DEFN-SK: An Extension of the Boyer-Moore Theorem Prover to Handle First-Order Quantifiers ***DRAFT***

Matt Kaufmann

Technical Report #43 May, 1989

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 successful use of the Boyer-Moore Theorem Prover to proof-check a diverse variety of theorems is well-documented in Boyer and Moore's book [1]. Nevertheless, there are occasions when their quantifier-free logic is awkward or inadequate because of its lack of explicit quantification. Actually, the current "NQTHM" logic does have a version of bounded quantification, but that notion involves a somewhat tricky concept of evaluation, similar in flavor to the Lisp EVAL construct. What's more, that logic is considerably weaker than full first-order logic.[1]

The DEFN-SK event is simply an interface from first-order logic into the Boyer-Moore logic. It introduces a new function symbol which abbreviates a first-order formula. For example, the event

```
(defn-sk arb-lg-p (n)
  (forall x (exists y
    (and (lessp x y)
         (p n y)))))
```

causes the prover to add the following two rewrite rules to its database (each of them called ARB-LG-P, by the way):

```
(IMPLIES (AND (LESSP (X N) Y)
              (P N Y))
         (ARB-LG-P N))

(IMPLIES (NOT (AND (LESSP X (Y N X))
                   (P N (Y N X))))
         (NOT (ARB-LG-P N)))
```

Actually, the second of these rules may be easier to read in the contrapositive,

```
(IMPLIES (ARB-LG-P N)
         (AND (LESSP X (Y N X))
              (P N (Y N X)))) .
```

However, this latter version will not necessarily be at all usable as a rewrite rule in general; consider for example the situation when `(p n y)` is replaced by an `IF` term in the original DEFN-SK formula.

The following section (Section 2) carefully documents the new DEFN-SK event. That is followed in Section 3 by a brief discussion of a notion of *normalization* that we employ to allow simple first-order reasoning during processing of a DEFN-SK event. Soundness is addressed in Section 4. We conclude in Section 5 with brief descriptions of three successful experiments employing DEFN-SK.

[1]For logicians, let us point out that all integer functions definable in the NQTHM logic are easily seen to be recursive in the halting problem, hence definable with at most two alternations of quantifiers.

There are also several appendices for the sections that we feel may be of less general interest. The first appendix documents a macro DEFN-FO built on top of DEFN-SK which does some first-order simplification (normalization). The second appendix lays out our Skolemization algorithm and proves its correctness. We follow this with Appendix C, which extends the soundness proof in [2]. The remaining appendices contain the lists of events summarized in Section 5.

Acknowledgements. I'd like to thank my colleagues at Computational Logic for useful conversations and suggestions during the course of this work. I especially thank Bob Boyer for useful suggestions for the implementation and J Moore for a code review.

2. Documentation

This note introduces a new NQTHM event, DEFN-SK, together with a corresponding extension to the Boyer-Moore Computational Logic. The "SK" is in honor of Thoralf Skolem, for whom the various Skolemization algorithms are named. All of these algorithms have the property that they take a first-order formula and return an "appropriate" term (without quantifiers). This notion of "appropriate" is a rather slippery one, and we will treat it with care. The important thing for us here is that Skolemization gives us a way to "interpret" first-order formulas in a slight extension of the quantifier-free "computational logic" of Boyer and Moore [1], with minimal changes to the NQTHM theorem-prover.

Of course, we want the output of our Skolemization procedure to provide terms which are helpful when we attempt to reason about them. It might be tempting then to write a Skolemizer which is "smart" in some ways. However, a basic design decision here is to provide a rather basic Skolemization procedure, but to allow a mildly "smart" *normalization* process. Then "smart" Skolemizations are obtained by replacing the input formula by a "better" formula[2] which the normalizer shows is equivalent to the input formula, and then this "better" formula is the one that is (routinely) Skolemized. Nevertheless, we want our Skolemizer to avoid some obvious pitfalls; such considerations are discussed in Appendix B, together with the actual Skolemization algorithm.

For this section, however, we confine ourselves to specifying the following: the official and extended

[2]The FORMULA directive (hint) described below is the interface, and the DEFN-FO facility described in Appendix A can be used to generate this directive.

syntax for formulas (Subsection 2.1), Skolemization's main properties (2.2), normalization (2.3)[3], and the DEFN-SK event (2.4). We close in Subsection 2.5 with some remarks.

2.1 Formulas

Fix a history. A (first-order) *formula* is either a Boyer-Moore term (with respect to the given history), a *propositional combination of* formulas, or a *quantification of* a formula. A *propositional combination of* formulas is any expression of the form **`(NOT x)`**, **`(AND x y)`**, **`(OR x y)`**, **`(IMPLIES x y)`**, or **`(IF t x y)`**, where **`x`** and **`y`** are formulas and **`u`** is a term. A *quantification of* a formula is any expression of the form **`(FORALL v x)`** or **`(EXISTS v x)`**, where **`v`** is a variable and **`x`** is a formula.

In analogy to (and extending) the *extended syntax for terms* as described in [1], i.e. the syntax that the user may type in, we extend the syntax for formulas to an *extended syntax for formulas* as follows. First, any term in the extended syntax for terms is in the extended syntax for formulas. Next, it is a permissible abbreviation to give **`AND`** and **`OR`** more than two arguments. It is also acceptable to write **(Q (v$_1$ ... v$_n$) x)**, where **`Q`** is **`FORALL`** or **`EXISTS`** and *n* is a non-negative integer, as an abbreviation for **(Q v$_1$ (Q v$_2$ ... (Q v$_n$ x) ...))**.[4] Finally, **`(IFF x y)`** abbreviates **`(AND (IMPLIES x y) (IMPLIES y x))`** and if **`u`** is not a term then **`(IF u x y)`** abbreviates **`(AND (IMPLIES u x) (IMPLIES (NOT u) y))`**.

We will feel free to use familiar notions about formulas, such as the notion of *free variable*. Such notions are explained in any logic textbook; see for example [3].

2.2 Skolemization in Brief

For now let us discuss Skolemization in an abstract framework. We defer to Appendix B the presentation of the particular Skolemization algorithm that we use. Let *F* be a set of function symbols and let ϕ be a first-order sentence (i.e. formula without free variables), all of whose function symbols are contained in *F*. Let ψ be a quantifier-free formula (i.e. term) with universal closure[5] ψ'. Then ψ is said to be a *Skolemization of* ϕ *with respect to F* if for any set *S* of first-order sentences, all of whose function symbols are included in *F*, (1)

[3]See Section 3 for our normalization *algorithm*

[4]In particular, **`(Q () x)`** abbreviates **`x`**.

[5]Recall that the *universal closure* of a formula ψ is a formula **(FORALL (v$_1$ ... v$_n$) ψ)** where **(v$_1$ ... v$_n$)** is an enumeration of the free variables of ψ.

the extension of *S* by the sentence **`[`**$\phi \rightarrow \psi'$**`]`** is a conservative extension[6] of *S*, and (2) the sentence **`[`**$\psi' \rightarrow \phi$**`]`** is a theorem of *S*. If we don't mention *F*, then it is taken to be the union of the set of function symbols in ϕ together with those of the current history. The function symbols of ψ which do not occur in *F* are called the *Skolem functions* of ψ. The variables occurring in ψ which are not free in ϕ are called the *Skolem variables introduced into* ψ.

Consider for example the formula **`(FORALL x (EXISTS y (p x y)))`**. It's easy to see that **`(p x (f x))`** is a Skolemization of this formula in the sense above, as long as **`f`** does not occur in that formula or in the current history.

We need one more property of Skolemization (a rather obscure one) for the proof in Appendix C. That property is called the "Skolemization Extension Lemma" in that appendix, and since we expect that most readers will not be interested in reading that proof, we say no more here.

2.3 Specification of Normalization

We also present here a notion of *normalization* in an abstract framework. We take *normalization* to be any function on the set of first-order formulas which has the property that every formula is logically equivalent to its normalization. We defer to Section 3 the description of our particular notion of normalization. For now let us simply note that we push in quantifiers wherever possible and drop irrelevant quantifiers.

For example, the normalization of **`(EXISTS X (FORALL Y (IMPLIES (P X Z) (P Y Z))))`** is **`(IMPLIES (FORALL X (P X Z)) (FORALL Y (P Y Z)))`**. The idea is that one imagines first pushing the quantifier on **`X`** past the implication, which is legal since **`Y`** is not free in the first argument of **`IMPLIES`**, and then pushing the quantifier **`(FORALL X)`** past the resulting implication, which is legal since **`X`** is not free in the conclusion (second argument) of that implication. Notice that it's much easier to see that the resulting ("normalized") sentence is a theorem than that the original sentence is a theorem. That's why we've implemented a notion of normalization. Note that in fact a smart enough procedure, such as the one in [4], would immediately recognize the result as a theorem. We choose however to keep our procedure, as described in Section 3 below, very simple.

[6]See Section 4 for a review of the notion of conservative extension.

2.4 New event: Defining first-order notions

(formal section)

Let ϕ be a first-order formula with respect to a given history *h* whose free variables are contained among the set **{v_1, ..., v_n}**, where **v_1**, ..., **v_n** are distinct variables. Let **g** be a function symbol of arity **n** which is new for *h*. Suppose that **sk** is a Skolemization of the first-order sentence

(FORALL (v_1 ... v_n) (IFF (g v_1 ... v_n) ϕ))

with respect to a set of function symbols which contains **g** together with all function symbols of *h*. Then it is permitted to add **sk** as an axiom to extend *h*.

(manual section)

```
General Form:  (DEFN-SK name args form &optional directives)
Example Form:  (DEFN-SK BAR (Z)
                 (EXISTS X (FORALL Y (IMPLIES (P X Z) (P Y Z))))
                 ((FORMULA (IMPLIES (FORALL X (P X Z))
                                    (FORALL Y (P Y Z))))
                  (PREFIX SK-)))
```

Here **form** is a first-order formula (in the syntax described above), as is the argument to the FORMULA directive (if supplied), and **args** is a list of distinct variables which includes all free variables of **form**. If the FORMULA directive is supplied, then that formula and **form** must have normalizations (as specified in the subsection above) which are the same, up to renaming of bound variables. Let **args** be **(arg_1 ... arg_n)**, let **u** be **(name arg_1 ... arg_n)**, and let **body** be **form** unless a FORMULA directive is supplied, in which case **body** is that formula. Consider the following first-order sentence:

(*)   (FORALL (arg_1 arg_2 ... arg_n) (IFF u body))

where **args** is **(arg_1 ... arg_n)**. The effect of this event is to add as an axiom a Skolemization of this sentence, with respect to the result of adding **name** to the set of function symbols of the current history. In fact this axiom is added as a pair of rewrite rules, one having conclusion of **u** and the other having conclusion of **(NOT u)**. Moreover, the entire axiom becomes the formula associated with **name**[7], so that it may be referred to in, for example, a USE hint of a PROVE-LEMMA. If a PREFIX hint is supplied, then the indicated prefix is tacked on to the front of the name of each Skolem function generated. A similar SUFFIX hint is allowed; in fact, PREFIX and SUFFIX hints may be used together. Finally, **name** is given a Boolean type set.

[7]i.e., the result of evaluating **(FORMULA-OF 'name)**

2.5 Further remarks

Consider the example above. In that case, the formula associated to BAR for the purpose of USE hints is the conjunction of the following two formulas, each of which is made into a rewrite rule named BAR. (Notice that the second of these is written as the contrapositive of what one might have otherwise expected, in order to put it in a form guaranteed appropriate for a rewrite rule.)

```
(IMPLIES (IMPLIES (P X Z) (P (SK-Y Z) Z))
         (BAR Z))

(IMPLIES (NOT (IMPLIES (P (SK-X Z) Z) (P Y Z)))
         (NOT (BAR Z)))
```

This conjunction is what is supplied when BAR is given in a USE hint to PROVE-LEMMA.

We claim that it is a theorem of the new history that the application of **`name`** to **`args`** is equivalent to **`form`**. That is, the formula (*) in the "formal section" above is a theorem of the new history. Why? Because the new axiom provably yields (*) by the specification of Skolemization given in Subsection 2.2, and since this new axiom is added to the new history, (*) follows.

One rather subtle point may be worth mentioning. The last point in the "manual section" above is that **`name`** is given a boolean type set. This reflects the fact that if one uses a term as a formula, then one is "coercing" it to a Boolean value in the sense that it is viewed there as denoting a member of the set **`{T, F}`**. More precisely, the point here is that function symbols defined by DEFN-SK are always given Boolean type sets. For example, suppose we have

```
(DEFN-SK FOO (X Y) (PLUS X Y)) .
```

Then we can prove

```
(EQUAL (FOO X Y) T)
```

because PLUS never returns **`F`**.

3. Normalization

We have already specified *normalization* as any function which returns a first-order formula that is logically equivalent to the given one. In [4] de Champeaux gives a rather elaborate "miniscoping" procedure for first-order simplification. However, we confine ourselves to the simplest parts of this procedure. We omit its proof of correctness (i.e. that it meets the above specification), which is routine.

Here is the algorithm. Let us define the *negation* of a formula ϕ to be **`(NOT`** ϕ**`)`** unless ϕ is of the form

(NOT ψ**)**, in which case it is ψ. We use this notion to maintain the invariant that there are no double negations in the resulting formula. Let ϕ be an arbitrary formula; here then is how we normalize ϕ.

- If ϕ is **(NOT** ψ**)**, then return the negation of the result of normalizing ψ.
- If ϕ is any other propositional combination of formulas, then return the application of the same propositional operator to the result of normalizing all the arguments of that operator.
- Otherwise, if ϕ is not the application of a quantifier to a formula, return ϕ.
- Otherwise ϕ is of the form **(Q x** ψ**)** where **Q** is **FORALL** or **EXISTS**, and we return the result of *pushing* **Q** *into* the normalization ρ of ψ, where:

We define the notion of *pushing* a quantifier **(Q x)** *into* a formula ρ as follows. We will write **Q*** for the *dual* of **Q**, i.e. **FORALL** if **Q** is **EXISTS** and **EXISTS** if **Q** is **FORALL**. First we need an auxiliary notion. A tuple **(Q,v,**ψ**)**, where **Q** is **FORALL** or **EXISTS**, **v** is a variable, and ψ is a formula, is *distributable* if

- ψ is of the form **(IF u** ψ_1 ψ_2**)** where **v** does not occur in the term **u**; or
- ψ is of the form **(AND** ψ_1 ψ_2**)** and **Q** is **FORALL**; or
- ψ is of the form **(OR** ψ_1 ψ_2**)** or **(IMPLIES** ψ_1 ψ_2**)** and **Q** is **EXISTS**; or
- ψ is of the form **(C** ψ_1 ψ_2**)**, **C** $\in$ {**AND**, **OR**, **IMPLIES**}, and it is not the case that **v** is free in both ψ_1 and ψ_2.

Finally, then, here is the definition of the operation of *pushing* a quantifier **(Q x)** *into* a formula ρ.

- If the variable **x** is not free in ρ, return ρ.
- Otherwise, if ρ is of the form **(NOT** γ**)**, there are two subcases. If **(Q*,x,**γ**)** is distributable (see definition above), then return the negation of the result of pushing **(Q* x)** into γ. Otherwise return **(Q x** ρ**)**.
- Otherwise, if **(Q,x,**ρ**)** is not distributable, then return **(Q x** ρ**)**.
- Otherwise, if ρ is of the form **(IF u** ρ_1 ρ_2**)** then let ρ_i' be the result of pushing **(Q x)** into ρ_i (for **i** = 1, 2), and return **(IF u** ρ_1' ρ_2'**)**.
- Otherwise, if ρ is of the form **(IMPLIES** ρ_1 ρ_2**)** then let ρ_1' be the result of pushing **(Q* x)** into ρ_1 and let ρ_2' be the result of pushing **(Q x)** into ρ_2, and return **(IMPLIES** ρ_1' ρ_2'**)**.
- Otherwise ρ must be of the form **(C** ρ_1 ρ_2**)** where **C** is **AND** or **OR**. Then let ρ_i' be the result of pushing **(Q x)** into ρ_i (for **i** = 1, 2), and return **(C** ρ_1' ρ_2'**)**.

Implementation note. In our implementation we have a translator which takes expressions typed by the user (in the extended syntax for formulas) and returns a parse of that formula in the "official syntax" (see Subsection 2.1). This translator has the property that it puts in formula nodes wherever possible. For example, suppose that $\mathbf{t_1}$ and $\mathbf{t_2}$ are terms and consider the expression **(AND** $\mathbf{t_1}$ $\mathbf{t_2}$**)**. This expression may be translated to (parsed as) either a formula or a term, and our translator would parse this as a formula. We do things this way so that the normalization procedure has the opportunity to dive into such an **AND** expression. As an optimization for the Skolemizer, since the Skolemization of a quantifier-free formula is itself, we actually re-parse such a formula as a term before going into Skolemization. We actually do this re-parse even before comparing normalized formulas (from the body and the FORMULA hint) for equality (modulo renaming of bound variables).

4. Soundness

In this section we state and prove theorems which demonstrate the soundness of our approach. It is helpful to recall that DEFN-SK extends a history by adding a **`term`** in the (quantifier-free) logic. However, it was shown above that the first-order formula (*) implicit in that DEFN-SK event is indeed a theorem of the resulting history (when that history is viewed as a set of first-order formulas).

First, recall the following standard definition: a set $\mathtt{S}_1$ of first-order sentences which contains a set $\mathtt{S}_2$ is a *conservative extension* of $\mathtt{S}_2$ if any theorem provable from $\mathtt{S}_1$ which is in the language (alphabet) of $\mathtt{S}_2$ is in fact already a theorem of $\mathtt{S}_2$.

The theorem below shows that adding DEFN-SK gives conservative extensions. It follows that consistency is preserved by DEFN-SK, since conservative extensions preserve consistency: using the notation of the definition above, if $\mathtt{S}_1$ is a conservative extension of a consistent set $\mathtt{S}_2$ then since **`F`** (*false*) is not a theorem of $\mathtt{S}_2$, **`F`** is not a theorem of $\mathtt{S}_1$.

SOUNDNESS THEOREM FOR DEFN-SK[8]. Suppose we imbed the Boyer-Moore logic into a traditional first-order logic, such as that of [3], turning the induction principle into a collection of axioms, admitting existential quantifiers and the existential-quantifier introduction-rule. Then a DEFN-SK event, as defined above, results in a conservative extension of the previous theory.

Proof. Suppose we extend a history *h* to a new history *h'* by adding a Skolemization of the first-order sentence **`(FORALL (v1 ...  vn) (IFF (g v1 ... vn) body))`**, where all free variables of **`body`** are among **`{v1, ..., vn}`** and **`g`** is a new function symbol (and the Skolemization is with respect to some set containing **`g`** along with all function symbols of *h*). Let h_1 be the history obtained by adding the above first-order sentence to *h*, and let h_2 be the union of h_1 and *h'*. Then h_1 is a conservative extension of *h* since it's simply a definitional extension (cf. [3]). And h_2 is a conservative extension of h_1 by our (abstract) definition of Skolemization. So for any first-order sentence **`A`** in the language of *h*, if *h'* **`|- A`** then h_2 **`|- A`** (since h_2 extends *h'*), hence h_1 **`|- A`** (by conservativity of h_2 over h_1), hence *h* **`|- A`** (by conservativity of h_1 over *h*). -|

[8]This is analogous (even to its wording!) to a corresponding note about CONSTRAIN in [2].

The paper [2] contains an argument which shows the correctness of the implementation of the event FUNCTIONALLY-INSTANTIATE. However, that argument assumes an underlying logic which does not have DEFN-SK. In Appendix C we provide the main lemma required to extend the arguments in [2] to the case that the existing "starting point" logic does include DEFN-SK.

5. Examples

The purpose of this section is to demonstrate that our simple DEFN-SK interface from first-order logic into the Boyer-Moore logic enables one to mechanically proof-check interesting theorems. We treat three separate examples here. Complete proof scripts are in the final three appendices.

These proofs make heavy use of the induction capabilities of the Boyer-Moore prover. The main idea of the DEFN-SK approach is to retain the current prover's strengths while allowing first-order reasoning.

All of these examples introduce axioms using the CONSTRAIN mechanism reported in [2]. Appendix C extends the proof in that paper to show conservativity of CONSTRAIN events even in the presence of DEFN-SK events. The reader not familiar with [2] can view CONSTRAIN simply as a consistent way of adding axioms. The first subsection below says a little more about CONSTRAIN.

All three examples made heavy use of the "proof-checker" enhancement (PC-NQTHM) of the Boyer-Moore prover, as reported first in [5] and then extended in [6] to implement a notion of free variables. The proofs especially used the macro command SK* documented in [6] to eliminate notions defined by DEFN-SK by automatically applying the Skolem axioms and the BACKCHAIN macro command to provide appropriate instantiations for the free variables introduced by SK*. However, in the first two examples we were able to extract, usually without much difficulty, the requisite applications of the Skolem axioms so that we could create USE hints for the PROVE-LEMMA events and eliminate the need for the proof-checker on the final events lists. Thus, the first two examples replay in the unadorned Boyer-Moore prover as extended by the functional variables work (especially CONSTRAIN) reported in [2] together with DEFN-SK, but *not* extended by the proof-checker. We hope to implement something akin to the proof-checker's treatment of free variables, as used in these proofs, in the Boyer-Moore prover someday; then the aforementioned USE hints should not, we hope, be necessary.

5.1 Koenig's Tree Lemma.

Koenig's Tree Lemma states that every infinite, rooted, finitely-branching tree has an infinite branch. Rather than explain these terms here, we will simply state the axiom in our formalization of Koenig's Tree Lemma after providing some intuition. The proof (sequence of events) which is discussed here is included in Appendix D.

Intuitively, we identify a *node* of such a tree with the sequence of positive integers which represents the path from the root of the tree to that node. For example the node `r321` pictured below corresponds to the sequence `(3 2 1)`, since it's the third child of the second child of the first child of the root `r`.

```
r121  r221 r321   r112
    \  |  /      /
r11   r21     r12
  \  /        /
   r1       r2
     \     /
        r
```

In particular, the empty sequence `NIL` represents the root of the tree. And the less-than relation is represented by the notion of "terminal subsequence", which (in the usual style of the Boyer-Moore logic) we define recursively:

```
(defn subseq (s1 s2)
  ;; s1 is a terminal subsequence (nthcdr) of s2
  (if (equal s1 s2)
      t
    (if (nlistp s2)
        f
      (subseq s1 (cdr s2)))))
```

Here then is the axiom which introduces an appropriate tree, by way of a predicate `node-p`. The predicate `subseq` is defined so that `(subseq s1 s2)` holds for sequences `s1` and `s2` when `s2` consists exactly of the last `n` elements of `s1`, `n` being the length of `s2`. We enforce that the tree is finitely-branching by introducing a function `succard` ("Successors Cardinality") which, for a given node in the tree, returns the number of immediate successors of that node. The CONSTRAIN mechanism guarantees consistency of the axiom by requiring us to give example functions which make the axiom true. In this case the "witness alist"

```
((node-p all-ones)
 (succard (lambda (s) 1))
 (s-n ones))
```

suggests that if `(NODE-P S)` holds exactly when `S` is a sequence of ones, `SUCCARD` always returns 1 (i.e. every node has exactly one successor), and `S-N` is the function `ONES` where `(ONES N)` is a sequence of `N` ones, then the axiom holds. In fact the theorem-prover checks this.

```
(constrain koenig-intro (rewrite)
  ;; Introduce the predicate node-p for the nodes of (i.e. finite paths
  ;; through) the tree.  That is, node-p recognizes the legal paths.
  (and
   ;; nil is the root
   (node-p nil)
   ;; node-p is a predicate
   (or (truep (node-p s))
       (falsep (node-p s)))
   ;; the successors of s are determined by succard (Successors Cardinality)
   (implies (node-p s)
            (equal (node-p (cons n s))
                   ;; n is in {1, ..., (succard s)}
                   (and (lessp 0 n)
                        (not (lessp (succard s) n)))))
   (implies (and (node-p s1)
                 (subseq s s1))
            (node-p s))
   ;; we stipulate that the tree is infinite by saying that s-n is a one-one enumeration of nodes
   (node-p (s-n n))
   (implies (and (numberp i) (numberp j) (not (equal i j)))
            (not (equal (s-n i) (s-n j))))
   ;; nodes are proper
   (implies (not (plistp s))
            (not (node-p s))))
  ((node-p all-ones)
   (succard (lambda (s) 1))
   (s-n ones))
  ((disable subseq)))
```

Our approach was to follow the natural proof in which one builds a branch through the tree "nonconstructively" as follows. Starting at the root, we maintain the invariant that the top node on the branch constructed so far has infinitely many successors (not *immediate* successors, of course!) in the tree. So given the current top node, we extend the branch by choosing an immediate successor which has infinitely many successors -- note that if each immediate successor had only finitely many successors, then the given top node would have only finitely many successors (since the tree is finitely-branching), which would violate the invariant.[9]

Here is the DEFN-SK event which formalizes the notion of "infinitely many successors":

```
(defn-sk inf (s)
  ;; says that s has arbitrarily high successors
  (forall big-h (exists big-s
    (and (subseq s big-s)
         (node-p big-s)
         (lessp big-h (length big-s))))))
```

This adds the Skolem axiom

[9]Comment for logicians. It's well known that there are infinite, recursive, finitely-branching, rooted trees without infinite recursive branches, and in fact this result relativizes to any oracle. This strongly suggests that there is no way to prove Koenig's Lemma in the Boyer-Moore logic.

```
(AND (IMPLIES (AND (SUBSEQ S BIG-S)
                   (NODE-P BIG-S)
                   (LESSP (BIG-H S) (LENGTH BIG-S)))
              (INF S))
     (IMPLIES (NOT (AND (SUBSEQ S (BIG-S BIG-H S))
                        (NODE-P (BIG-S BIG-H S))
                        (LESSP BIG-H
                               (LENGTH (BIG-S BIG-H S)))))
              (NOT (INF S))))
```

An earlier approach used a slightly different CONSTRAIN axiom, where we assumed that there was a function **`S-HEIGHT`** instead of **`S-N`**, where **`(S-HEIGHT N)`** was postulated to be a node of height **`N`**. After completing that proof, and cleaning it up with the help of J Moore, we went about the task of using the CONSTRAIN event shown above and then *defining* (with DEFN) a function **`S-HEIGHT`** which had the desired property. This turned out to be a substantial portion of the total effort. Those events are near the front of Appendix D and are clearly labeled.

Our methodology for reasoning about **`INF`** was to introduce events INF-SUFF and INF-NECC (see the appendix). We used the proof-checker [5] enhancement of the Boyer-Moore prover, as extended to support free variables [6], to deal with the free variables introduced by backchaining with the Skolem axioms. A number of events in the appendices have a comment like ";; done with help of proof-checker" to indicate that the first proof was found using the proof-checker, including macro-commands to help with unification and backchaining. However, the final list of events in Appendix D has been successfully run in the extension of NQTHM by CONSTRAIN [2] and DEFN-SK. The final theorem is as follows:

```
(implies (numberp n)
         (and (node-p (k n))
              (implies (not (lessp j i))
                       (subseq (k i) (k j)))
              (equal (length (k n)) n)))
```

Let us make one last technical point, referring the reader to the events in Appendix D for details (since of course the events speak for themselves!). A key to the proof is the following definition.

```
(defn all-big-h (s n)
  (if (zerop n)
      (add1 (length s))
    (plus (big-h (cons n s))
          (all-big-h s (sub1 n)))))
```

This function returns a number that is bigger than **`(big-h (cons s i))`** for all positive **`i`** $\leq$ **`n`** and is also bigger than length of **`s`**. Here, **`BIG-H`** comes from the definition (DEFN-SK) of **`INF`** above; roughly, **`BIG-H`** is a function which has the property that if there is a node of length at least **`(BIG-H S)`** which extends a given node **`S`**, then there are nodes of arbitrarily large height extending **`S`**. When we are looking to extend a given node **S** to a branch as in the proof outline above, the idea is to use the **(INF S)** hypothesis (invariant) to find an extension **S1** of **S** of length at least **(ALL-BIG-H S)**. We may then note that **S1** extends some immediate successor **(NEXT S (SUCCARD S))** of **S**, and by definition of **ALL-BIG-H** this node is high enough in the tree to guarantee **(INF (NEXT S (SUCCARD S)))**.

5.2 Ramsey's Theorem.

Recall that Ramsey's Theorem for exponent 2 says the following. Let **P** be a partition of the 2-element subsets of a given infinite set **A** into a finite number of pieces, say **n** pieces. (One can visualize here a space of nodes where each pair of nodes is connected by an edge of one of **n** given colors.) Then there is an infinite subset **H** of **A** such that all pairs from **H** lie in the same piece of the partition. (In terms of our graph picture, all edges between pairs of nodes from **H** are of the same color.)

This theorem has a proof which is rather similar in flavor to the Koenig's lemma proof described in the immediately preceding subsection. For simplicity we assume that the partition is on pairs of natural numbers, and we call it **P-NUM**. However, we immediately define an auxiliary function **P** which conceptually makes sense for all pairs (by coercing them to numbers for **P-NUM**), and prove all our lemmas about **P-NUM**. (This circuitous route is partly an artifact of our original approach, which didn't make the restriction to numbers in the axiom.) Think of **BOUND** below as the number of "colors"; we are partitioning the pairs into the set {1, ..., **(bound)**}.

```
(constrain p-num-intro (rewrite)
  (implies (and (numberp x) (numberp y))
           (and (lessp 0 (p-num x y))
                (not (lessp (bound) (p-num x y)))
                (equal (p-num x y) (p-num y x))))
  ((p-num (lambda (x y) 1))
   (bound (lambda () 2))))

(disable p-num-intro)

(defn p (x y)
  (p-num (fix x) (fix y)))

(prove-lemma p-intro (rewrite)
  (and (lessp 0 (p x y))
       (not (lessp (bound) (p x y)))
       (equal (p x y) (p y x)))
  ((use (p-num-intro (x (fix x)) (y (fix y))))))
```

We follow a standard proof of this theorem, which goes as follows. One defines a notion of *prehomogeneous set*, which says that for numbers **i < j** in such a set, the value of the partition on the pair **{i,j}** should depend only on **i**. In order to formalize this notion we actually consider lists of pairs **<i,c>** whose first component **i** is the number and whose second component **c** is the intended color. The auxiliary function **PREHOM-SEQ-1** takes arguments **a** and **x**, where one may think of **x** as being a list of pairs **<i,c>** as above, each **i** being less than **a**, where this function checks that **c** is the right color for the pair **{i,a}**.

```
(defn prehom-seq-1 (a x)
  ;; x is a list of pairs (i . c), and this says that for each
  ;; such pair, p(i,a) <-> c.
  (if (listp x)
      (and (equal (p (caar x) a) (cdar x))
           (prehom-seq-1 a (cdr x)))
    t))

(defn prehom-seq (x)
  ;; a list of pairs in decreasing order, such that
  ;; if (j . ?) is before (i . c) (and hence presumably i < j),
  ;; then p(i,j) <-> c.
  (if (listp x)
      (if (listp (cdr x))
          (and (lessp (car (cadr x)) (car (car x)))
               (prehom-seq-1 (caar x) (cdr x))
               (prehom-seq (cdr x)))
        t)
    t))
```

Continuing with the proof: the idea now is to define an increasing sequence of natural numbers which forms a prehomogeneous sequence. Each member **`i`** of this sequence is "colored" in the sense that the value of the partition on any pair **`{i,j}`**, where **`j`** is also in the sequence and **`i < j`**, depends only on the color associated with **`i`**. But since there are only finitely many colors, one must appear infinitely often in this prehomogenous sequence. Then the subsequence corresponding to that color forms the desired homogeneous set.

However, in order to construct the desired prehomogeneous sequence we need a stronger invariant than prehomogeneity. That invariant is expressed by the first of two introductions of quantifiers in this proof, which says roughly that **`s`** has arbitrarily large extensions to a prehomogeneous sequence:

```
(defn-sk extensible (s)
  ;; s is a list of pairs (i . c), and extensible means that there
  ;; are infinitely many a for which prehom-seq-1(a,s) holds.
  (forall above
          (exists next
                  (and (lessp above next)
                       (prehom-seq-1 next s)))))
```

The associated Skolem axiom is:

```
(AND (IMPLIES (AND (LESSP (ABOVE S) NEXT)
                   (PREHOM-SEQ-1 NEXT S))
              (EXTENSIBLE S))
     (IMPLIES (NOT (AND (LESSP ABOVE (NEXT ABOVE S))
                        (PREHOM-SEQ-1 (NEXT ABOVE S) S)))
              (NOT (EXTENSIBLE S))))
```

We omit here the definition of **`(RAMSEY-SEQ N)`**, which is a strictly decreasing prehomogeneous sequence of length **`N`**. It gives rise to the definition of the prehomogeneous sequence that we are looking for:

```
(defn ramsey (n)
  (car (car (ramsey-seq (ramsey-index n)))))
```

The other quantifier introduction is through an event which defines the notion of a "good color", i.e. one which is the color associated with arbitrarily large members of the range of **RAMSEY**:

```
(defn-sk good-color-p (c)
  ;; says that arbitrarily large elements of ramsey-seq agree with c
  (forall big
          (exists good-c-index
                  (and (lessp big good-c-index)
                       (equal c (cdr (car (ramsey-seq good-c-index))))))))
```

And here is its Skolem axiom:

```
(AND
 (IMPLIES (AND (LESSP (BIG C) GOOD-C-INDEX)
               (EQUAL C
                      (CDAR (RAMSEY-SEQ GOOD-C-INDEX))))
          (GOOD-COLOR-P C))
 (IMPLIES (NOT (AND (LESSP BIG (GOOD-C-INDEX BIG C))
                    (EQUAL C
                           (CDAR (RAMSEY-SEQ (GOOD-C-INDEX BIG C))))))
          (NOT (GOOD-COLOR-P C))))
```

We actually obtain a good color by defining a function **GOOD-C-INDEX-WIT** which is quite analogous to the function **ALL-BIG-H** in the preceding subsection (on Koenig's Lemma). The important lemma about this function is the following. In fact we're interested in the special case where the variable **bound** equals the constant **(BOUND)**. However, we state it this way so that the prover may prove it by induction on **bound**.

```
(prove-lemma lessp-big-good-c-index (rewrite)
  (implies (and (lessp 0 c)
                (not (lessp bound c)))
           (equal (lessp (big c) (good-c-index-wit bound))
                  t)))
```

This lemma, together with the Skolem axiom for **GOOD-COLOR-P** above, gives us a sufficiently large index **(GOOD-C-INDEX-WIT (BOUND))** to guarantee that the color **(COLOR)** defined by

```
(defn color ()
  (cdar (ramsey-seq (good-c-index-wit (bound)))))
```

appears infinitely often in the sequence represented by **RAMSEY**. The final theorems are as follows.

```
(prove-lemma ramsey-increasing nil
  (implies (lessp i j)
           (lessp (ramsey i) (ramsey j))))

(prove-lemma ramsey-seq-hom
             (rewrite)
             (implies (and (numberp i) (numberp j) (not (equal i j)))
                      (equal (p-num (ramsey i) (ramsey j))
                             (color)))
             ((use (ramsey-seq-hom-lessp)
                   (ramsey-seq-hom-lessp (i j) (j i)))))
```

5.3 Schroeder-Bernstein Theorem.

The Schroeder-Bernstein Theorem says that for any sets **`a`** and **`b`**, if there is a one-to-one function from **`a`** to **`b`** and also a one-to-one function from **`b`** to **`a`**, then there is a one-to-one correspondence between **`a`** and **`b`**. We followed the proof sketch given in Exercise 8 of Chapter 1 of [7]. The following axiom introduces our assumptions.

```
(constrain fa-and-fb-are-one-one (rewrite)
  (and
   ;; fa is one-to-one
   (implies (and (a x) (a y) (not (equal x y)))
            (not (equal (fa x) (fa y))))
   ;; fb is one-to-one
   (implies (and (b x) (b y) (not (equal x y)))
            (not (equal (fb x) (fb y))))
   ;; the image of fa on a is contained in b
   (implies (a x) (b (fa x)))
   ;; the image of fb on b is contained in a
   (implies (b x) (a (fb x)))
   (or (truep (a x)) (falsep (a x)))
   (or (truep (b x)) (falsep (b x))))
  ;; let fa and fb be the identity function and let a and b be the universe
  ((fa id)
   (fb id)
   (a (lambda (x) t)) (b (lambda (x) t))))
```

Here is the idea of the proof. Imagine that sets **`a`** and **`b`** and functions **`fa`** and **`fb`** are given as above:

```
                    --> fa ---->
 --------------------         --------------------
|                    |       |                    |
|                    |       |                    |
|                    |       |                    |
|                    |       |                    |
|                    |       |                    |
|                    |       |                    |
|                    |       |                    |
|                    |       |                    |
|                    |       |                    |
 --------------------         --------------------
                    <---- fb <--
         a                              b
```

Now imagine that we take the images of **`a`** and **`b`** under **`fa`** and **`fb`**, respectively. These images represent the inner boxes above. Let us write **`a0`** and **`b0`** to denote **`a`** and **`b`**, respectively, and let **`a1`** and **`b1`** be the respective images.

```
                        --> fa ---->
     --------------------          --------------------
    |                    |        |                    |
    |   --------------   |        |   --------------   |
    |  |              |  |        |  |              |  |
    |  |              |  |        |  |              |  |
    |  |              |  |        |  |              |  |
    |  |              |  |        |  |              |  |
    |  |              |  |        |  |              |  |
    |   ----- a1 -----   |        |   ----- b1 -----   |
    |                    |        |                    |
     -------- a0 --------          -------- b0 --------
                        <---- fb <--
```

We can of course repeat this picture as many times as we like, letting **a(i+1)** be the image of **b(i)** under **fb** and **b(i+1)** be the image of **a(i)** under **fa**. Let **a*** be the intersection of the sets **a(i)** and let ***** be the intersection of the sets **b(i)**.

```
                        --> fa ---->
     --------------------          --------------------
    |                    |        |                    |
    |   --------------   |        |   --------------   |
    |  |  ----------  |  |        |  |  ----------  |  |
    |  | |   ____   | |  |        |  | |   ____   | |  |
    |  | |  \_a*_/  | |  |        |  | |  \_b*_/  | |  |
    |  | |    ..    | |  |        |  | |    ..   | |  |
    |  |  --- a2 ---  |  |        |  |  --- b2 ---  |  |
    |   ----- a1 -----   |        |   ----- b1 -----   |
    |                    |        |                    |
     -------- a0 --------          -------- b0 --------
                        <---- fb <--
```

We might say that a point in **a(i)** is *circled by* **i**. Let us define the **A-LEVEL** of a point **x** in **a** to be that **i** such that **x** belongs to **a(i)** but not to **a(i+1)**, i.e. that **i** such that **x** is circled by **i** but not by **i+1**. Define analogous notions for **b**. Then a one-to-one correspondence may be constructed by mapping the even a-levels and **a*** into **b** *via* **fa** and by mapping the odd a-levels into **b** *via* the inverse of **fb**.

Here are events that represent the following ideas. They have been culled and edited from the event list in the final appendix in order to focus the reader's attention on those that are the most interesting. The Skolem axiom for each DEFN-SK event is placed in uppercase after that event for reference.

```
;;;;;;;;; DEFINITIONS AND "SKOLEM" CONSEQUENCES ;;;;;;;;;

(defn-sk in-fa-range (x)
  (exists fa-1 (and (a fa-1)
                    (equal (fa fa-1) x))))

Skolem axiom:
      (AND (IMPLIES (AND (A FA-1) (EQUAL (FA FA-1) X))
                    (IN-FA-RANGE X))
           (IMPLIES (NOT (AND (A (FA-1 X))
                              (EQUAL (FA (FA-1 X)) X)))
                    (NOT (IN-FA-RANGE X))))
```

```
(defn-sk in-fb-range (x)
  (exists fb-1 (and (b fb-1)
                    (equal (fb fb-1) x))))

Skolem axiom:
      (AND (IMPLIES (AND (B FB-1) (EQUAL (FB FB-1) X))
                    (IN-FB-RANGE X))
           (IMPLIES (NOT (AND (B (FB-1 X))
                              (EQUAL (FB (FB-1 X)) X)))
                    (NOT (IN-FB-RANGE X))))

(defn circled (flg x n)
  ;; If flg is 'a, returns t iff x is in a-n in the sense of Kunen's proof.
  ;; Similarly for b if flg is not 'a.
  (if (equal flg 'a)
      (if (zerop n)
          (a x)
        (and (in-fb-range x)
             (circled 'b (fb-1 x) (sub1 n))))
    (if (zerop n)
        (b x)
      (and (in-fa-range x)
           (circled 'a (fa-1 x) (sub1 n))))))

(defn-sk a-core (x)
  ;; introduced "inductively" so that the level will be tight
  (and (a x)
       (forall a-level
               (implies (and (numberp a-level)
                             (circled 'a x a-level))
                        (circled 'a x (add1 a-level))))))

Skolem axiom:
      (AND (IMPLIES (AND (A X)
                         (IMPLIES (AND (NUMBERP (A-LEVEL X))
                                       (CIRCLED 'A X (A-LEVEL X)))
                                  (CIRCLED 'A X (ADD1 (A-LEVEL X)))))
                    (A-CORE X))
           (IMPLIES (NOT (AND (A X)
                              (IMPLIES (AND (NUMBERP A-LEVEL)
                                            (CIRCLED 'A X A-LEVEL))
                                       (CIRCLED 'A X (ADD1 A-LEVEL)))))
                    (NOT (A-CORE X))))

LEMMA.  A-CORE-NECC
  ;; the conclusion implies the more obvious consequence of a-core
  (implies (not (circled 'a x n))
           (not (a-core x)))

LEMMA.  A-CORE-SUFF
  (implies (and (a x)
                (implies (and (numberp (a-level x))
                              (circled 'a x (a-level x)))
                         (circled 'a x (add1 (a-level x)))))
           (a-core x))
```

```
(defn-sk b-core (x)
  ;; introduced "inductively" so that the level will be tight
  (and (b x)
       (forall b-level
               (implies (and (numberp b-level)
                             (circled ’b x b-level))
                        (circled ’b x (add1 b-level))))))

Skolem axiom:
      (AND (IMPLIES (AND (B X)
                         (IMPLIES (AND (NUMBERP (B-LEVEL X))
                                       (CIRCLED ’B X (B-LEVEL X)))
                                  (CIRCLED ’B X (ADD1 (B-LEVEL X)))))
                    (B-CORE X))
           (IMPLIES (NOT (AND (B X)
                              (IMPLIES (AND (NUMBERP B-LEVEL)
                                            (CIRCLED ’B X B-LEVEL))
                                       (CIRCLED ’B X (ADD1 B-LEVEL)))))
                    (NOT (B-CORE X))))

LEMMA.  B-CORE-NECC
    ;; the conclusion implies the more obvious consequence of b-core
  (implies (not (circled ’b x n))
           (not (b-core x)))

LEMMA.  B-CORE-SUFF
  (implies (and (b x)
                (implies (and (numberp (b-level x))
                              (circled ’b x (b-level x)))
                         (circled ’b x (add1 (b-level x)))))
           (b-core x))

(defn parity (n)
  (if (zerop n)
      t
    (not (parity (sub1 n)))))

(defn j (x)
  ;; the isomorphism
  (if (or (a-core x)
          (parity (a-level x)))
      (fa x)
    (fb-1 x)))

(defn j-1 (y)
  ;; the isomorphism’s inverse
  (if (or (b-core y)
          (not (parity (b-level y))))
      (fa-1 y)
    (fb y)))

;;;;;;;;;; MAIN LEMMAS ;;;;;;;;;;

LEMMA.  B-CORE-FA
  (implies (a x)
           (iff (b-core (fa x)) (a-core x)))

LEMMA.  CIRCLED-MONOTONE
  (implies (and (circled flg x j)
                (not (lessp j i)))
           (circled flg x i))

LEMMA.  B-LEVEL-FA
  (implies (and (a x) (not (a-core x)))
           (equal (b-level (fa x))
                  (add1 (a-level x))))

LEMMA.  A-CORE-FB
  (implies (b x)
           (iff (a-core (fb x)) (b-core x)))
```

```
LEMMA.  A-LEVEL-FB
  (implies (and (b x) (not (b-core x)))
           (equal (a-level (fb x))
                  (add1 (b-level x))))

LEMMA.  B-LEVEL-FB-1
  (implies (and (a x)
                (not (a-core x))
                (in-fb-range x))
           (equal (b-level (fb-1 x))
                  (sub1 (a-level x))))

LEMMA.  B-CORE-FB-1
  (implies (and (a x)
                (not (a-core x))
                (not (parity (a-level x))))
           (not (b-core (fb-1 x))))

LEMMA.  J-1-J
  (implies (a x)
           (equal (j-1 (j x)) x))

LEMMA.  A-CORE-FA-1
  (implies (in-fa-range y)
           (iff (a-core (fa-1 y)) (b-core y)))

LEMMA.  A-LEVEL-FA-1
  (implies (and (b y)
                (not (b-core y))
                (in-fa-range y))
           (equal (a-level (fa-1 y))
                  (sub1 (b-level y))))

LEMMA.  J-J-1
  (implies (b y)
           (equal (j (j-1 y)) y))

LEMMA.  J-IS-ONE-ONE
  (implies (and (a x1)
                (a x2)
                (not (equal x1 x2)))
           (not (equal (j x1) (j x2))))
```

```
(defn-sk j-iso nil
  (and
   ;; j maps a into b
   (forall x
           (implies (a x)
                    (b (j x))))
   ;; j is one-one
   (forall (x1 x2)
           (implies (and (a x1)
                         (a x2)
                         (equal (j x1) (j x2)))
                    (equal x1 x2)))
   ;; j is onto
   (forall y (implies (b y)
                      (exists x (and (a x)
                                     (equal (j x) y))))))
  ((prefix j-iso-)))

Skolem axiom:
      (AND (IMPLIES (AND (IMPLIES (A (J-ISO-X))
                                  (B (J (J-ISO-X))))
                         (IMPLIES (AND (A (J-ISO-X1))
                                       (A (J-ISO-X2))
                                       (EQUAL (J (J-ISO-X1)) (J (J-ISO-X2))))
                                  (EQUAL (J-ISO-X1) (J-ISO-X2)))
                         (IMPLIES (B (J-ISO-Y))
                                  (AND (A X) (EQUAL (J X) (J-ISO-Y)))))
                    (J-ISO))
           (IMPLIES (NOT (AND (IMPLIES (A X) (B (J X)))
                              (IMPLIES (AND (A X1)
                                            (A X2)
                                            (EQUAL (J X1) (J X2)))
                                       (EQUAL X1 X2))
                              (IMPLIES (B Y)
                                       (AND (A (J-ISO-X-1 Y))
                                            (EQUAL (J (J-ISO-X-1 Y)) Y)))))
                    (NOT (J-ISO))))

LEMMA.  J-IS-AN-ISOMORPHISM
  (j-iso)
```

Appendix A
The DEFN-FO mechanism

A convenient "pseudo-event" DEFN-FO is provided. Its syntax is exactly the same as DEFN-SK. However, if no FORMULA hint is provided and if a "good" hint can be deduced, then that hint is added. So for example, the example in the "manual section" (see Subsection 2.4) could be generated by:

```
(DEFN-FO BAR (Z)
  (EXISTS X
          (FORALL Y (IMPLIES (P X Z) (P Y Z))))
  ((PREFIX SK-)))
```

Moreover, if the Lisp variable **`*ADD-SKOLEM-PREFIXES*`** (which is initially **`NIL`**) is set to anything other than **`NIL`** and if no PREFIX hint is given to a DEFN-FO, then the hint **`(PREFIX <name>-)`** will be added where **`<name>`** is the name of the function being defined. The analogous situation applies for **`*ADD-SKOLEM-SUFFIXES*`** and **`(SUFFIX -<name>)`**.

NOTE: DEFN-FO is just a convenient macro, *not* an event form suitable for an events file. When it generates new hints, it prints out the new hints. The user is strongly encouraged to copy these hints into a DEFN-SK form that he places into his events file (possibly with a remark that the hints were generated with DEFN-FO).

Appendix B
Skolemization

This appendix is included primarily for completeness only, as our Skolemization algorithm is quite standard. We do make a few comments on some choices that we have made, however.

The Algorithm

Fix a history. We present Skolemization as a function which takes arguments

- a formula ϕ
- a boolean *parity*
- a set *F* of function names, each associated with an arity
- a set *V* of variables

and returns a term. Typically *F* will contain all function symbols of ϕ and of the current history, while *V* will contain all free variables of ϕ. When the parity is **`T`** we generally call the resulting term a *Skolemization* of the given formula. The definition is by recursion along the structure of formulas.

- If ϕ is a term, then return ϕ.
- If ϕ is **`(Q x ψ)`** where **`Q`** is **`FORALL`** or **`EXISTS`**, then there are two subcases.
  - If **`Q`** is **`FORALL`** and *parity* is *true* or if **`Q`** is **`EXISTS`** and *parity* is *false*, then let **`v`** be a variable not in the set *V*, and return the result of Skolemizing **`ψ/{<x,v>}`** with the same *parity* and *F* but with *V* replaced by $V \cup$ **`{v}`**.
  - Otherwise, let **`fn`** be any function symbol which does not belong to *F*. Let **`u`** be the term **(`fn` x_1 `...` x_n)**, where **(**x_1 `...` x_n**)** is an enumeration of all free variables in ϕ. Then we add **`fn`** (with arity **`n`**) to *F*, leave *V* and *parity* unchanged, and Skolemize the result of substituting **`u`** for **`x`** in ψ. As usual, it may be necessary to rename bound variables in ψ in order to substitute correctly; we omit details.
- If ϕ is **`(NOT ψ)`**, then return the negation of the result of Skolemizing ψ with respect to the opposite *parity*, and with *F* and *V* unchanged.
- Otherwise, ϕ must be of the form **`(C ψ θ)`**, where **`C`** is **`OR`**, **`AND`**, or **`IMPLIES`**, or of the form **`(IF u ψ θ)`**. There are two subcases.
  - Suppose that *parity* is *true* and the connective is **`AND`** or **`IF`**, or that *parity* is *false* and the connective is **`OR`** or **`IMPLIES`**. Let **`ψ′`** be the Skolemization of ψ with respect to the given *parity* (but negated if the connective is **`IMPLIES`**), *V*, and *F*. Let **`ρ′`** be the Skolemization of ρ with respect to *parity*, *V*, and the union of *F* with the set of function symbols of **`ψ′`** (each associated with arity). Then return **`(C ψ′ ρ′)`** (except for the **`IF`** case, in which case return **`(IF u ψ′ ρ′)`**).
  - Otherwise, the result is the same as in the first case, except that when we Skolemize ρ, we replace *V* by the union of *V* with the set of the variables occurring in **`ψ′`**.

Correctness

We present here a proof of the specification of Skolemization stated in Subsection 2.2. Actually, we break the property into its two halves, one for conservativity and one for provability in the converse direction. The actual theorems that we prove are slightly more general, in that they apply to formulas with free variables and to either parity; this seems necessary for the inductive proofs. First we state an easy but helpful lemma. We omit its proof, which is a simple proof by induction on the structure of the given formula. It captures the idea that the variables in *V* are "protected" from use as Skolem variables.

LEMMA 1. Let **sk** be a Skolemization of ϕ with respect to *V*, *F*, and *parity*, where *V* contains all free variables of ϕ. Then no variable in *V* both occurs in **sk** and is not free in ϕ. -|

PROPOSITION 1 (Skolemization is strong enough). Let ϕ be a formula, let *F* be a set of function symbols, let *V* be a set of variables which includes all free variables of ϕ, and let *parity* be *true* or *false*. Let **sk** be the Skolemization of ϕ with respect to *F*, *V*, and *parity*, and let **(**v_1 **...** v_n**)** be an enumeration of the variables occurring in **sk** which are not members of *V* (equivalently, not free in ϕ, by Lemma 1). Then:

(i) If *parity* is *true*, then **|- (IMPLIES (FORALL (**v_1 **...** v_n**) sk)** ϕ**)**;

(ii) If *parity* is *false*, then **|- (IMPLIES** ϕ **(EXISTS (**v_1 **...** v_n**) sk))**.

Proof. We induct on the structure of the formula ϕ. If ϕ is a term, then **sk** equals ϕ and we're done.

Next suppose that ϕ is **(Q x** ψ**)** where **Q** is **FORALL** or **EXISTS**. There are several subcases. First suppose that **Q** is **FORALL** and *parity* is *true*. (The next subcase would be where **Q** is **EXISTS** and *parity* is *false*; but that is similar to the current case, and is therefore left to the reader.) Then by definition there is a variable **v** not in the set *V* such that **sk** is the result of Skolemizing ψ**/{<x,v>}** with respect to the same *true* parity and same set *F*, but with *V* replaced by $V \cup$ **{v}**. By the inductive hypothesis, we have

(1) |- (IMPLIES (FORALL (v_1 **...** v_n**) sk)** ψ**/{<x,v>})**

where **(**v_1 **...** v_n**)** is an enumeration of the variables occurring in **sk** which are not free in ψ**/{<x,v>}**. Notice that none of these variables is **v** (unless **x** does not occur free in ψ, an easy case which we leave to the reader). Therefore (1) implies

|- (IMPLIES (FORALL (v v_1 **...** v_n**) sk) (FORALL v** ψ**/{<x,v>}))**

which (by a familiar renaming argument from predicate logic) implies

```
|- (IMPLIES (FORALL (v v_1 ... v_n) sk) φ)
```

as desired.

For the remaining quantifier cases, let **u** be a term of the form **(fn** x_1 **...** x_n**)**, where **(**x_1 **...** x_n**)** is an enumeration of all free variables in ϕ, and where **fn** is a function symbol which does not belong to *F*. Then **sk** is a Skolemization of the result of substituting such a term **u** for **x** in ψ, where renaming of bound variables is done in ψ as required so that the familiar properties of substitution hold, and where the Skolemization is with respect to same *V* and *parity*, but where we add **fn** (with arity **n**) to *F*. Let us consider the case where *parity* is *false* and hence ϕ is **(FORALL x** ψ**)**; the other case is similar. By the inductive hypothesis, we know

```
(2)   |- (IMPLIES ψ′ (EXISTS (v_1 ... v_n) sk))
```

where ψ' is the result of substituting **u** for **x** in ψ (as indicated above) and where **(**v_1 **...** v_n**)** is an enumeration of the variables occurring in **sk** which are not free in ψ', or equivalently, not free in ϕ. The conclusion then follows from the observation that by a familiar rule of predicate logic, we have **|- (IMPLIES (FORALL x** ψ**)** ψ'**)**.

It remains only to consider the various propositional subcases. The case where ϕ is **(NOT** ψ**)** is easy and left to the reader. So ϕ must be of the form **(C** ψ θ**)**, where **C** is **AND**, **OR**, or **IMPLIES**, or of the form **(IF u** ψ θ**)**. Consider for example the case that *parity* is *true* and that ϕ is **(AND** ψ_1 ψ_2**)**. Let sk_i be the appropriate Skolemization of ψ_i (**i** = 1,2). By the inductive hypothesis, we have

```
(3)   |- (IMPLIES (FORALL (v_1 ... v_m) sk_1) ψ_1)

(4)   |- (IMPLIES (FORALL (w_1 ... w_n) sk_2) ψ_2)
```

where **(**v_1 **...** v_m**)** is an enumeration of the variables occurring in sk_1 which are not in *V* and **(**w_1 **...** w_n**)** is an enumeration of the variables occurring in sk_2 which are not in *V*. Let **(**y_1 **...** y_k**)** be an enumeration of **(**v_1 **...** v_m**)** $\cup$ **(**w_1 **...** w_n**)**. It then follows from (3) and (4) and predicate logic that

```
(5)   |- (IMPLIES (FORALL (y_1 ... y_k)
                          (AND sk_1 sk_2))
                  (AND ψ_1 ψ_2))
```

which is what we wanted to prove. The case that ϕ is **(OR** ψ_1 ψ_2**)** is similar; but we need to know that the sets **{**v_1 **...** v_m**}** and **{**w_1 **...** w_n**}** are disjoint in order to obtain the analogue of (5) from (3) and (4). However, the definition of Skolemization guarantees this. Each remaining propositional cases is similar to either the **AND** or **OR** case with *true* parity. -|

Let us turn now to the property of Skolemization which is complementary to the one just proved. For convenience, let us say that a formula ρ with universal closure ρ' *conservatively extends* a set *h* of sentences if *h* ∪ *{*ρ'*}* is a conservative extension of *h*, i.e. if every theorem of this union which is in the language of *h* is already a theorem of *h*. The following lemma is the key to Proposition 2 below.

LEMMA 2. Let **(EXISTS v ψ)** be a formula and let **(v_1 ... v_n)** be an enumeration of its free variables. Let **fn** be a function symbol which does not occur in a given set *h* of sentences or in ψ. Let ρ be the result of substituting the term **(fn v_1 ... v_n)** for **v** in ψ, renaming bound variables as necessary to that the substitution is acceptable in the usual sense of predicate calculus. Then the formulas **(IMPLIES (EXISTS v ψ) ρ)** and **(IMPLIES ρ (FORALL v ψ))** each conservatively extend *h*.

Proof sketch. First let us note that an arbitrary model of *h* can be expanded to a model of the universal closure of **(IMPLIES (EXISTS v ψ) ρ)** (respectively, of **(IMPLIES ρ (FORALL v ψ))**). One simply interprets the new function symbol **fn** so that for any assignment of values to **{v_1, ..., v_n}**, whenever there exists a value for **v** which makes ψ true (respectively false, for the second case) under that assignment, the application (interpretation) of **fn** to those values should make ψ true (respectively false). That there are no new theorems in the extension of *h* by that universal closure now follows easily from the completeness theorem. For suppose that φ is a sentence in the language of *h* which is not provable in *h*. Then there is a model of *h* which satisfies **(NOT φ)**. As we described above, there is an interpretation of **fn** which gives an expansion of this model satisfying the universal closure of **(IMPLIES (EXISTS v ψ) ρ)** (respectively, of **(IMPLIES ρ (FORALL v ψ))**). Since that model still satisfies **(NOT φ)**, φ is still not a theorem of the extension of *h* by that same universal closure. -|

PROPOSITION 2 (conservativity of Skolemization). Assume the hypotheses of Proposition 1, except replace the hypothesis about *V* with a hypothesis that *F* includes all function symbols of the current history *h* and the formula φ.[10] Then:

(i) If *parity* is *true*, then **(IMPLIES φ (FORALL (v_1 ... v_n) sk))** conservatively extends *h*.

(ii) If *parity* is *false*, then **(IMPLIES (EXISTS (v_1 ... v_n) sk) φ)**. conservatively extends *h*.

[10]In the implementation we also check that all function symbols occurring in φ are in *h*.

Proof. We follow the structure in the proof of Proposition 1. Specifically, as with Proposition 1, we induct on the structure of the formula ϕ, noting that the base case where ϕ is a term is trivial.

Next suppose that ϕ is **(Q x** ψ**)** where **Q** is **FORALL** or **EXISTS**. There are several subcases. First suppose that **Q** is **FORALL** and *parity* is *true*. (The next subcase would be where **Q** is **EXISTS** and *parity* is *false*; but that is similar to the current case, and is therefore left to the reader.) Then by definition there is a variable **v** not in the set *V* such that **sk** is the result of Skolemizing ψ**/{<x,v>}** with respect to the same *true* parity and same set *F*, but with *V* replaced by $V \cup$ **{v}**. By the inductive hypothesis, the following formula conservatively extends *h*:

(6)   (IMPLIES ψ**/{<x,v>} (FORALL (**v_1 **...** v_n**) sk))**

where **(**v_1 **...** v_n**)** is an enumeration of the variables occurring in **sk** which are not free in ψ**/{<x,v>}**. Notice that none of these variables is **v** (unless **x** does not occur free in ψ, an easy case which we leave to the reader). Therefore the universal closure of (6) implies the universal closure of

(IMPLIES (FORALL v ψ**/{<x,v>}) (FORALL (v** v_1 **...** v_n**) sk))**

which (by a familiar renaming argument from predicate logic) implies

(IMPLIES ϕ **(FORALL (v** v_1 **...** v_n**) sk))**

and hence this formula conservatively extends *h* as well, as desired.

There are two remaining quantifier cases. Let us consider the case where *parity* is *false* and hence ϕ is **(FORALL x** ψ**)**; the other case is similar. Let **u** be a term of the form **(fn** x_1 **...** x_n**)**, where **(**x_1 **...** x_n**)** is an enumeration of all free variables in ϕ, and where **fn** is a function symbol which does not occur in the current history, in *F*, or in ϕ. Then **sk** is a Skolemization of the result of substituting such a term **u** for **x** in ψ, where renaming of bound variables is done in ψ as required (so that the familiar properties of substitution hold), and where the Skolemization is with respect to same *V* and *parity*, but where we add **fn** (with arity **n**) to *F*. By Lemma 2, the following formula conservatively extends *h*:

(7)   (IMPLIES ψ' ϕ**)**

where ψ' is the result of substituting **u** for **x** in ψ. By applying the inductive hypothesis, replacing *h* with its extension by the universal closure of (7), we find that the formula

(8)   (IMPLIES (EXISTS (v_1 **...** v_n**) sk)** ψ'**)**

conservatively extends the result of adding the universal closure of (7) to *h*. By transitivity of conservative extension, the conservativity of (7) and (8) yields conservativity of

```
     (IMPLIES (EXISTS (v_1 ... v_n) sk) φ)
```

as desired.

It remains only to consider the various propositional subcases. The case where φ is **(NOT ψ)** is easy and left to the reader. The other cases are all similar, so let us do the case that *parity* is *true* and that φ is **(AND ψ_1 ψ_2)**. Let **sk_i** be the Skolemization of ψ_1 (**i** = 1,2), both with respect to the given set *V* of variables and with parity *true*, but where we use *F* for **sk_1** but the union of *F* with the set of function symbols (with arity) in **sk_1** to obtain **sk_2**. By the inductive hypothesis, we have conservativity of

```
(9)   (IMPLIES ψ_1 (FORALL (v_1 ... v_m) sk_1))
```

over *h*, and then conservativity of

```
(10)  (IMPLIES ψ_2 (FORALL (w_1 ... w_n) sk_2))
```

over the extension of *h* by the universal closure of (9), where **(v_1 ... v_m)** is an enumeration of the variables occurring in **sk_1** which are not free in ψ_1 and **(w_1 ... w_n)** is an enumeration of the variables occurring in **sk_2** which are not free in ψ_2. (Notice that in order to apply the inductive hypothesis for the case of (10), we use the fact that the Skolemization of ψ_2 is with respect to a set of function symbols which includes all function symbols of formula (9).) Let **(z_1 ... z_k)** be an enumeration of **(v_1 ... v_m)** ∪ **(w_1 ... w_n)**. The result now follows from the observation that the following formula is a consequence of (9) and (10) by predicate logic:

```
    (IMPLIES (AND ψ_1 ψ_2)
             (FORALL (z_1 ... z_k)
                     (AND sk_1 sk_2)))
```

The remaining propositional cases are similar, so we omit them. -|

Remarks on choices

This section contains a summary of remarks made in [8]; indeed, some of the text here is taken directly from that reference. However, we occasionally refer the reader to [8] for a more complete discussion.

Remark 1. Let's keep Skolem terms small!

Suppose that one introduces a predicate φ of no arguments as follows:

```
(11)   φ ≡ (EXISTS x (IMPLIES (p x) (FORALL y (p y))))
```

A Skolemization of formula (11) is the following, where **sk-x** and **sk-y** are the Skolem functions introduced:

```
(and (implies (implies (p x) (p (sk-y)))
              φ)
     (implies (not (implies (p (sk-x)) (p y)))
              (not φ)))
```

Thus, to prove φ, it suffices to find an **x** such that we can prove

```
(IMPLIES (p x) (p (sk-y)))
```

and of course, instantiating **x** to be **(sk-y)** does the trick.

But a perhaps common approach to Skolemization is, roughly speaking, is to let the Skolem functions depend on all the "governing" quantified variables that have been encountered as one dives into the formula. With that approach one would instead get the following Skolemization:

```
(AND (IMPLIES (IMPLIES (p x) (p (sk-y x)))
              φ)
     (IMPLIES (NOT (IMPLIES (p (sk-x)) (p y)))
              (NOT φ)))
```

Unfortunately it now seems difficult to prove φ using the first conjunct (in a simple rewriting approach, at any rate -- as Bob Boyer points out, resolution knocks it off in 3 unit resolutions!), since if one instantiates this formula by substituting **(sk-y x)** for **x**, one obtains

```
(IMPLIES (IMPLIES (p (sk-y x)) (p (sk-y (sk-y x))))
         φ)
```

which isn't helpful. The desirability of minimality of the set of Skolem variables is formalized in the first section of [8].

Remark 2. Pushing quantifiers inward.

Having asserted that we want the new function symbols to depend on as few variables as possible, we now ask: how are we to accomplish this? The design decision outlined in the introduction to Section 2 suggests that one method is to push quantifiers inward. The example in Subsection 2.4 ("manual section"),

```
(BAR Z) ≡ (EXISTS x (FORALL y (IMPLIES (p x z) (p y z))))
```

would Skolemize with the Skolem function for **y** depending on **x** and **z**

```
(AND (IMPLIES (IMPLIES (p x z) (p (sk-y x z) z))
              (bar z))
     (IMPLIES (NOT (IMPLIES (p (sk-x z) z) (p y z)))
              (NOT (bar z)))) .
```

However, as in the previous example, the first conjunct is not quite sufficient to prove **(bar z)** "naturally". If instead we move quantifiers inward and observe

```
(BAR Z) ≡ (IMPLIES (FORALL x (p x z)) (FORALL y (p y z)))
```

we obtain instead a "better" Skolemization from the point of view of allowing a trivial derivation of `(bar z)`:

```
(AND (IMPLIES (IMPLIES (p x z) (p (sk-y z) z))
              (BAR z))
     (IMPLIES (NOT (IMPLIES (p (sk-x z) z) (p y z)))
              (NOT (bar z))))
```

For, now `(bar z)` is clear by instantiating `x` to `(sk-y z)` in the first conjunct above. The moral of this story is: "smart" Skolemizations can be obtained by being smart about where the quantifiers are placed *before* going into Skolemization. But that’s not quite our entire story......

Remark 3. Outside-in vs. inside-out.

At one point we were tempted to Skolemize "inside-out" instead of "outside-in", in the following sense. Suppose that one wants to Skolemize, for example, `(FORALL x (EXISTS y (p x y)))`. The algorithm presented in this appendix works "outside-in" by stripping off the `(FORALL x)` and calling the Skolemizer on `(EXISTS y (p x y))`, which in turn gives us a result of the form `(p x (f x))`. However, we might imagine a different, "inside-out" version, which says first that `(EXISTS y (p x y))` Skolemizes to `(p x (f x))` and hence `(FORALL x (EXISTS y (p x y)))` Skolemizes to this same thing.

However, here is an example which shows that the "inside-out" method can lead to a more cumbersome result than the "outside-in" method. (That’s too bad, because an "inside-out" approach would be simpler since one would never be performing the delicate operation of substituting into quantified formulas.) Consider the formula

```
(EXISTS y (FORALL z (EXISTS w (p y z w))))
```

The ("outside-in") algorithm presented in this appendix yields a Skolemization of `(p (sk-y) z (sk-w z))`. However, the "inside-out" method would produce a result as shown by the following "trace":

```
>> (EXISTS y (FORALL z (EXISTS w (p y z w))))

     >> (FORALL z (EXISTS w (p y z w)))

          >> (EXISTS w (p y z w))

          << (p y z (sk-w y z))

     << (p y z (sk-w y z))

<< (p (sk-y) z (sk-w (sk-y) z))
```

Now of course one might argue that `w` is "governed" only by `z`, not by `y`. However, Remark 1 above shows the danger of using such a "governs" notion. Perhaps someone can tell us a way to do "inside-out" Skolemization which avoids the ugliness demonstrated above and yet avoids excess free variables (as discussed in Remark 1 above). In the meantime, we’ll continue to use an "outside-in" method, since it seems to suffer no deficiences for our purposes.

Appendix C
Extending the soundness proof of [2]

This appendix is rather tedious and technical. Its purpose is to extend the argument given in [2] to the case where DEFN-SK is included in the underlying logic. Specifically, the notion of the *obvious extension* of an *extensible functional substitution* and history is defined and shown to have the following property:

Theorem [2]. The obvious extension of an extensible functional substitution and history is itself extensible.

The problem is that this theorem was proved in [2] for a logic not including DEFN-SK. Below we extend the notion of *obvious extension* so that this theorem remains true in the context of DEFN-SK. It is then routine to check that the arguments in [2] all extend to the logic with DEFN-SK. (One need only change all references to "DEFN or CONSTRAIN" to references to "DEFN-SK or DEFN or CONSTRAIN".) In particular, the following theorem holds in for the version of the logic which has DEFN-SK:

Lemma. Justification of Functional Instantiation with Extension (adapted from [2]).
Suppose
h is a history,
`fs` is a tolerable functional substitution,
`p` is a proof of **`thm`** with respect to *h*,
no variable free in **`fs`** occurs in **`p`**,
and **`<h, fs>`** is extensible, and, furthermore,
for each DEFN or CONSTRAIN or DEFN-SK of *h* whose instance under **`fs`**
is not a theorem under **`fs`**, none of the function symbols introduced
by the DEFN or CONSTRAIN or DEFN-SK is ancestral in **`thm`**.
Then **`thm\fs`** is a theorem in a DEFN/CONSTRAIN/DEFN-SK extension of *h*.

We will prove the theorem above; the justification lemma immediately above then follows from that theorem just as it does in [2]. That is, we need to extend the notion of *obvious extension* from [2] to the case where the existing logic includes DEFN-SK, so that the obvious extension of an extensible functional substitution and history is itself extensible. We omit all definitions from the following exposition; in fact, we gear it to those who have already read through [2]. In particular, we assume that the reader has read through the DEFN and CONSTRAIN cases of the definition of obvious extension. The truth is that we include this argument for completeness only; if anyone reads this the author would be interested to know it!

(c) Suppose that the event is a DEFN-SK with axiom **`ax`**, where **`ax`** is a Skolemization of a first-order formula ϕ of the form

(#) **(FORALL (x_1 ... x_k) (IFF (f x_1 ... x_k) θ])**

(where θ may be from the formula hint or the body of the DEFN-SK). We will find an extension **FS′′** of **FS**, with the same free variables as **FS** and domain extended only by function symbols introduced with **ax**, together with a DEFN-SK event, such that **ax\FS′′** is an instance of the axiom added by this new event. Moreover, this new axiom will not mention any free variable of **FS′′**; hence the result will remain extensible.

The proof will follow from the following lemma. For a first-order formula ϕ and a functional substitution **FS**, we write ϕ**\FS** to denote the result of substituting **FS** into ϕ in the most obvious way possible, i.e. passing through the quantifiers (without worrying about renaming bound variables) and connectives. (The Skolemization algorithm that we use is given in Appendix B.)

Skolemization Extension Lemma. Let ϕ be a first-order formula and let **FS** be a functional substitution with free variables $\mathbf{z_1}$, ..., $\mathbf{z_n}$, such that none of these occur bound in ϕ. Let **sk** be a Skolemization of ϕ with respect to a set *V* of variables which contains all free variables of ϕ and a set *F* of function symbols including all those occurring in ϕ, and parity *true* or *false*. Then there is an extension **FS′** of **FS** to the Skolem functions of **sk**, such that **FS′** has no free variables other than $\mathbf{z_1}$, ..., $\mathbf{z_n}$, and a Skolemization **sk′** of ϕ**\FS**, with respect to $V \cup$ **{**$\mathbf{z_1}$**, ...,** $\mathbf{z_n}$**}** together with any extension of *F* which includes the domain of **FS** and the same parity as for **sk**, such that **sk′** = **sk\FS′**.

Deferring the proof of this lemma for a moment, let us show why it suffices. Let $\mathbf{FS_0}$ be the result of replacing each free occurrence of any $\mathbf{a_i}$ in **FS** with a corresponding distinct variable $\mathbf{a_i}$***** which is new for the current history and for **FS** and does not occur in ϕ or **ax**. Then let $\mathbf{FS_1}$ be the extension of $\mathbf{FS_0}$ by the pair **<f, (LAMBDA (**$\mathbf{x_1}$**′ ...** $\mathbf{x_k}$**′) (f*** $\mathbf{x_1}$**′ ...** $\mathbf{x_k}$**′** $\mathbf{a_1}$*** ...** $\mathbf{a_n}$***))>**, where **f*** is a new function symbol and each $\mathbf{x_i}$**′** is brand new. We may now apply the lemma to the original ϕ with **sk** = **ax**, **FS** = $\mathbf{FS_1}$, and variables **{**$\mathbf{a_1}$***, ...,** $\mathbf{a_n}$***}** and thus obtain an extension **FS′** of $\mathbf{FS_1}$ with free variables contained in **{**$\mathbf{a_1}$***, ...,** $\mathbf{a_n}$***}**, as well as a Skolemization **sk′** of ϕ****$\mathbf{FS_1}$, such that **sk′** = **sk\FS′**. That is, **ax\FS′** = **sk′**, where **sk′** is the Skolem axiom for a new DEFN-SK event introducing the function **f*** with formal parameters **(**$\mathbf{x_1}$ **...** $\mathbf{x_k}$ $\mathbf{a_1}$*** ...** $\mathbf{a_n}$***)** and body θ****$\mathbf{FS_1}$ (with θ from (#) above). Notice that no $\mathbf{a_i}$ occurs in **sk′**. The only reason we're not done is that **FS′** does not extend the original functional substitution, but rather extends the result $\mathbf{FS_0}$ of replacing each free occurrence of any $\mathbf{a_i}$ with $\mathbf{a_i}$*****. However, if we let **FS′′** be the result of substituting into **FS′** each occurrence of any $\mathbf{a_i}$***** with $\mathbf{a_i}$, and let **A*** be the substitution **{<**$\mathbf{a_i}$***,**$\mathbf{a_i}$**>}**, then by the claim following this paragraph, **ax\FS′′** = **ax\FS′/A***, which equals **sk′/A***.

(In order to apply that claim, we need to know that no $\mathtt{a_i*}$ is in the scope of a LAMBDA in `FS'` where $\mathtt{a_i}$ is bound. This is clear for $\mathtt{FS_1}$ since each $\mathtt{a_i*}$ was introduced only for free occurrences of $\mathtt{a_i}$, and hence is true without loss of generality for `FS'` as well, by renaming bound variables in the pairs belonging to `FS'` and not to $\mathtt{FS_1}$. Actually we also need to know that no $\mathtt{a_i*}$ is a bound variable of `FS'`, which is also true without loss of generality by renaming.) Therefore `ax\FS''` is an instance of the axiom added by the new DEFN-SK event. Also notice that `FS''` has no free variables other than $\mathtt{a_1,\ ...,\ a_n}$, and none of these occurs in `sk'`.

We promised just above to state and prove a claim, and here it is. Suppose `t` is a term, $\mathtt{FS_1}$ and $\mathtt{FS_2}$ are functional substitutions and `s` is a substitution such that $\mathtt{FS_2}$ is `{<f,(LAMBDA vars (u/s))>: <f,(LAMBDA vars u)>` $\in$ $\mathtt{FS_1}$`}`. Assume furthermore that no variable occurring in `t` belongs to the domain of `s`, that no variable in the domain of `s` is a bound variable in $\mathtt{FS_1}$, and that

(**) for each `(LAMBDA vars u)` as above, it is the case that for every variable `v` in the domain of `s` which occurs in `u`, `s(v)` contains no occurrence of a variable which belongs to `vars`.

Then $\mathtt{t\backslash FS_2}$ = $\mathtt{t\backslash FS_1/s}$. The proof is of course by structural induction on `t`. If `t` is a variable then the result is clear since by hypothesis, `t` is not in the domain of `s`. If `t` is $\mathtt{(f\ t_1\ ...\ t_n)}$ where `f` is not in the domain of $\mathtt{FS_1}$ then the claim is clear by the inductive hypothesis. Finally suppose that `t` is $\mathtt{(f\ t_1\ ...\ t_n)}$ where the pair $\mathtt{<f,(LAMBDA\ (x_1\ ...\ x_n)\ u)>}$ belongs to $\mathtt{FS_1}$. Then $\mathtt{t\backslash FS_2}$ = $\mathtt{u/s/\{<x_i,t_i\backslash FS_2>\}}$ = $\mathtt{u/(s\ \cup\ \{<x_i,t_i\backslash FS_2>\})}$, by (**) and the fact that no $\mathtt{x_i}$ is in the domain of `s`. On the other hand, $\mathtt{t\backslash FS_1/s}$ = $\mathtt{u/\{<x_i,t_i\backslash FS_1\}>/s}$ = $\mathtt{u/(<x_i,t_i\backslash FS_1/s>\ \cup\ s)}$. Therefore we conclude by applying the inductive hypothesis.

It remains then only to prove the Skolemization Extension Lemma, which we do by structural induction on the formula ϕ. Actually, we prove a slightly stronger result. The stronger version says that, moreover, the Skolem variables introduced into `sk'` are identical to the Skolem variables introduced into `sk`.

If ϕ is quantifier-free then we may take `sk'` to be ϕ`\FS` and `FS'` to be `FS`.

Next suppose that ϕ is $\mathtt{(OR}\ \phi_1\ \phi_2\mathtt{)}$, and write `sk` as $\mathtt{(OR\ sk_1\ sk_2)}$ where $\mathtt{sk_1}$ and $\mathtt{sk_2}$ are (respectively) Skolemizations of ϕ_1 and ϕ_2. (The other propositional cases are similar so we'll omit them.) By the inductive hypothesis we may find extensions $\mathtt{FS_1}$ and $\mathtt{FS_2}$ of `FS` to the Skolem functions of $\mathtt{sk_1}$ and $\mathtt{sk_2}$ and corresponding Skolemizations $\mathtt{sk_1'}$ and $\mathtt{sk_2'}$ of ϕ_1`\FS` and ϕ_2`\FS` (respectively) such that $\mathtt{sk_1'}$ =

$\mathtt{sk_1\backslash FS_1}$ and $\mathtt{sk_2'}$ = $\mathtt{sk_2\backslash FS_2}$, where the respective Skolem variables are the same as for $\mathtt{sk_1}$ and $\mathtt{sk_2}$. Since the Skolem functions introduced by $\mathtt{sk_1}$ and $\mathtt{sk_2}$ are disjoint, $\mathtt{FS_1} \cup \mathtt{FS_2}$ is a function; call it $\mathtt{FS'}$. Let $\mathtt{sk'}$ = $\mathtt{(OR\ sk_1'\ sk_2')}$. Then $\mathtt{sk'}$ is a Skolemization of $\phi\backslash\mathtt{FS}$ (here we use the strengthening to guarantee that the Skolem variables are disjoint where they should be) and $\mathtt{sk'}$ = $\mathtt{sk\backslash FS'}$.

Consider next the case that ϕ is $\mathtt{(EXISTS\ v\ P)}$ or $\mathtt{(FORALL\ v\ P)}$, with a parity that requires any Skolemization $\mathtt{sk}$ of ϕ to be a Skolemization of $\mathtt{P/\{<v,v'>\}}$ for an appropriate variable $\mathtt{v'}$ (which may be $\mathtt{v}$ itself). By the inductive hypothesis we may choose an extension $\mathtt{FS'}$ of $\mathtt{FS}$ to the Skolem functions of $\mathtt{sk}$ and a Skolemization $\mathtt{sk'}$ of $\mathtt{P/\{<v,v'>\}\backslash FS}$ (same parity) such that $\mathtt{sk'}$ = $\mathtt{sk\backslash FS'}$. But $\mathtt{P/\{<v,v'>\}\backslash FS}$ = $\mathtt{P\backslash FS/\{<v,v'>\}}$ by the Commutativity Lemma in [2], since by hypothesis $\mathtt{v}$ is not any of the $\mathtt{z_i}$. Therefore $\mathtt{sk'}$ is a Skolemization of $\phi\backslash\mathtt{FS}$. (Notice also that the additional condition about having the same Skolem variables also holds, by the inductive hypothesis.)

Finally consider the case that ϕ is $\mathtt{(EXISTS\ v\ P)}$ or $\mathtt{(FORALL\ v\ P)}$, with a parity that requires any Skolemization $\mathtt{sk}$ of ϕ to be a Skolemization of $\mathtt{P/\{<v,(g\ v_1\ ...\ v_k)>\}}$, where $\mathtt{g}$ is a new Skolem function and $\mathtt{(v_1\ ...\ v_k)}$ is an enumeration of the free variables of ϕ. Let $\mathtt{FS^*}$ be $\mathtt{FS} \cup \mathtt{\{<g,(LAMBDA\ (v_1\ ...\ v_k)\ (g^*\ w_1\ ...\ w_q))>\}}$, where $\mathtt{(w_1\ ...\ w_q)}$ is a list of the free variables of $\phi\backslash\mathtt{FS}$ which we know (by a lemma with a simple inductive proof, which we omit) is contained in the set $\mathtt{\{v_1,\ ...,\ v_k,\ z_1,\ ...\ z_n\}}$, and $\mathtt{g^*}$ is a new function symbol. By the inductive hypothesis, there is an extension $\mathtt{FS'}$ of $\mathtt{FS^*}$ to the Skolem functions of $\mathtt{sk}$ other than $\mathtt{g}$ (having no extra free variables), together with a corresponding Skolemization $\mathtt{sk'}$ of $\mathtt{P/\{<v,(g\ v_1\ ...\ v_k)>\}\backslash FS^*}$, such that $\mathtt{sk'}$ = $\mathtt{sk\backslash FS'}$. But as in the previous case, $\mathtt{P/\{<v,(g\ v_1\ ...\ v_k)>\}\backslash FS^*}$ = $\mathtt{P\backslash FS^*/\{<v,(g\ v_1\ ...\ v_k)>\}}$, which in turn is the same as $\mathtt{P\backslash FS/<v,(g\ v_1\ ...\ v_k)>}$ since $\mathtt{g}$ does not occur in $\mathtt{P}$. Therefore $\mathtt{sk'}$ is a Skolemization of $\phi\backslash\mathtt{FS}$, as required. -|

Appendix D
Koenig's Lemma Events List

The following events go through in a version of NQTHM extended by the functional variables work [2] and DEFN-SK. All text on a line following a semicolon ';' is a comment.

```
;; All this initial stuff is just to get the CONSTRAIN below accepted.

(defn ones (n)
  ;; a list of n ones
  (if (zerop n)
      nil
    (cons 1 (ones (sub1 n)))))

(defn all-ones (s)
  ;; all are ones
  (if (listp s)
      (and (equal (car s) 1)
           (all-ones (cdr s)))
    (equal s nil)))

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

(defn subseq (s1 s2)
  ;; s1 is a terminal subsequence (nthcdr) of s2
  (if (equal s1 s2)
      t
    (if (nlistp s2)
        f
      (subseq s1 (cdr s2)))))

(prove-lemma subseq-all-ones (rewrite)
  (implies (and (all-ones s1)
                (subseq s2 s1))
           (all-ones s2)))

(defn plistp (s)
  (if (listp s)
      (plistp (cdr s))
    (equal s nil)))

(prove-lemma plistp-all-ones (rewrite)
  (implies (all-ones s)
           (plistp s)))

(prove-lemma all-ones-ones (rewrite)
  (all-ones (ones n)))

(prove-lemma ones-is-injective (rewrite)
  (implies (and (numberp i) (numberp j) (not (equal i j)))
           (not (equal (ones i) (ones j))))
  ((induct (lessp i j))))
```

```
(constrain koenig-intro (rewrite)
  ;; Introduce the predicate node-p for the nodes of (i.e. finite paths
  ;; through) the tree.  That is, node-p recognizes the legal paths.
  (and
   ;; nil is the root
   (node-p nil)
   ;; node-p is a predicate
   (or (truep (node-p s))
       (falsep (node-p s)))
   ;; the successors of s are determined by succard (Successors Cardinality)
   (implies (node-p s)
            (equal (node-p (cons n s))
                   ;; n is in {1, ..., (succard s)}
                   (and (lessp 0 n)
                        (not (lessp (succard s) n)))))
   (implies (and (node-p s1)
                 (subseq s s1))
            (node-p s))
   ;; we stipulate that the tree is infinite by saying that s-n is a one-one
   ;; enumeration of nodes
   (node-p (s-n n))
   (implies (and (numberp i) (numberp j) (not (equal i j)))
            (not (equal (s-n i) (s-n j))))
   ;; nodes are proper
   (implies (not (plistp s))
            (not (node-p s))))
  ((node-p all-ones)
   (succard (lambda (s) 1))
   (s-n ones))
  ((disable subseq)))

;; We want to define a function s-height which returns an element of a given height.
;; The next several events culminate in the following lemma:

;; (prove-lemma length-s-height (rewrite)
;;   (equal (length (s-height n)) (fix n)))

(defn succ-aux (s n)
  (if (zerop n)
      nil
    (cons (cons n s) (succ-aux s (sub1 n)))))

(defn successors (s)
  (succ-aux s (succard s)))

(defn successors-list (ss)
  ;; given a list ss of nodes, returns a list of all successor
  ;; to all elements in ss
  (if (listp ss)
      (append (successors (car ss))
              (successors-list (cdr ss)))
    nil))

(defn level (n)
  (if (zerop n)
      (list nil)
    (successors-list (level (sub1 n)))))

(defn init-tree (n)
  ;; level <= n
  (if (zerop n)
      (list nil)
    (append (level n)
            (init-tree (sub1 n)))))
```

```
(defn remove1 (a x)
  ;; remove the first occurrence of a from the list x
  (if (listp x)
      (if (equal a (car x))
          (cdr x)
        (cons (car x)
              (remove1 a (cdr x))))
    x))

(prove-lemma length-remove1 (rewrite)
  (implies (member a x)
           (lessp (length (remove1 a x))
                  (length x))))

(defn first-non-member-index (i x)
  ;; returns index some (s-n j) (j >= i) not in the set x
  (if (member (s-n i) x)
      (first-non-member-index (add1 i) (remove1 (s-n i) x))
    i)
  ((lessp (length x))))

(defn nthcdr (n s)
  (if (zerop n)
      s
    (nthcdr (sub1 n) (cdr s))))

(defn s-height (n)
  ;; should return a node of height n
    (nthcdr (difference (length (s-n (first-non-member-index 0 (init-tree n)))) n)
            (s-n (first-non-member-index 0 (init-tree n)))))

(prove-lemma nthcdr-subseq (rewrite)
  (implies (not (lessp (length s) n))
           (subseq (nthcdr n s) s)))

(prove-lemma node-p-nthcdr (rewrite)
  (implies (and (node-p s)
                (not (lessp (length s) n)))
           (node-p (nthcdr n s))))

(prove-lemma lessp-difference-1 (rewrite)
  (equal (lessp x (difference x n)) f))

(prove-lemma node-p-s-height (rewrite)
  (node-p (s-height n)))

(prove-lemma length-nthcdr (rewrite)
  (equal (length (nthcdr n s))
         (difference (length s) n)))

(prove-lemma first-non-member-index-lessp (rewrite)
  (not (lessp (first-non-member-index i x) i)))

(prove-lemma s-n-first-non-member-index-not-equal (rewrite)
  (implies (numberp i)
           (not (equal (s-n (first-non-member-index (add1 i)
                                                    (remove1 (s-n i) x)))
                       (s-n i)))))

(prove-lemma member-remove1 (rewrite)
  (implies (not (equal a b))
           (equal (member a (remove1 b x))
                  (member a x))))

(prove-lemma s-n-first-non-member-index (rewrite)
  (implies (numberp i)
           (not (member (s-n (first-non-member-index i x))
                        x))))

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

```
(prove-lemma member-cons-succ-aux (rewrite)
             (equal (member (cons z v) (succ-aux v n))
                    (and (lessp 0 z) (not (lessp n z))))
             ((induct (succ-aux v n))))

(prove-lemma node-p-cons-lemma ()
  (implies (not (node-p s))
           (not (node-p (cons n s)))))

(prove-lemma node-p-cons (rewrite)
  (equal (node-p (cons n s))
         (and (node-p s)
              (lessp 0 n)
              (not (lessp (succard s) n))))
  ((use (node-p-cons-lemma))))

(defn all-length-n (ss n)
  (if (listp ss)
      (and (equal (length (car ss)) n)
           (all-length-n (cdr ss) n))
    t))

(prove-lemma all-length-n-append (rewrite)
  (equal (all-length-n (append ss1 ss2) n)
         (and (all-length-n ss1 n)
              (all-length-n ss2 n))))

(prove-lemma all-length-n-succ-aux (rewrite)
  (implies (equal (length s) n)
           (all-length-n (succ-aux s k) (add1 n))))

(prove-lemma all-length-n-successors-list (rewrite)
  (implies (all-length-n ss n)
           (all-length-n (successors-list ss) (add1 n))))

(prove-lemma length-0 (rewrite)
  (equal (equal (length s) 0)
         (nlistp s)))

(defn member-level-induction (s n)
  (if (zerop n)
      t
    (member-level-induction (cdr s) (sub1 n))))

(prove-lemma succ-aux-listp (rewrite)
  (implies (not (listp s))
           (not (member s (succ-aux z n)))))

(prove-lemma successors-list-listp (rewrite)
  (implies (not (listp s))
           (not (member s (successors-list ss)))))

(prove-lemma member-succ-aux nil
  (implies (member s (succ-aux x n))
           (equal (cdr s) x)))

(prove-lemma member-successors-list-successors-list-witness (rewrite)
  (equal (member s (successors-list ss))
         (and (member (cdr s) ss)
              (member s (successors (cdr s)))))
  ((use (member-succ-aux (x (car ss)) (n (succard (car ss)))))
   (induct (successors-list ss))))

(prove-lemma member-level (rewrite)
  (implies (and (numberp n)
                (node-p s))
           (equal (member s (level n))
                  (equal (length s) n)))
  ((induct (member-level-induction s n))))

(prove-lemma member-init-tree (rewrite)
  (implies (node-p s)
           (equal (member s (init-tree n))
                  (not (lessp n (length s))))))
```

```
(prove-lemma length-s-non-member-index (rewrite)
 (implies (numberp i)
          (lessp n
                 (length (s-n (first-non-member-index i
                                                      (init-tree n))))))
 ((use (s-n-first-non-member-index (x (init-tree n))))
  (disable s-n-first-non-member-index)))

(prove-lemma length-s-height (rewrite)
  (equal (length (s-height n)) (fix n)))

(disable s-height)

;; End of s-height excursion.

;; Our goal:

;(prove-lemma konig-tree-lemma nil
;  (and (node-p (k n))
;       (implies (not (lessp j i))
;                (subseq (k i) (k j)))
;       (equal (length (k n)) (fix n))))

(defn-sk inf (s)
  ;; says that s has arbitrarily high successors
  (forall big-h (exists big-s
    (and (subseq s big-s)
         (node-p big-s)
         (lessp big-h (length big-s))))))

;; The following three events were generated mechanically.  They are
;; useful especially for applying the Skolem axioms for INF inside the
;; proof-checker, via the macro command SK*.

(DISABLE INF)

(PROVE-LEMMA INF-SUFF
             (REWRITE)
             (IMPLIES (AND (SUBSEQ S BIG-S)
                           (NODE-P BIG-S)
                           (LESSP (BIG-H S) (LENGTH BIG-S)))
                      (INF S))
             ((USE (INF))))

(PROVE-LEMMA INF-NECC
             (REWRITE)
             (IMPLIES (NOT (AND (SUBSEQ S (BIG-S BIG-H S))
                                (NODE-P (BIG-S BIG-H S))
                                (LESSP BIG-H
                                       (LENGTH (BIG-S BIG-H S)))))
                      (NOT (INF S)))
             ((USE (INF))))

(defn next (s max)
  ;; picks a successor of s that has infinitely many successors,
  ;; if there is one
  (if (zerop max)
      (cons 0 s)                       ;not a legal node --
                                       ;was 0, but this is easier for length calculations
    (if (inf (cons max s))
        (cons max s)
      (next s (sub1 max)))))
```

```
;; We want to show that NEXT gives us a successor with infinitely many
;; successors.

; INF-IMPLIES-INF-NEXT:
;(implies (and (node-p s)
;              (inf s))
;         (inf (next s (succard s))))

;; Note that if some successor of s has infinitely many successors, so
;; does (NEXT S (SUCCARD S)).  This is the lemma
;; INF-CONS-IMPLIES-INF-NEXT below.  But first note:

(prove-lemma inf-implies-node-p (rewrite)
  ;; Done initially with proof-checker.
  (implies (inf s)
           (node-p s))
  ((use (inf-necc))))

(prove-lemma not-inf-zerop (rewrite)
  (implies (zerop i)
           (not (inf (cons i s))))
  ((use (inf-implies-node-p (s (cons i s))))
   (disable inf-implies-node-p)))

(prove-lemma inf-cons-implies-inf-next (rewrite)
  (implies (and (node-p s)
                (inf (cons i s))
                (not (lessp n i)))
           (inf (next s n)))
  ((induct (next s n))))

;; Our goal now is to apply this lemma by proving that
;; (inf (cons i s)) for some i <= (succard s).

(defn all-big-h (s n)
  ;; at least as big as (big-h (cons s i)) for all i <= n and is bigger than length of s
  (if (zerop n)
      (add1 (length s))
    (plus (big-h (cons n s))
          (all-big-h s (sub1 n)))))

(prove-lemma all-big-h-length (rewrite)
  (lessp (length s)
         (all-big-h s n)))

(prove-lemma all-big-h-lessp (rewrite)
             (implies (and (lessp 0 i)
                           (not (lessp n i)))
                      (equal (lessp (big-h (cons i s))
                                    (all-big-h s n))
                             t))
             ((induct (all-big-h s n))))

;; Here’s a function which tells us which way s first branches on its
;; way to extending to s1.

(defn first-branch (s s1)
  ;; assumes s is a proper subsequence of s1
  (if (equal s (cdr s1))
      (car s1)
    (if (nlistp s1)
        0
      (first-branch s (cdr s1)))))

(prove-lemma subseq-cons-first-branch (rewrite)
  (implies (and (subseq s x)
                (not (equal s x)))
           (subseq (cons (first-branch s x) s)
                   x)))
```

```
(prove-lemma length-non-equal (rewrite)
  (implies (lessp (length x) (length y))
           (equal (equal x y) f)))

(prove-lemma first-branch-ok-for-succard (rewrite)
            (implies (and (subseq s big-s)
                          (node-p big-s)
                          (not (equal s big-s)))
                     (and (numberp (first-branch s big-s))
                          (lessp 0 (first-branch s big-s))
                          (not (lessp (succard s)
                                      (first-branch s big-s))))))

(prove-lemma all-big-h-lessp-linear
             (rewrite)
             (implies (and (lessp 0 i) (not (lessp (succard s) i)))
                      (lessp (big-h (cons i s))
                             (all-big-h s (succard s)))))

(disable all-big-h-lessp)

(prove-lemma inf-implies-inf-next (rewrite)
  (implies (and (node-p s) (inf s))
           (inf (next s (succard s))))
  ((use (inf-suff (big-s (big-s (all-big-h s (succard s)) s))
                  (s (cons (first-branch s
                                         (big-s (all-big-h s (succard s)) s))
                           s)))
        (inf-necc (big-h (all-big-h s (succard s))) (s s))
        (all-big-h-lessp-linear
         (i (first-branch s
                          (big-s (all-big-h s (succard s)) s)))))
   (disable all-big-h-lessp-linear)))

(defn k (n)
  ;; picks out the path
  (if (zerop n)
      nil
    (next (k (sub1 n)) (succard (k (sub1 n))))))

(prove-lemma subseq-nil (rewrite)
  (equal (subseq nil x)
         (plistp x))
  ((enable subseq)))

(prove-lemma node-p-implies-plistp (rewrite)
  ;; contrapositive of part of KOENIG-INTRO; needed for proof of INF-NIL
  (implies (node-p s)
           (plistp s)))

(prove-lemma inf-nil
          ;; done with help of proof-checker
          (rewrite)
          (inf nil)
          ((use (inf-suff
                 (big-s (s-height (add1 (big-h nil))))
                 (s nil)))))

(disable node-p-implies-plistp)

(prove-lemma konig-tree-lemma-1 (rewrite)
  (inf (k n)))

(prove-lemma length-next (rewrite)
  (implies (inf x)
           (equal (length (next s n))
                  (add1 (length s)))))

(prove-lemma konig-tree-lemma-2 (rewrite)
  (equal (length (k n)) (fix n)))

(prove-lemma subseq-next (rewrite)
  (implies (subseq s1 s2)
           (subseq s1 (next s2 n))))
```

```
(prove-lemma konig-tree-lemma-3 (rewrite)
  (implies (not (lessp j i))
           (subseq (k i) (k j)))
  ((induct (k j))))

(prove-lemma konig-tree-lemma nil
  (and (node-p (k n))
       (implies (not (lessp j i))
                (subseq (k i) (k j)))
       (equal (length (k n)) (fix n))))

;; or, if one prefers:

(prove-lemma konig-tree-lemma-again nil
  (implies (numberp n)
           (and (node-p (k n))
                (implies (not (lessp j i))
                         (subseq (k i) (k j)))
                (equal (length (k n)) n))))
```

Appendix E
Ramsey Theorem Events List

The following events go through in a version of NQTHM extended by the functional variables work [2] and DEFN-SK. All text on a line following a semicolon ';' is a comment.

```
(constrain p-num-intro (rewrite)
  (implies (and (numberp x) (numberp y))
           (and (lessp 0 (p-num x y))
                (not (lessp (bound) (p-num x y)))
                (equal (p-num x y) (p-num y x))))
  ((p-num (lambda (x y) 1))
   (bound (lambda () 2))))

(disable p-num-intro)

(defn p (x y)
  (p-num (fix x) (fix y)))

(prove-lemma p-intro (rewrite)
  (and (lessp 0 (p x y))
       (not (lessp (bound) (p x y)))
       (equal (p x y) (p y x)))
  ((use (p-num-intro (x (fix x)) (y (fix y))))))

(disable p)

(defn prehom-seq-1 (a x)
  ;; x is a list of pairs (i . c), and this says that for each
  ;; such pair, p(i,a) <-> c.
  (if (listp x)
      (and (equal (p (caar x) a) (cdar x))
           (prehom-seq-1 a (cdr x)))
    t))

(defn prehom-seq (x)
  ;; a list of pairs in decreasing order, such that
  ;; if (j . ?) is before (i . c) (and hence presumably i < j),
  ;; then p(i,j) <-> c.
  (if (listp x)
      (if (listp (cdr x))
          (and (lessp (car (cadr x)) (car (car x)))
               (prehom-seq-1 (caar x) (cdr x))
               (prehom-seq (cdr x)))
        t)
    t))

(defn-sk extensible (s)
  ;; s is a list of pairs (i . c), and extensible means that there
  ;; are infinitely many a for which prehom-seq-1(a,s) holds.
  (forall above
          (exists next
                  (and (lessp above next)
                       (prehom-seq-1 next s)))))

;; The following two lemmas are for the benefit of the proof-checker
;; macro-command SK*.

(prove-lemma extensible-suff (rewrite)
  (implies (and (lessp (above s) next)
                (prehom-seq-1 next s))
           (extensible s)))
```

```
(prove-lemma extensible-necc (rewrite)
  (implies (not (and (lessp above (next above s))
                     (prehom-seq-1 (next above s) s)))
           (not (extensible s)))
  ((use (extensible))
   (disable extensible)))

(defn above-all-aux (a s n)
  (if (zerop n)
      0
    (plus (above (cons (cons a n) s))
          (above-all-aux a s (sub1 n)))))

(defn above-all (a s)
  (above-all-aux a s (bound)))

(prove-lemma lessp-above-all-aux (rewrite)
  (implies (lessp 0 n)
           (not (lessp (above-all-aux a s n)
                       (above (cons (cons a n) s))))))

(prove-lemma above-all-aux-monotone (rewrite)
  (implies (not (lessp bound n))
           (not (lessp (above-all-aux a s bound)
                       (above-all-aux a s n))))
  ((induct (above-all-aux a s bound))))

(prove-lemma lessp-above-all-bound (rewrite)
  (implies (and (lessp 0 n)
                (not (lessp (bound) n)))
           (not (lessp (above-all a s)
                       (above (cons (cons a n) s))))))

(defn next-element (a s)
  (next (above-all a s) s))

(defn next-color (a s)
  (p a (next-element a s)))

(prove-lemma numberp-p (rewrite)
  (numberp (p x y))
  ((use (p-intro))))

(prove-lemma next-color-bound (rewrite)
             (and (lessp 0 (next-color a s))
                  (not (lessp (bound) (next-color a s)))))

(disable above-all)

;; first of two goals for extensible-cons
(prove-lemma lessp-next-element (rewrite)
  (implies (extensible s)
           (lessp (above (cons (cons a (next-color a s)) s))
                  (next-element a s)))
  ((use (extensible-necc (above (above-all a s))))))

(prove-lemma prehom-seq-1-next-element
       (rewrite)
       (implies (extensible s)
                (prehom-seq-1 (next-element a s) s))
       ((use (extensible-necc (above (above-all a s))))))

(disable next-element)

;; second of two goals for extensible-cons
(prove-lemma prehom-seq-1-next-element-cons
             (rewrite)
             (implies (extensible s)
                      (prehom-seq-1 (next-element a s)
                                    (cons (cons a (next-color a s)) s))))

(disable next-color)
```

```
(prove-lemma extensible-cons (rewrite)
 (implies (extensible s)
          (extensible (cons (cons a (next-color a s)) s)))
 ((use (extensible-suff (next (next-element a s))
                        (s (cons (cons a (next-color a s)) s))))))

(defn next-pair (s)
  (cons (next (caar s) s) (next-color (next (caar s) s) s)))

(prove-lemma next-pair-extends (rewrite)
  (implies (extensible s)
           (extensible (cons (next-pair s) s))))

(defn ramsey-seq-p (s)
  ;; a list of pairs in decreasing order, such that
  ;; if (j . ?) is before (i . c) (and hence presumably i < j),
  ;; then p(i,j) <-> c.
  (and (extensible s)
       (prehom-seq s)))

(prove-lemma extensible-next-property (rewrite)
 (implies
  (extensible s)
  (and (lessp a (next a s))
       (prehom-seq-1 (next a s) s))))

(prove-lemma ramsey-seq-p-extends (rewrite)
  (implies (ramsey-seq-p s)
           (ramsey-seq-p (cons (next-pair s) s))))

(prove-lemma extensible-nlistp (rewrite)
  (IMPLIES (NOT (LISTP S))
           (EXTENSIBLE S))
  ((use (extensible (NEXT (add1 (ABOVE S)))))))

(prove-lemma ramsey-seq-p-nlistp (rewrite)
  (implies (nlistp s)
           (ramsey-seq-p s)))

(disable ramsey-seq-p)

(disable next-pair)

(defn ramsey-seq (n)
  ;; gives decreasing prehomogeneous seq. of length n -- largest
  ;; one is on the front!
  (if (zerop n)
      nil
    (cons (next-pair (ramsey-seq (sub1 n)))
          (ramsey-seq (sub1 n)))))

(prove-lemma ramsey-seq-has-ramsey-seq-p (rewrite)
  (ramsey-seq-p (ramsey-seq n)))

;; Now we want to cut down this prehomogenous sequence to one that’s
;; homogeneous.  First let’s define the flag.

(defn-sk good-color-p (c)
  ;; says that arbitrarily large elements of ramsey-seq agree with c
  (forall big
          (exists good-c-index
                  (and (lessp big good-c-index)
                       (equal c (cdr (car (ramsey-seq good-c-index))))))))

;; The following two lemmas are for the benefit of the proof-checker
;; macro-command SK*.

(prove-lemma good-color-p-suff (rewrite)
  (implies (and (lessp (big c) good-c-index)
                (equal c
                       (cdar (ramsey-seq good-c-index))))
           (good-color-p c)))
```

```
(prove-lemma good-color-p-necc (rewrite)
  (implies
   (not (and (lessp big (good-c-index big c))
             (equal c
                    (cdar (ramsey-seq (good-c-index big c))))))
   (not (good-color-p c)))
  ((use (good-color-p))
   (disable good-color-p)))

(defn good-c-index-wit (bound)
  (if (zerop bound)
      1
    (plus (big bound)
          (good-c-index-wit (sub1 bound)))))

(prove-lemma good-c-index-wit-positive (rewrite)
  (lessp 0 (good-c-index-wit bound)))

(prove-lemma lessp-big-good-c-index (rewrite)
  (implies (and (lessp 0 c)
                (not (lessp bound c)))
           (equal (lessp (big c) (good-c-index-wit bound))
                  t)))

(disable good-c-index-wit)

(defn color ()
  (cdar (ramsey-seq (good-c-index-wit (bound)))))

(prove-lemma ramsey-seq-has-colors-in-bounds (rewrite)
  (implies (lessp 0 n)
           (and (lessp 0 (cdar (ramsey-seq n)))
                (not (lessp (bound) (cdar (ramsey-seq n))))))
  ((enable next-pair next-color)))

(prove-lemma color-in-bounds (rewrite)
  (and (lessp 0 (color))
       (not (lessp (bound) (color)))))

(prove-lemma color-is-good (rewrite)
  (good-color-p (color))
  ((use (good-color-p-suff (good-c-index (good-c-index-wit (bound)))
                           (c (color))))))

(disable color)
(disable *1*color)

(defn ramsey-index (n)
  ;; returns the index of the nth member of ramsey-seq which has color (color)
  (if (zerop n)
      (good-c-index 0 (color))
    (good-c-index (ramsey-index (sub1 n)) (color))))

(prove-lemma color-properties (rewrite)
  (and (lessp big (good-c-index big (color)))
       (equal (cdr (car (ramsey-seq (good-c-index big (color)))))
              (color)))
  ((use (good-color-p-necc (c (color))))
   (disable good-color-p-necc)))

(prove-lemma ramsey-index-increasing (rewrite)
  (implies (lessp i j)
           (lessp (ramsey-index i) (ramsey-index j)))
  ((induct (plus j q))))

(defn ramsey (n)
  (car (car (ramsey-seq (ramsey-index n)))))
```

```
; Next goal:
;(prove-lemma ramsey-increasing (rewrite)
;  (implies (lessp i j)
;           (lessp (ramsey i) (ramsey j))))

(prove-lemma good-c-index-numberp (rewrite)
 (numberp (good-c-index big (color)))
 ((use (color-properties))
  (disable color-properties)))

(prove-lemma ramsey-index-numberp (rewrite)
  (numberp (ramsey-index n)))

(prove-lemma car-next-pair (rewrite)
  (equal (car (next-pair s))
         (next (caar s) s))
  ((enable next-pair)))

(prove-lemma ramsey-seq-extensible (rewrite)
  (extensible (ramsey-seq n)))

(prove-lemma next-not-zerop (rewrite)
  (implies (extensible s)
           (and (numberp (next a s))
                (not (equal (next a s) 0))))
 ((use (extensible-next-property))
  (disable extensible-next-property)))

(prove-lemma ramsey-seq-increasing (rewrite)
  (implies (lessp i j)
           (equal (lessp (caar (ramsey-seq i))
                         (caar (ramsey-seq j)))
                  t))
  ((induct (plus j q))))

(prove-lemma ramsey-index-increasing-rewrite (rewrite)
  (implies (lessp i j)
           (equal (lessp (ramsey-index i) (ramsey-index j)) t)))

(disable ramsey-index-increasing)

(prove-lemma ramsey-increasing nil
  (implies (lessp i j)
           (lessp (ramsey i) (ramsey j))))

;; Now we want to show that ramsey is homogeneous for (color):

;(prove-lemma ramsey-homogeneous (rewrite)
;  (implies (lessp i j)
;           (iff (p (ramsey i) (ramsey j))
;                (color))))

(defn restn
  ;; from Bill Bevier’s library
  (n l)
  (if (listp l)
      (if (zerop n)
          l
          (restn (sub1 n) (cdr l)))
      l))

;; We’ve already proved (ramsey-seq-p (ramseq-seq i)).  So, in order
;; to prove the key lemma ramsey-seq-prehom below, we’ll use this
;; together with an appropriate fact about restn and prehom seqs.

(prove-lemma ramsey-seq-restn-length (rewrite)
  (equal (restn n (ramsey-seq n))
         nil))
```

```
(prove-lemma prehom-seq-ramsey (rewrite)
  (prehom-seq (ramsey-seq n))
  ((use (ramsey-seq-has-ramsey-seq-p))
   (disable ramsey-seq-has-ramsey-seq-p)
   (enable ramsey-seq-p)))

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

(prove-lemma prehom-seq-1-restn (rewrite)
  (implies (and (prehom-seq-1 a s)
                (lessp x (length s)))
           (equal (p (caar (restn x s)) a)
                  (cdar (restn x s)))))

(prove-lemma prehom-seq-restn (rewrite)
  (implies (and (prehom-seq s)
                (lessp 0 x)
                (lessp x (length s)))
           (equal (p (caar (restn x s)) (caar s))
                  (cdar (restn x s)))))

(prove-lemma ramsey-seq-plus (rewrite)
  (equal (restn x (ramsey-seq (plus x y)))
         (ramsey-seq y)))

(prove-lemma plus-comm (rewrite)
             ;; added to raw version to help with lemma below
             (equal (plus x y) (plus y x)))

(prove-lemma ramsey-seq-plus-commuted (rewrite)
  (equal (restn x (ramsey-seq (plus y x)))
         (ramsey-seq y)))

(prove-lemma length-ramsey-seq (rewrite)
  (equal (length (ramsey-seq n)) (fix n)))

;; The lemmas from here to RAMSEY-SEQ-PREHOM were to eliminate the
;; proof-checker hints from that lemma.

(prove-lemma plus-difference-elim (elim)
  (implies (and (numberp j)
                (not (lessp j i)))
           (equal (plus i (difference j i))
                  j)))

(prove-lemma restn-difference-ramsey-seq (rewrite)
  (implies (and (lessp 0 i)
                (lessp i j))
           (equal (restn (difference j i)
                         (ramsey-seq j))
                  (ramsey-seq i))))

(prove-lemma prehom-seq-restn-commuted (rewrite)
  (implies (and (prehom-seq s)
                (lessp 0 x)
                (lessp x (length s)))
           (equal (p (caar s) (caar (restn x s)))
                  (cdar (restn x s))))
  ((use (prehom-seq-restn))
   (disable prehom-seq-restn)))
```

```
(prove-lemma ramsey-seq-prehom-lemma nil
             (implies (and (lessp 0 i) (lessp i j))
                      (equal (p (caar (restn (difference j i)
                                             (ramsey-seq j)))
                                (caar (ramsey-seq j)))
                             (cdar (restn (difference j i)
                                          (ramsey-seq j)))))
             ((disable restn-difference-ramsey-seq
                       ramsey-seq-plus-commuted)))

(prove-lemma ramsey-seq-prehom
             ;; originally done with proof-checker
             (rewrite)
             (implies (and (lessp 0 i) (lessp i j))
                      (equal (p (caar (ramsey-seq i))
                                (caar (ramsey-seq j)))
                             (cdar (ramsey-seq i))))
             ((use (ramsey-seq-prehom-lemma))))

(prove-lemma cdar-ramseq-seq-ramsey-index
             (rewrite)
             (equal (cdar (ramsey-seq (ramsey-index n)))
                    (color)))

(prove-lemma lessp-good-c-index (rewrite)
  (equal (lessp big
                (good-c-index big (color)))
         t))

(disable color-properties)

(prove-lemma good-c-index-non-zero (rewrite)
  (equal (lessp 0
                (good-c-index big (color)))
         t)
  ((use (lessp-good-c-index))
   (disable lessp-good-c-index)))

(prove-lemma ramsey-index-positive
             (rewrite)
             (lessp 0 (ramsey-index n))
             ((expand (ramsey-index n))))

(prove-lemma ramsey-seq-hom-lessp
             nil
             (implies (lessp i j)
                      (equal (p (ramsey i) (ramsey j))
                             (color))))

(prove-lemma p-num-is-p (rewrite)
  (implies (and (numberp x) (numberp y))
           (equal (p-num x y) (p x y)))
  ((enable p)))

(prove-lemma numberp-ramsey (rewrite)
  (numberp (caar (ramsey-seq (ramsey-index n))))
  ((enable next-pair)
   (expand (ramsey-seq (ramsey-index n)))))

(prove-lemma ramsey-seq-hom
             (rewrite)
             (implies (and (numberp i) (numberp j) (not (equal i j)))
                      (equal (p-num (ramsey i) (ramsey j))
                             (color)))
             ((use (ramsey-seq-hom-lessp)
                   (ramsey-seq-hom-lessp (i j) (j i)))))
```

```
;; The above, together with what was already proved, i.e.

;; (prove-lemma ramsey-increasing (rewrite)
;;  (implies (lessp i j)
;;           (lessp (ramsey i) (ramsey j))))

;; and the fact that p-num was arbitrary (and there are no add-axioms)
;; finishes the job.
```

Appendix F
Schroeder-Bernstein Theorem Events List

The following events go through in a version of the proof-checker (PC-NQTHM) enhancement of the Boyer-Moore prover, as documented in [5] and [6], extended by the functional variables work [2] and DEFN-SK. All text on a line following a semicolon ‘;’ is a comment.

```
;;; The proof in here is based on the hint to exercise (8) on
;;; page 43 of Kunen’s Set Theory book.

(defn id (x) x)

(constrain fa-and-fb-are-one-one (rewrite)
  (and
   ;; fa is one-to-one
   (implies (and (a x) (a y) (not (equal x y)))
            (not (equal (fa x) (fa y))))
   ;; fb is one-to-one
   (implies (and (b x) (b y) (not (equal x y)))
            (not (equal (fb x) (fb y))))
   ;; the image of fa on a is contained in b
   (implies (a x) (b (fa x)))
   ;; the image of fb on b is contained in a
   (implies (b x) (a (fb x)))
   (or (truep (a x)) (falsep (a x)))
   (or (truep (b x)) (falsep (b x))))
  ;; let fa and fb be the identity function and let a and b be the universe
  ((fa id)
   (fb id)
   (a (lambda (x) t)) (b (lambda (x) t))))

(defn-sk in-fa-range (x)
  (exists fa-1 (and (a fa-1)
                    (equal (fa fa-1) x))))

;; The following 3 events were generated automatically to help with
;; the proof-checker’s application of the macro command SK*.

(disable in-fa-range)

(prove-lemma in-fa-range-suff
             (rewrite)
             (implies (and (a fa-1) (equal (fa fa-1) x))
                      (in-fa-range x))
             ((use (in-fa-range))))

(prove-lemma in-fa-range-necc
             (rewrite)
             (implies (not (and (a (fa-1 x))
                                (equal (fa (fa-1 x)) x)))
                      (not (in-fa-range x)))
             ((use (in-fa-range))))

(defn-sk in-fb-range (x)
  (exists fb-1 (and (b fb-1)
                    (equal (fb fb-1) x))))

;; The following 3 events were generated automatically to help with
;; the proof-checker’s application of the macro command sk*.

(disable in-fb-range)
```

```
(prove-lemma in-fb-range-suff
             (rewrite)
             (implies (and (b fb-1) (equal (fb fb-1) x))
                      (in-fb-range x))
             ((use (in-fb-range))))

(prove-lemma in-fb-range-necc
             (rewrite)
             (implies (not (and (b (fb-1 x))
                                (equal (fb (fb-1 x)) x)))
                      (not (in-fb-range x)))
             ((use (in-fb-range))))

(defn circled (flg x n)
  ;; If flg is 'a, returns t iff x is in a-n in the sense of Kunen's proof.
  ;; Similarly for b if flg is not 'a.
  (if (equal flg 'a)
      (if (zerop n)
          (a x)
        (and (in-fb-range x)
             (circled 'b (fb-1 x) (sub1 n))))
    (if (zerop n)
        (b x)
      (and (in-fa-range x)
           (circled 'a (fa-1 x) (sub1 n))))))

(defn-sk a-core (x)
  ;; introduced "inductively" so that the level will be tight
  (and (a x)
       (forall a-level
               (implies (and (numberp a-level)
                             (circled 'a x a-level))
                        (circled 'a x (add1 a-level))))))

(disable a-core)

(prove-lemma a-core-necc-base (rewrite)
  (implies (and (zerop n)
                (a-core x))
           (circled 'a x n))
  ((use (a-core))))

(prove-lemma a-core-necc-induction (rewrite)
  (implies (and (not (zerop n))
                (a-core x)
                (circled 'a x (sub1 n)))
           (circled 'a x n))
  ((use (a-core (a-level (sub1 n))))))

(prove-lemma a-core-necc (rewrite)
  ;; the conclusion implies the more obvious consequence of a-core
  (implies (not (circled 'a x n))
           (not (a-core x)))
  ((disable circled)
   (induct (plus n q))))

(disable a-core-necc-base)
(disable a-core-necc-induction)

(prove-lemma a-core-suff (rewrite)
     (implies (and (a x)
                   (implies (and (numberp (a-level x))
                                 (circled 'a x (a-level x)))
                            (circled 'a x (add1 (a-level x)))))
              (a-core x))
     ((use (a-core))))
```

```
(defn-sk b-core (x)
  ;; introduced "inductively" so that the level will be tight
  (and (b x)
       (forall b-level
               (implies (and (numberp b-level)
                             (circled ’b x b-level))
                        (circled ’b x (add1 b-level))))))

(disable b-core)

(prove-lemma b-core-necc-base (rewrite)
  (implies (and (zerop n)
                (b-core x))
           (circled ’b x n))
  ((use (b-core))))

(prove-lemma b-core-necc-induction (rewrite)
  (implies (and (not (zerop n))
                (b-core x)
                (circled ’b x (sub1 n)))
           (circled ’b x n))
  ((use (b-core (b-level (sub1 n))))))

(prove-lemma b-core-necc (rewrite)
    ;; the conclusion implies the more obvious consequence of b-core
  (implies (not (circled ’b x n))
           (not (b-core x)))
  ((disable circled)
   (induct (plus n q))))

(disable b-core-necc-base)
(disable b-core-necc-induction)

(prove-lemma b-core-suff (rewrite)
     (implies (and (b x)
                   (implies (and (numberp (b-level x))
                                 (circled ’b x (b-level x)))
                            (circled ’b x (add1 (b-level x)))))
              (b-core x))
     ((use (b-core))))

(defn parity (n)
  (if (zerop n)
      t
    (not (parity (sub1 n)))))

(defn j (x)
  ;; the isomorphism
  (if (or (a-core x)
          (parity (a-level x)))
      (fa x)
    (fb-1 x)))

(defn j-1 (y)
  ;; the isomorphism’s inverse
  (if (or (b-core y)
          (not (parity (b-level y))))
      (fa-1 y)
    (fb y)))
```

```
; Our goals:
;(prove-lemma j-1-j (rewrite)
;  (implies (a x)
;           (equal (j-1 (j x)) x)))

;(prove-lemma j-j-1 (rewrite)
;  (implies (b y)
;           (equal (j (j-1 y)) y)))

;; We’ll start on the first of these.  The theorem-prover output
;; suggests the following lemma:

;(prove-lemma b-core-fa (rewrite)
;  (implies (a x)
;           (iff (b-core (fa x))
;                (a-core x))))

;; A main lemma is that (a-core x) => (b-core (fa x)) for x in a.

(prove-lemma fa-1-inverts-fa
             ;; originally with proof-checker
             (rewrite)
             (implies (a x)
                      (equal (fa-1 (fa x)) x))
             ((use (in-fa-range (x (fa x)) (fa-1 x)))))

;; The following is useful for proof-checker rewriting:

(prove-lemma in-fa-range-fa (rewrite)
   (implies (a x)
            (in-fa-range (fa x))))

(prove-lemma b-core-fa
           ;; originally with proof-checker
           (rewrite)
           (implies (a x)
                    (iff (b-core (fa x)) (a-core x)))
           ((use (a-core-suff (x x))
                 (b-core-necc (n (add1 (add1 (a-level x)))) (x (fa x)))
                 (a-core-necc (n (if (numberp (b-level (fa x)))
                                     (b-level (fa x))
                                     0))
                              (x x))
                 (b-core-suff (x (fa x))))))

;; On to Case 2 of j-1-j.  We find there the following contradictory hyps:

;(AND (A X)
;     (PARITY (A-LEVEL X))
;     (NOT (A-CORE X))
;     (PARITY (B-LEVEL (FA X))))

;; We want to prove:

;; B-LEVEL-FA

;(IMPLIES (AND (A X) (NOT (A-CORE X)))
;         (EQUAL (B-LEVEL (FA X))
;                (ADD1 (A-LEVEL X))))

(prove-lemma b-fa-equality-rewrite
             (rewrite)
             (implies (and (a x) (not (b y)))
                      (not (equal (fa x) y))))
```

```
(prove-lemma fa-range-contained-in-b
             ;; originally done with proof-checker;
             ;; proved B-FA-EQUALITY-REWRITE to help
             (rewrite)
             (implies (not (b y))
                      (not (in-fa-range y)))
             ((use (in-fa-range (x y)))))

(prove-lemma circled-implies-b (rewrite)
  (implies (circled 'b y n)
           (b y)))

(prove-lemma a-fb-equality-rewrite
             (rewrite)
             (implies (and (b y) (not (a x)))
                      (not (equal (fb y) x))))

(prove-lemma fb-range-contained-in-a
             ;; originally done with proof-checker;
             ;; proved A-FB-EQUALITY-REWRITE to help
             (rewrite)
             (implies (not (a x))
                      (not (in-fb-range x)))
             ((use (in-fb-range))))

(prove-lemma circled-implies-a (rewrite)
  (implies (circled 'a y n)
           (a y)))

(prove-lemma circled-monotone (rewrite)
  (implies (and (circled flg x j)
                (not (lessp j i)))
           (circled flg x i)))

(prove-lemma circled-b-fa
             (rewrite)
             (implies (a x)
                      (equal (circled 'b (fa x) n)
                             (circled 'a x (sub1 n)))))

(prove-lemma b-level-fa-hack-lemma nil
  (implies (numberp (b-level (fa x)))
           (equal (equal (b-level (fa x))
                         (add1 (a-level x)))
                  (not (or (lessp (b-level (fa x))
                                  (add1 (a-level x)))
                           (lessp (add1 (a-level x))
                                  (b-level (fa x))))))))

(prove-lemma b-level-fa
 ;; originally done by proof-checker
 (rewrite)
 (implies (and (a x) (not (a-core x)))
          (equal (b-level (fa x))
                 (add1 (a-level x))))
 ((use (a-core-suff)
       (b-core-suff (x (fa x)))
       (b-level-fa-hack-lemma))
  (disable circled)))
```

```
;; On to Case 3 of j-1-j.  Our first subgoal is:

;(IMPLIES (AND (A X)
;              (NOT (A-CORE X))
;              (NOT (PARITY (A-LEVEL X)))
;              (NOT (B-CORE (FB-1 X)))
;              (PARITY (B-LEVEL (FB-1 X))))
;         (EQUAL (FB (FB-1 X)) X))

;; and this follows from the following two lemmas.

(prove-lemma not-parity-a-level-implies-in-fb-range
             ;; I tried an automatic proof with lots of hints, but failed
             (rewrite)
             (implies (and (a x) (not (parity (a-level x))))
                      (in-fb-range x))
             ((instructions promote
                            (claim (not (equal (a-level x) 0)))
                            (use-lemma a-core
                                       ((a-level (a-level x)) (x x)))
                            (claim (circled ’a x (a-level x))
                                   ((expand (circled ’a x (a-level x)))))
                            (drop 4)
                            (prove (expand (circled ’a x (a-level x)))))))

;; The next subcase of j-1-j is:

;(implies (and (a x)
;              (not (a-core x))
;              (not (parity (a-level x)))
;              (not (parity (b-level (fb-1 x)))))
;         (equal (fa-1 (fb-1 x)) x))

;; and this follows from:

;(prove-lemma b-level-fb-1 (rewrite)
;  (implies (and (a x)
;                (not (a-core x))
;                (in-fb-range x))
;           (equal (b-level (fb-1 x))
;                  (sub1 (a-level x)))))

;; I’ve proved nearly the analogous thing already for fa.  Let’s prove that
;; one for fb and then deduce this from it.

(prove-lemma fb-1-inverts-fb
             ;; originally done with proof-checker
             (rewrite)
             (implies (b x)
                      (equal (fb-1 (fb x)) x))
             ((use (in-fb-range (x (fb x)) (fb-1 x)))))

(prove-lemma in-fb-range-fb
             (rewrite)
             (implies (b x) (in-fb-range (fb x))))

(prove-lemma circled-a-fb
             (rewrite)
             (implies (b x)
                      (equal (circled ’a (fb x) n)
                             (circled ’b x (sub1 n)))))

(prove-lemma a-level-fb-hack-lemma nil
  (implies (numberp (a-level (fb x)))
           (equal (equal (a-level (fb x))
                         (add1 (b-level x)))
                  (not (or (lessp (a-level (fb x))
                                  (add1 (b-level x)))
                           (lessp (add1 (b-level x))
                                  (a-level (fb x))))))))
```

```
(prove-lemma a-core-fb
           ;; done originally with proof-checker
           (rewrite)
           (implies (b x)
                    (iff (a-core (fb x)) (b-core x)))
           ((use (b-core-suff (x x))
                 (a-core-necc (n (add1 (add1 (b-level x)))) (x (fb x)))
                 (b-core-necc (n (if (numberp (a-level (fb x)))
                                     (a-level (fb x))
                                     0))
                              (x x))
                 (a-core-suff (x (fb x))))))

(prove-lemma a-level-fb
 (rewrite)
 (implies (and (b x) (not (b-core x)))
          (equal (a-level (fb x))
                 (add1 (b-level x))))
 ((use (b-core-suff)
       (a-core-suff (x (fb x)))
       (a-level-fb-hack-lemma))
  (disable circled)))

(prove-lemma b-level-fb-1
             ;; equality reasoning was awkward for me here so I used the proof-checker
             (rewrite)
             (implies (and (a x)
                           (not (a-core x))
                           (in-fb-range x))
                      (equal (b-level (fb-1 x))
                             (sub1 (a-level x))))

             ;((use (a-level-fb (x (fb-1 x)))
             ;      (fb-fb-1))
             ; (disable a-level-fb fb-fb-1))

             ((instructions (use-lemma a-level-fb ((x (fb-1 x))))
                            promote
                            (claim (equal x (fb (fb-1 x))))
                            (claim (b (fb-1 x)))
                            (generalize (((fb-1 x) y)))
                            prove)))

;; It remains only to prove the following subcase, and then we're done with j-1-j:

;(implies (and (a x)
;              (not (a-core x))
;              (not (parity (a-level x)))
;              (b-core (fb-1 x)))
;         (equal (fa-1 (fb-1 x)) x))

;; I've already proved a-core-fb, and this should be useful.

(prove-lemma b-core-implies-b (rewrite)
  (implies (b-core x) (b x))
  ((use (b-core))
   (disable b-core-suff b-core-necc)))

(prove-lemma fb-fb-1 (rewrite)
  (implies (and (a x)
                (in-fb-range x))
           (equal (fb (fb-1 x)) x)))
```

```
(prove-lemma b-core-fb-1 (rewrite)
             (implies (and (a x)
                           (not (a-core x))
                           (not (parity (a-level x))))
                      (not (b-core (fb-1 x))))
             ((instructions promote
                            (contradict 2)
                            (= x (fb (fb-1 x)))
                            (drop 1 3)
                            (generalize (((fb-1 x) y)))
                            prove)))

;; Now finally for our first main goal:

(prove-lemma j-1-j (rewrite)
  (implies (a x)
           (equal (j-1 (j x)) x)))

;;;; We're ready now for the converse.  Here's the first sticking point:

;(implies (and (b-core y)
;              (a-core (fa-1 y)))
;         (equal (fa (fa-1 y)) y))

;; We need only the following two easy lemmas:

(prove-lemma b-core-implies-in-fa-range (rewrite)
  ;; instantiations found easily using sk* and put in proof-checker
  (implies (b-core y)
           (in-fa-range y))
  ((use (in-fa-range-suff (fa-1 (fa-1 y)) (x y))
        (b-core-necc (n 1) (x y)))))

(prove-lemma fa-fa-1 (rewrite)
  (implies (in-fa-range y)
           (equal (fa (fa-1 y)) y))
  ((use (in-fa-range (x y)))))

;; Our next goal is:

;(implies (and (b-core y)
;              (not (a-core (fa-1 y)))
;              (not (parity (a-level (fa-1 y)))))
;         (equal (fb-1 (fa-1 y)) y))

;;; which is taken care of simply by:

(prove-lemma a-core-fa-1
             (rewrite)
             (implies (in-fa-range y)
                      (iff (a-core (fa-1 y)) (b-core y)))
             ((instructions promote
                            (dive 2)
                            (= y (fa (fa-1 y)))
                            top
                            (claim (a (fa-1 y)))
                            (generalize (((fa-1 y) x)))
                            prove)))
```

```
;; next we have to prove

;(implies (and (b y)
;               (not (parity (b-level y)))
;               (a-core (fa-1 y)))
;          (equal (fa (fa-1 y)) y))

;; which follows from fa-fa-1 together with an obvious analog of
;; NOT-PARITY-A-LEVEL-IMPLIES-IN-FB-RANGE:

(prove-lemma not-parity-b-level-implies-in-fa-range
             ;; simply modified from NOT-PARITY-A-LEVEL-IMPLIES-IN-FB-RANGE:
             (rewrite)
             (implies (and (b y) (not (parity (b-level y))))
                      (in-fa-range y))
             ((instructions promote
                            (claim (not (equal (b-level y) 0)))
                            (use-lemma b-core
                                       ((b-level (b-level y)) (x y)))
                            (claim (circled 'b y (b-level y))
                                   ((expand (circled 'b y (b-level y)))))
                            (drop 4)
                            (prove (expand (circled 'b y (b-level y)))))))

;; It remains only to prove:

;(implies (and (b y)
;               (not (parity (b-level y)))
;               (not (b-core y))
;               (not (parity (a-level (fa-1 y)))))
;          (equal (fb-1 (fa-1 y)) y))

;; which follows from an analogue of B-LEVEL-FB-1:

(prove-lemma a-level-fa-1
             (rewrite)
             (implies (and (b y)
                           (not (b-core y))
                           (in-fa-range y))
                      (equal (a-level (fa-1 y))
                             (sub1 (b-level y))))
             ((instructions (use-lemma b-level-fa ((x (fa-1 y))))
                            promote
                            (claim (equal y (fa (fa-1 y))))
                            (claim (a (fa-1 y)))
                            (generalize (((fa-1 y) x)))
                            prove)))

;;; and we're done!!!

(prove-lemma j-j-1 (rewrite)
  (implies (b y)
           (equal (j (j-1 y)) y)))
```

```
;; To summarize:

;; From the axiom saying that fa maps a one-one into b and fb maps b one-one into a,

;(constrain fa-and-fb-are-one-one (rewrite)
;  (and (implies (and (a x) (a y) (not (equal x y)))
;                (not (equal (fa x) (fa y))))
;       (implies (and (b x) (b y) (not (equal x y)))
;                (not (equal (fb x) (fb y))))
;       (implies (a x) (b (fa x)))
;       (implies (b x) (a (fb x)))
;       (or (truep (a x)) (falsep (a x)))
;       (or (truep (b x)) (falsep (b x))))
;  ((fa id) (fb id) (a (lambda (x) t)) (b (lambda (x) t))))

;; to finish this off we simply prove the obvious lemmas needed for
;; j-iso below.  For the first, j-range,
;; we need a lemma (as seen from observing the failed proof transcript).

(prove-lemma in-fb-range-implies-b-fb-1 (rewrite)
  (implies (in-fb-range y)
           (b (fb-1 y))))

(prove-lemma j-range (rewrite)
  (implies (a x)
           (b (j x))))

(prove-lemma in-fa-range-implies-a-fa-1 (rewrite)
  (implies (in-fa-range y)
           (a (fa-1 y))))

(prove-lemma j-1-range (rewrite)
  (implies (b x)
           (a (j-1 x))))

(disable j)
(disable j-1)

(prove-lemma j-is-one-one
             ;; I tried briefly to prove a rewrite rule to help
             ;; with the CLAIM, but didn't try very hard.
             (rewrite)
             (implies (and (a x1)
                           (a x2)
                           (not (equal x1 x2)))
                      (not (equal (j x1) (j x2))))
             ((instructions promote
                            (contradict 3)
                            (claim (equal (j-1 (j x1)) (j-1 (j x2))))
                            (drop 3)
                            (prove))))
```

```
;; we were able to conservatively extend the theory culminating in
;; definitions of a function j which maps a one-one into b:

(defn-sk j-iso nil
  (and
   ;; j maps a into b
   (forall x
           (implies (a x)
                    (b (j x))))
   ;; j is one-one
   (forall (x1 x2)
           (implies (and (a x1)
                         (a x2)
                         (equal (j x1) (j x2)))
                    (equal x1 x2)))
   ;; j is onto
   (forall y (implies (b y)
                      (exists x (and (a x) ;the only mod -- Thanks, Bob
                                     (equal (j x) y))))))
  ((prefix j-iso-)))

(prove-lemma j-is-an-isomorphism nil
             (j-iso)
             ((use (j-iso (x (j-1 (j-iso-y)))))
              (disable j-iso)))
```

References

1. R. S. Boyer and J S. Moore, *A Computational Logic Handbook*, Academic Press, Boston, 1988.
2. Robert S. Boyer, David M. Goldschlag, Matt Kaufmann, and J Strother Moore, ‘‘Functional Instantiation in First Order Logic’’, Tech. report 44, Computational Logic, Inc., May 1989.
3. J. R. Shoenfield, *Mathematical Logic*, Addison-Wesley, Reading, Ma., 1967.
4. D. de Champeaux, ‘‘Subproblem Finder and Instance Checker, Two Cooperating Modules for Theorem Provers’’, *J. Assoc. for Comp. Mach.*, Vol. 33, October 1986, pp. 633-657.
5. Matt Kaufmann, ‘‘A User’s Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover’’, Tech. report 19, Computational Logic, Inc., May 1988.
6. Matt Kaufmann, ‘‘Addition of Free Variables to an Interactive Enhancement of the Boyer-Moore Theorem Prover’’, Tech. report 42, Computational Logic, Inc., May 1989.
7. Kenneth Kunen, *Set Theory: An Introduction to Independence Proofs*, North-Holland, New York, 1980.
8. Matt Kaufmann, ‘‘Another Note on Skolemization’’, Internal Note 127, Computational Logic, Inc., March 1989.

Table of Contents