
  ((disable compose-property)))

;; << 55 >>
(disable subst-apply-to-subst-eliminator)

;; The proof of the remaining goal should go through now, one might
;; think.  However, we need one more observation first, because we
;; need to apply the following lemma.


;;; (PROVE-LEMMA SUBST-CO-RESTRICT
;;;              (REWRITE)
;;;              (IMPLIES (AND (DISJOINT X
;;;                                      (INTERSECTION (DOMAIN S)
;;;                                                    (ALL-VARS FLG TERM)))
;;;                            (VARIABLE-LISTP (DOMAIN S))
;;;                            (TERMP FLG TERM))
;;;                       (EQUAL (SUBST FLG (CO-RESTRICT S X) TERM)
;;;                              (SUBST FLG S TERM))))


;; But, the first hypothesis of this lemma needs special handling
;; because of free variables in the relevant rewrite rules.  The lemma
;; DISJOINT-SUBSETP-HACK was proved at this point, and appears now in
;; sets.events.

;; And finally, we finish.  During polishing I suddenly needed the
;; lemma SUBSETP-INTERSECTION-MONOTONE-2, which is now included in
;; "sets.events", and which in turn suggested
;; SUBSETP-INTERSECTION-COMMUTER there.

;; << 56 >>
(prove-lemma main-hyps-relieved-6-rest-generalization (rewrite)
  (let ((s1 (restrict s domain-1))
        (s2 (apply-to-subst (nullify-subst sg)
                            (co-restrict s domain-1))))
    (implies (and (var-substp sg)
                  (var-substp s)
                  (subsetp (domain s) new-free)
                  (termp f p)
                  (theorem-list (subst f s p))
                  (disjoint (domain sg)
                            (all-vars f (goals-disjoint-from-vars
                                         p new-free domain-1)))
                  (all-vars-disjoint-or-subsetp p new-free domain-1))
```

```
             (theorem-list (subst f (append s1 s2) p)))))

;; Now to clean up the goals that have been pushed above:

;; << 57 >>
(prove-lemma main-hyps-relieved-6-rest (rewrite)
  (let ((g (caar state))
        (p (cdar state))
        (free (cdr state))
        (s (witnessing-instantiation (generalize sg state))))
    (let ((new-g (subst t (invert sg) g)))
      (let ((domain-1
             (gen-closure (cons new-g p) free (all-vars t new-g))))
        (let ((s1 (restrict s domain-1))
              (s2 (apply-to-subst (nullify-subst sg)
                                  (co-restrict s domain-1))))
          (implies (and (generalize-okp sg state)
                        (valid-state (generalize sg state)))
                   (theorem-list (subst f (append s1 s2) p)))))))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable theorem-list subst
           car-generalize ;; so that we can get at p from (car state)
           ;; relieving hyps of main-hyps-relieved-6-rest-generalization:
           main-hyps-relieved-6-rest-lemma-1 main-hyps-relieved-6-rest-lemma-2
           ;; to relieve the (termp f p) hypothesis
           ;; in main-hyps-relieved-6-rest-generalization:
           statep termp-list-cons
           generalize-okp valid-state-opener
           main-hyps-relieved-6-rest-generalization)))

;; << 58 >>
(prove-lemma main-hyps-relieved-6
 (rewrite)
 (let ((g (caar state))
       (p (cdar state))
       (free (cdr state))
       (s (witnessing-instantiation (generalize sg state))))
   (let ((new-g (subst t (invert sg) g)))
     (let ((domain-1
            (gen-closure (cons new-g p) free (all-vars t new-g))))
       (let ((s1 (restrict s domain-1))
             (s2 (apply-to-subst (nullify-subst sg)
                                 (co-restrict s domain-1))))
         (implies (and (generalize-okp sg state)
                       (valid-state (generalize sg state)))
                  (theorem-list (subst f (append s1 s2)
                                       (cons new-g p)))))))))
 ((disable-theory t)
  (enable-theory ground-zero)
  (enable main-hyps-relieved-6-first main-hyps-relieved-6-rest subst
          theorem-list)))

;; << 59 >>
(prove-lemma main-hyps-relieved
 (rewrite)
 (let ((g (caar state))
       (p (cdar state))
       (free (cdr state))
       (s (witnessing-instantiation (generalize sg state))))
   (let ((new-g (subst t (invert sg) g)))
     (let ((domain-1
            (gen-closure (cons new-g p) free (all-vars t new-g))))
       (let ((s1 (restrict s domain-1))
             (s2 (apply-to-subst (nullify-subst sg)
                                 (co-restrict s domain-1))))
         (implies (and (generalize-okp sg state)
                       (valid-state (generalize sg state)))
                  (main-hyps s1 s2 sg g p))))))
```

```
 ((disable-theory t)
  (enable-theory ground-zero)
  (enable main-hyps main-hyps-relieved-1 main-hyps-relieved-2
          main-hyps-relieved-3 main-hyps-relieved-4 main-hyps-relieved-5
          main-hyps-relieved-6)))

;; << 60 >>
(prove-lemma main-theorem-1-case-4
             (rewrite)
             (implies (and (generalize-okp sg state)
                           (valid-state (generalize sg state)))
                      (theorem-list (subst f
                                           (gen-inst sg state)
                                           (car state))))
             ((disable-theory t)
              (enable-theory ground-zero)
              (enable gen-inst main-hyps-suffice generalize-okp
                      main-hyps-relieved)))

;; << 61 >>
(prove-lemma main-theorem-1 (rewrite)
  (let ((wit (gen-inst sg state)))
    (implies (and (generalize-okp sg state)
                  (valid-state (generalize sg state)))
             (and (statep state)
                  (var-substp wit)
                  (subsetp (domain wit) (cdr state))
                  (theorem-list (subst f wit (car state))))))
  ((disable-theory t)
   (enable-theory ground-zero)
   (enable main-theorem-1-case-1 main-theorem-1-case-2
           main-theorem-1-case-3 main-theorem-1-case-4)))

;; << 62 >>
(prove-lemma generalize-is-correct
          (rewrite)
          (implies (and (generalize-okp sg state)
                        (valid-state (generalize sg state)))
                   (valid-state state))
          ((disable-theory t)
           (enable-theory ground-zero)
           (enable main-theorem-1)
           (use (valid-state (witnessing-instantiation (gen-inst sg state))))))

))
```

References

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

[2] R.S. Boyer and J S. Moore.
Metafunctions: proving them correct and using them efficiently as new proof procedures.
The Correctness Problem in Computer Science.
Academic Press, 1981, pages 103-185.

[3] R. S. Boyer and J S. Moore.
A Computational Logic Handbook.
Academic Press, Boston, 1988.

[4] R. S. Boyer, D. M. Goldschlag, M. Kaufmann, and J S. Moore.
Functional Instantiation in First Order Logic, Report 44.
Technical Report, Computational Logic, 1717 W. 6th St., Austin, Texas, 78703, U.S.A., 1989.
To appear in the proceedings of the 1989 Workshop on Programming Logic, Programming Methodology Group, University of Goteborg.

[5] R.L. Constable, et al.
Implementing Mathematics with the Nuprl Proof Development System.
Prentice Hall, 1986.

[6] M. Davis and J. T. Schwartz.
Metamatehmatical extensibility for theorem verifiers and proof-checkers.
Computers and Mathematics with Applications 5:217-230, 1979.

[7] M. J. Gordon, A. J. Milner, and C. P. Wadsworth.
Edinburgh LCF.
Springer-Verlag, New York, 1979.

[8] M. Gordon.
HOL: A Proof Generating System for Higher-Order Logic.
Technical Report 103, University of Cambridge, Computer Laboratory, 1987.

[9] D. J. Howe.
Computational metatheory in Nuprl.
In *9th International Conference on Automated Deduction*, pages 238-257. Springer-Verlag, 1988.

[10] Matt Kaufmann.
A user's manual for an interactive enhancement to the Boyer-Moore Theorem Prover.
Technical Report 19, Computational Logic, Inc., Austin, Texas, May, 1988.

[11] Matt Kaufmann.
Addition of free variables to an interactive enhancement of the Boyer-Moore Theorem Prover.
Technical Report 42, Computational Logic, Inc., Austin, Texas, May, 1989.

[12] Matt Kaufmann.
DEFN-SK: An extension of the Boyer-Moore Theorem Prover to handle first-order quantifiers.
Technical Report 43, Computational Logic, Inc., Austin, Texas, June, 1989.

[13] Todd B. Knoblock.
A formal metalanguage for NuPrl.
to appear.

[14] T. B. Knoblock and R. L. Constable.
Formalized metareasoning in type theory.
In *Proceedings of the First Annual Symposium on Logic in Computer Science*. IEEE, 1976.

[15] A. Quaife.
Automated proofs of Loeb's Theorem and Goedel's two imcompleteness theorems.
Journal of Automated Reasoning 4:219-231, 1988.

[16] N. Shankar.
Towards Mechanical Metamathematics.
Journal of Automated Reasoning 1(1), 1985.

[17] Guy L. Steele Jr.
Common Lisp: The Language.
Digital Press, 1984.

[18] R. W. Weyhrauch.
Prolegomena to a theory of formal reasoning.
Artificial Intelligence 13:133-170, 1980.

Table of Contents