Quantification in Nqthm:
a Recognizer and Some
Constructive Implementations

Matt Kaufmann

Technical Report 81 August, 1992

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

This research was supported in part by ONR Contract N00014-91-C-0130. The views and conclusions contained in this document are those of the author(s) 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.

ABSTRACT: We present an implementation of a *recognizer* for quantified notions in the Boyer-Moore Theorem Prover, Nqthm. That is, we provide a method for checking that a given function does indeed represent a quantified notion. We also present methods for *generating* constructively-presented functions that represent quantified notions, including definitions using only bounded quantifiers.

1. Introduction

It is often the case that users of Nqthm [1, 2] want to be able to express first-order quantified notions in Nqthm, or in Pc-Nqthm [3, 4].[1] Quantifiers can be useful in creating elegant specifications: from a specification standpoint it is sometimes more appropriate to assert that a certain value exists, than to specify that a particular ‘‘witnessing’’ function (Skolem function) serves to provide that value. In some cases, the quantifiers are in fact necessary both for the specification and the proof; see for example the discussion of Ramsey’s theorem in [5].

In fact, Pc-Nqthm provides the DEFN-SK feature [5] as a direct implementation of first-order quantification, but not all Nqthm users want to use DEFN-SK for representing quantified notions. For example, both Nqthm and Pc-Nqthm users often prefer to use recursive functions to model bounded quantification, for at least two reasons. For one, they may prefer constructivity either on ‘‘ethical’’ grounds of parsimony or perhaps because they want to execute their forms. Second, it is often useful to take advantage of the prover’s careful heuristics for dealing with recursive functions rather than its relatively weak heuristics for dealing with free variables.

Nqthm users know that it is often quite possible to define constructive ‘‘Skolem functions’’ to witness ordinary (unbounded) first-order quantifiers. For example, Yuan Yu has informed us of the existence of an unbounded quantifier in his MC68020 work that is really equivalent to a bounded one.

There are two goals in the present work. One goal is to implement a *recognizer* for quantified notions, i.e. some kind of predicate that can inform the user when he has implemented a quantified notion correctly. The other goal is to implement various utilities that *generate* quantified notions. We will deal with these two goals in the two respective sections that follow.

Thus in the first section below we present a new macro CHECK-QUANT that tells us when we have correctly implemented a quantified notion. Here is a brief introduction by way of example. Consider the form

```
(check-quant forall-p4 (i j y)
  (forall x (implies (and (numberp x)
                          (not (lessp x i))
                          (lessp x j))
                     (p4 x i j y)))
  forall-p4-intro)
```

Here, loosely speaking, this form returns *true* if and only if the system verifies that in the current theory, with the specific assistance of the event forall-p4-intro, it is the case that the existing function **forall-p4** has the property that

```
∀ i ∀ j ∀ y
  [(forall-p4 i j y)
   ↔
   ∀ x (implies (and (numberp x)
                     (not (lessp x i))
                     (lessp x j))
                   (p4 x i j y))]
```

[1] We assume familiary with Nqthm and Pc-Nqthm in this report.

The second section presents two new tools for generating representations of quantified notions. First we present DEFN-SK-CONSTRUCTIVE, a macro similar to DEFN-SK except that it requires witnessing functions and then generates an appropriate CONSTRAIN event [6]. Then we present an implementation BDD-FORALL of bounded universal quantification.

The first appendix is a transcript of a session using BDD-FORALL. The second appendix contains the Lisp code.

2. Recognizing quantified notions.

In this section we address the following question: How do we know when we have correctly represented a quantified concept? This question is important, since there are occasions when Nqthm users want to *present* their work in terms of quantifiers but do not want to use quantifiers explicitly to *do* their work.

We present a utility CHECK-QUANT that has been created to certify that a given function does indeed correctly represent a quantified notion in a given history. Let us be a bit more precise. Let ϕ be a first-order formula with free variables included in the set **`{x1 ... xn}`**. Also, let **`f`** be a function symbol with argument list **`(x1 ... xn)`**. The predicate CHECK-QUANT tells us when it is the case that ϕ is logically equivalent to **`(f x1 ... xn)`** in the current history. More precisely, CHECK-QUANT is conservative in the same sense that the Nqthm prover is conservative: if it answers affirmatively then this logical equivalence holds, but if not then we simply don't know.

We begin with an example illustrating the use of CHECK-QUANT as well as an associated utility CHECK-QUANT-HELP. The second subsection contains documentation and a proof of a specification of CHECK-QUANT.

2.1 A sample use of CHECK-QUANT

Consider the following rather standard definition of the property that **`x`** is a list of natural numbers.

```
(defn all-naturals (x)
  (if (listp x)
      (and (numberp (car x))
           (all-naturals (cdr x)))
    t))
```

In this simple example it is obvious that **`(all-naturals x)`** is equivalent to the formula

```
(forall a (implies (member a x) (numberp a)))
```

Now if we were to *define* a notion based on this quantified formula, using DEFN-SK, we might write

```
(defn-sk all-naturals-quant (x)
  (forall a (implies (member a x) (numberp a))))
```

This DEFN-SK event generates the Skolem axiom

```
(AND (IMPLIES (IMPLIES (MEMBER (A X) X)
                       (NUMBERP (A X)))
              (ALL-NATURALS-QUANT X))
     (IMPLIES (NOT (IMPLIES (MEMBER A X)
                            (NUMBERP A)))
              (NOT (ALL-NATURALS-QUANT X))))
```

However, rather than submit such a DEFN-SK event, suppose we prefer to use the original **`all-naturals`** function. Let us introduce a function that plays the role of the function **`A`** in the term above.[2] Intuitively, this function does its best at finding a counterexample to the universally quantified formula: if any counterexample exists, then this function should find it.

```
(defn bad-guy (x)
  (if (listp x)
      (if (not (numberp (car x)))
          (car x)
        (bad-guy (cdr x)))
    0))
```

Now we can *prove* a version of the Skolem axiom displayed above.

```
(prove-lemma all-naturals-is-correct ()
  (and (implies (implies (member (bad-guy x) x)
                         (numberp (bad-guy x)))
                (all-naturals x))
       (implies (not (implies (member a x)
                              (numberp a)))
                (not (all-naturals x)))))
```

Finally, we can assert that we have indeed defined a quantified notion as claimed. The following form returns T if the equivalence of **`(all-naturals x)`** with the indicated quantified formula can be seen to be a theorem of the current theory, with the help of the lemma ALL-NATURALS-IS-CORRECT above. We will see below that this equivalence guarantees that the function **`all-naturals`** correctly represents the indicated quantified notion.

```
>(check-quant all-naturals (x)
  (forall a (implies (member a x) (numberp a)))
  all-naturals-is-correct)
T

>
```

Finally, let us mention a utility that can be helpful in conjunction with CHECK-QUANT, called CHECK-QUANT-HELP. Imagine that one want to represent the quantified notion discussed above, **`(forall a (implies (member a x) (numberp a)))`**, but has not yet defined the ‘‘Skolem function’’ **`bad-guy`**. Then one could use CHECK-QUANT-HELP as follows:

```
(check-quant-help all-naturals (x)
  (forall a (implies (member a x) (numberp a))))
```

The system responds as follows.

---

[2] Ken Kunen suggested this trick to us some time ago.

```
The new function symbol with its argument list is (A-1 X).

(AND (IMPLIES (IMPLIES (MEMBER (A-1 X) X) (NUMBERP (A-1 X)))
              (ALL-NATURALS X))
     (IMPLIES (NOT (IMPLIES (MEMBER A X) (NUMBERP A)))
              (NOT (ALL-NATURALS X))))
```

We may then proceed to define **bad-guy** to have the property that **a-1** has in the term above. Alternatively, CHECK-QUANT may take a final argument that is a functional substitution, whose domain should contain exactly the new function symbols reported by CHECK-QUANT-HELP. In this case, then, we could submit the following form.

```
(check-quant all-naturals (x)
  (forall a (implies (member a x) (numberp a)))
  ((a-1 bad-guy)))
```

The theorem prover then checks that the functional instance of the term printed above by CHECK-QUANT-HELP, using the functional substitution **((a-1 bad-guy))**, is a theorem and returns **T** if and only if the proof succeeds.

2.2 Precise description of CHECK-QUANT and theoretical justification

In this subsection we begin by describing the usage of CHECK-QUANT, including a criterion that we guarantee will be met when CHECK-QUANT returns **T**. Then we move to the realm of formal logic and present a precise notion of what it means for a function to *represent* a quantified notion. Finally, we connect up the CHECK-QUANT utility and our logical notion of *represents* by showing that when the CHECK-QUANT criterion is met, then our notion of *represents* holds for the given function symbol and formula.

The usage of CHECK-QUANT is as follows.

(check-quant *function arguments*
  formula
  *previous-event-name***)**

Here, *function* should be a function symbol in the current history, and *arguments* should be a list of distinct variables that includes all free variables of *formula* and whose length equals the arity of *function*. *previous-event-name* should be the name of a previous event (except, see below for an exception). If this form returns **T** then the following condition must be met (but not necessarily conversely); otherwise this form returns **NIL**. (See [6] for a discussion of functional instances and functional substitutions.)

CHECK-QUANT CRITERION:

A Skolemization of the equivalence

(**iff** (*function . arguments*)
  formula)

is a functional instance of the Nqthm term[3] associated with *previous-event-name*.

[3] technically, the FORMULA-OF

Finally, as a convenience we allow *previous-event-name* to be a functional substitution *fs* instead of an event name, as long as certain conditions are met. Let ψ be the Skolemization given by DEFN-SK of the formula (**`iff`** (*function* . *arguments*) *formula*) displayed above. Then the domain of *fs* must consist of the Skolem functions generated by DEFN-SK in producing ψ, and its range should contain only symbols[4]. In addition, application of *f* to ψ, i.e. $\psi \setminus fs$ must be a theorem of the given history. (See [6] for more about functional substitutions.)

For the rest of this section we will consider only the case above where *previous-event-name* is a previous event name, rather than a functional substitution. The proof of the theorem below in the other case follows easily from the theorem we prove, and is left to the reader.

Let us now state just what it means for CHECK-QUANT to be a *correct* recognizer for first-order notions.

Definition. Let Γ be a first-order theory (set of sentences), let ϕ be a first-order sentence, let $\mathbf{f}$ be a function symbol not occurring in ϕ, and let $\mathbf{x_1}$, ..., $\mathbf{x_n}$ be distinct variables including all the free variables of ϕ, where $\mathbf{n}$ is the arity of $\mathbf{f}$. We say that $\mathbf{f(x_1,...,x_n)}$ *represents* ϕ *in* Γ if Γ |- $[f(x_1,...,x_n) \leftrightarrow \phi]$.

An important consequence of this notion of *represents*, which we will not need here, is the following trivial corollary. It is simply a formal statement asserting that ''represents'' has the desired property, namely that the given first-order formula can be replaced by the corresponding function application.

Proposition. Suppose that $\mathbf{f(x_1,...,x_n)}$ represents ϕ *in* Γ. Then for any substitution $\mathbf{s}$, Γ |- $[f(\mathbf{s}(x_1),...,\mathbf{s}(x_n)) \leftrightarrow \phi/\mathbf{s}]$ (where we define $\mathbf{s(x)} = \mathbf{x}$ for all $\mathbf{x}$ not in the domain of $\mathbf{s}$). -|

For the rest of this paper, functional substitutions will always associate function symbols (rather than LAMBDA terms) with function symbols.

Here is the theorem stating correctness of our approach. We think of the theory Γ below as being the set of theorems of the current history, and of $\mathbf{P}$ as being the formula associated with some existing event, i.e. of what we called *previous-event-name* above. In the theorem below, think of $\mathbf{fs}$ as substituting only for function symbols that do not occur in the current history, i.e. for the ones generated by the indicated Skolemization.

Theorem. Suppose that $\mathbf{f}$ is a function symbol, $\mathbf{x_1,...,x_n}$ is a list of distinct variables where $\mathbf{n}$ equals the arity of $\mathbf{f}$, and ψ is a Skolemization of the following equivalence:

$$\mathbf{(iff\ (f\ x_1\ ...\ x_n)\ \phi)}$$

where ϕ is a formula whose free variables all belong to $\mathbf{\{x_1\ ..\ x_n\}}$. Also let $\mathbf{P}$ be a theorem of Γ, and suppose that $\mathbf{fs}$ is a functional substitution such that $\psi\backslash\mathbf{fs}$ is $\mathbf{P}$, where the domain of $\mathbf{fs}$ consists of function symbols not occurring in ϕ, $\{\mathbf{f}\}$, or Γ. Then $\mathbf{f(x_1,...,x_n)}$ represents ϕ *in* Γ.

Proof. We have to show that Γ |- $[f(x_1,...,x_n) \leftrightarrow \phi]$. Now a basic property of Skolemization (see for example Lemma 2(2) of [7], or Proposition 1 in the second appendix of the technical report version of [5]) is that each Skolemization (actually its universal closure) implies the formula from which it comes. It is routine to check that in fact, this argument does not depend on the choice of Skolem function symbols. It is then clear then that $\psi\backslash\mathbf{fs}$ may itself be viewed as a Skolemization (in this more general sense) of the equivalence displayed above. Therefore, by the above property of Skolemization, since $\psi\backslash\mathbf{fs}$ is a

[4]Perhaps this latter restriction can be relaxed, but we haven't bothered to do so.

theorem of Γ, therefore so is the equivalence displayed above.

Here is an alternate, model-theoretic argument. We know that it is logically valid that $\psi' \rightarrow$ **equiv**, where **equiv** is the equivalence displayed above and ψ' is the universal closure of ψ. But logical validity is preserved by functional substitutions in which the range is disjoint from the domain, for if the negation of a sentence **A\fs** has a model *M*, then the negation of **A** has a model: simply expand *M* to interpret the function symbols in the domain of **fs** by the interpretations in *M* of what **fs** maps them to. Therefore, it is also logically valid that **(**$\psi' \rightarrow$ **equiv)\fs**, i.e. that **((**ψ**\fs)'** $\rightarrow$ **equiv)**. -|

3. Generating quantified notions

Recall that DEFN-SK [5] allows the user to introduce quantified notions into the Boyer-Moore paradigm. In the preceding section we introduced a recognizer for when one has correctly *represented* quantified notions, and in fact we believe that DEFN-SK always produces functions that represent desired quantified notions. For example, after the event

```
(defn-sk all-naturals-quant (x)
  (forall a (implies (member a x) (numberp a))))
```

we obtain a result of **T** upon evaluation of the form

```
(check-quant all-naturals-quant (x)
  (forall a (implies (member a x) (numberp a)))
  all-naturals-quant)
```

However, we have already pointed out in the introduction that there are times when the user would prefer to avoid the DEFN-SK mechanism. In this section we present two tools for this purpose. The first, DEFN-SK-CONSTRUCTIVE, is quite general: it behaves just like DEFN-SK, except that it requires the user to provide existing ‘‘witnessing functions’’ (in the spirit of CONSTRAIN, cf. [6]) for each of the newly-introduced functions. The second, BDD-FORALL, is an implementation of bounded universal quantification by way of primitive recursive functions. This latter construct is somewhat similar in intent to the more sophisticated and far-reaching ideas for bounded quantification in [8], but is simpler and is based on the DEFN-SK-CONSTRUCTIVE mechanism mentioned above, which in turn is based on CONSTRAIN [6]. Thus, BDD-FORALL is likely to be easily portable to the Acl2 system [9] that is currently under development, unlike the implementation of bounded quantification in [8], which is based on the EVAL$ construct of Nqthm [2] that is not likely to be ported to Acl2.

3.1 DEFN-SK-CONSTRUCTIVE

Suppose that one wants the elegance and rewrite rules provided by DEFN-SK but wishes to stay within the ‘‘constructive’’ world. For example, consider again the notion that every member of a given list is a natural number. Recall that we gave the following recursive definition of this notion,

```
(defn all-naturals (x)
  (if (listp x)
      (and (numberp (car x))
           (all-naturals (cdr x)))
    t))
```

and also the ‘‘nonconstructive’’ version that uses explicit quantification:

```
(defn-sk all-naturals-quant (x)
  (forall a (implies (member a x) (numberp a))))
```

This event adds the ‘‘Skolem’’ axiom

```
(AND (IMPLIES (IMPLIES (MEMBER (A X) X) (NUMBERP (A X)))
              (ALL-NATURALS-QUANT X))
     (IMPLIES (NOT (IMPLIES (MEMBER A X) (NUMBERP A)))
              (NOT (ALL-NATURALS-QUANT X))))
```

which is then stored as two rewrite rules.

We now present a similar mechanism for introducing a quantified notion, but one which does not rely on the nonconstructivity of DEFN-SK. Here is an auxiliary definition (repeated from the preceding section) followed by an example of the new ‘‘event.’’

```
(defn bad-guy (x)
  (if (listp x)
      (if (not (numberp (car x)))
          (car x)
        (bad-guy (cdr x)))
    0))

(defn-sk-constructive all-naturals-quant (x)
  (forall a (implies (member a x) (numberp a)))
  ((all-naturals-quant all-naturals)
   (a bad-guy)))
```

In this case the relevant event is called ALL-NATURALS-QUANT-INTRO, but the rewrite rules stored are the same ones stored for the DEFN-SK event above (except for their names). The ‘‘witnessing alist’’

```
  ((all-naturals-quant all-naturals)
   (a bad-guy))
```

instructs the system to take the Skolem axiom created by DEFN-SK (see above), then functionally instantiate it with this functional substitution, and finally prove that instantiated term:

```
(AND (IMPLIES (IMPLIES (MEMBER (bad-guy X) X)
                       (NUMBERP (bad-guy X)))
              (all-naturals X))
     (IMPLIES (NOT (IMPLIES (MEMBER A X) (NUMBERP A)))
              (NOT (all-naturals X))))
```

Fortunately, there is assistance in seeing the form required for the ‘‘witnessing alist.’’ One simply leaves off that argument and calls DEFN-SK-CONSTRUCTIVE-HELP instead of DEFN-SK-CONSTRUCTIVE.

```
>(defn-sk-constructive-help all-naturals-quant (x)
  (forall a (implies (member a x) (numberp a))))


The Skolemized formula will be:

      (AND (IMPLIES (IMPLIES (MEMBER (A X) X)
                             (NUMBERP (A X)))
                    (ALL-NATURALS-QUANT X))
           (IMPLIES (NOT (IMPLIES (MEMBER A X)
                                  (NUMBERP A)))
                    (NOT (ALL-NATURALS-QUANT X))))).

The witnessing function alist should have roughly the form:
((A ???) (ALL-NATURALS-QUANT ???)).

T

>
```

Actually, a call to DEFN-SK-CONSTRUCTIVE simply generates a call to CONSTRAIN for the Skolem axiom that DEFN-SK would have generated, as illustrated by the response below that Lisp gives to a request to expand the DEFN-SK-CONSTRUCTIVE macro.

```
>(macroexpand-1
  '(defn-sk-constructive all-naturals-quant (x)
     (forall a (implies (member a x) (numberp a)))
     ((all-naturals-quant all-naturals)
      (a bad-guy))))
(CONSTRAIN ALL-NATURALS-QUANT-INTRO (REWRITE)
    (AND (IMPLIES (IMPLIES (MEMBER (A X) X)
                           (NUMBERP (A X)))
                  (ALL-NATURALS-QUANT X))
         (IMPLIES (NOT (IMPLIES (MEMBER A X) (NUMBERP A)))
                  (NOT (ALL-NATURALS-QUANT X))))
    ((ALL-NATURALS-QUANT ALL-NATURALS) (A BAD-GUY)))
T

>
```

The general form for DEFN-SK-CONSTRUCTIVE is as follows:

(DEFN-SK-CONSTRUCTIVE *function-name formal-parameters*
  quantified-formula
  (... (new_i old_i **) ...)**
  &OPTIONAL *hints***)**

where *hints* are as in hints to PROVE-LEMMA and (... (new_i old_i) ...) is as in the corresponding argument to CONSTRAIN.

Finally, we observe that this call to DEFN-SK-CONSTRUCTIVE does indeed give us a history in which **(all-naturals-quant x)** represents the indicated quantified formula, in the sense described in the preceding section.

```
>(check-quant all-naturals-quant (x)
  (forall a (implies (member a x) (numberp a)))
  all-naturals-quant-intro)
T

>
```

Although we believe that using DEFN-SK-CONSTRUCTIVE provides functions that represent quantified formulas, we encourage the user to use CHECK-QUANT as indicated above in order to gain such assurance.

3.2 BDD-FORALL

The final utility we provide is one that generates appropriate events for defining notions with a single outermost universal quantifier. Of course it is then possible to allow arbitrary bounded quantifiers simply by introducing such notions in an inside-out manner.

An example should suffice to illustrate the technique. Suppose we have already declared or defined a 4-place predicate **p4**, e.g.

```
(dcl p4 (a b c d))
```

and wish to represent the following quantified notion **forall-p4** (with free variables **i**, **j**, and **y**):

```
For all natural numbers x such that i ≤ x < j,
  (p4 x i j y) holds
```

The following form is used to introduce a function **forall-p4**, with formal parameters **i**, **j**, and **y**, that defines this notion:

```
(bdd-forall forall-p4 x i j (i j y)
            (p4 x i j y))
```

Note however that any term can serve the function of **(p4 x i j y)**; that term need not be an application of a function symbol to variables.

In Appendix A we show what events are actually generated by this form. The only event generated by a call of BDD-FORALL that really matters for subsequent events, however, is the final one. The final one is actually a DEFN-SK-CONSTRUCTIVE (see the preceding subsection), and the other events are present simply to assist in the acceptance of this final event. Here is what that final event looks like for the example above, though since the functional substitution and hints are generated by the system, there is really no need for the user to look at these.

```
(DEFN-SK-CONSTRUCTIVE FORALL-P4 (I J Y)
  (FORALL X
          (IMPLIES (AND (NUMBERP X)
                        (NOT (LESSP X I))
                        (LESSP X J))
                   (P4 X I J Y)))
  ((X FORALL-P4-SK-MODEL)
   (FORALL-P4 FORALL-P4-MODEL))
  ((USE (FORALL-P4-MODEL-NECC)
        (FORALL-P4-MODEL-SUFF))
   (DISABLE-THEORY T)
   (ENABLE-THEORY GROUND-ZERO)))
```

Following our own admonition from the preceding subsection, let us check that we have correctly represented the quantified notion in question.

```
>(check-quant forall-p4 (i j y)
                (forall x (implies (and (numberp x)
                                        (not (lessp x i))
                                        (lessp x j))
                                   (p4 x i j y)))
                forall-p4-intro)
T

>
```

It should be straightforward to define the analog of BDD-FORALL for the existential quantifier, as well as versions for primitive recursion over finite lists instead of over numbers. We simply have not yet done so.

We may also wish to reimplement BDD-FORALL so that it is based entirely on DEFN events rather than on a CONSTRAIN event, so that the resulting function will be executable.

Appendix A
Transcript from an execution of a BDD-FORALL form

This appendix serves not only to illustrate BDD-FORALL, but in fact it illustrates all of the notions introduced in this paper. The run below took place on a Sun 3/60 with 24 megabytes of main memory.

```
client12:kaufmann[82]% pc-nqthm
AKCL (Austin Kyoto Common Lisp)  Version(1.602) Sun Sep 29 19:41:25 CDT 1991
Contains Enhancements by W. Schelter

Nqthm, with functional instantiation.
Initialized with (BOOT-STRAP NQTHM) on September 29, 1991  22:26:0.
>(load "defn-sk-constructive.o")
Loading defn-sk-constructive.o
start address -T 594000 Finished loading defn-sk-constructive.o
14016

>(dcl p4 (x i j y))

[ 0.0 0.0 0.0 ]
P4

>(bdd-forall forall-p4 x i j (i j y)
             ;; for all x with i <= x < j, (p4 x i j y) holds
             (p4 x i j y))

^L

(DEFN FORALL-P4-SK-MODEL-AUX
      (X I J Y)
      (IF (LESSP X J)
          (IF (NOT (P4 X I J Y))
              X
              (FORALL-P4-SK-MODEL-AUX (ADD1 X)
                                      I J Y))
          J)
      ((LESSP (DIFFERENCE (ADD1 J) X))))

     Linear arithmetic establishes that the measure (DIFFERENCE (ADD1 J) X)
decreases according to the well-founded relation LESSP in each recursive call.
Hence, FORALL-P4-SK-MODEL-AUX is accepted under the principle of definition.
Observe that:
      (OR (NUMBERP (FORALL-P4-SK-MODEL-AUX X I J Y))
          (EQUAL (FORALL-P4-SK-MODEL-AUX X I J Y)
                 X)
          (EQUAL (FORALL-P4-SK-MODEL-AUX X I J Y)
                 J))
is a theorem.



[ 0.1 0.0 0.1 ]

FORALL-P4-SK-MODEL-AUX
```

```
^L

(DEFN FORALL-P4-SK-MODEL
      (I J Y)
      (FORALL-P4-SK-MODEL-AUX (FIX I)
                              I J Y))

     From the definition we can conclude that:
      (OR (NUMBERP (FORALL-P4-SK-MODEL I J Y))
          (EQUAL (FORALL-P4-SK-MODEL I J Y) J))
is a theorem.



[ 0.1 0.0 0.0 ]

FORALL-P4-SK-MODEL
^L

(DEFN FORALL-P4-MODEL-AUX
      (X I J Y)
      (IF (LESSP X J)
          (IF (P4 X I J Y)
              (FORALL-P4-MODEL-AUX (ADD1 X) I J Y)
              F)
          T)
      ((LESSP (DIFFERENCE (ADD1 J) X))))

     Linear arithmetic informs us that the measure (DIFFERENCE (ADD1 J) X)
decreases according to the well-founded relation LESSP in each recursive call.
Hence, FORALL-P4-MODEL-AUX is accepted under the principle of definition.
Note that:
      (OR (FALSEP (FORALL-P4-MODEL-AUX X I J Y))
          (TRUEP (FORALL-P4-MODEL-AUX X I J Y)))
is a theorem.



[ 0.1 0.0 0.1 ]

FORALL-P4-MODEL-AUX
^L

(DEFN FORALL-P4-MODEL
      (I J Y)
      (FORALL-P4-MODEL-AUX (FIX I) I J Y))

     From the definition we can conclude that:
      (OR (FALSEP (FORALL-P4-MODEL I J Y))
          (TRUEP (FORALL-P4-MODEL I J Y)))
is a theorem.



[ 0.1 0.0 0.0 ]

FORALL-P4-MODEL
```

```
^L

(PROVE-LEMMA FORALL-P4-MODEL-AUX-NECC NIL
             (IMPLIES (NOT (IMPLIES (AND (NUMBERP X)
                                         (NUMBERP X**)
                                         (NOT (LESSP X X**))
                                         (LESSP X J))
                                    (P4 X I J Y)))
                      (NOT (FORALL-P4-MODEL-AUX X** I J Y))))


     Name the conjecture *1.

     We will appeal to induction.  There is only one plausible induction.  We
will induct according to the following scheme:
      (AND (IMPLIES (AND (LESSP X** J)
                         (P4 X** I J Y)
                         (p (ADD1 X**) I J Y X))
                    (p X** I J Y X))
           (IMPLIES (AND (LESSP X** J)
                         (NOT (P4 X** I J Y)))
                    (p X** I J Y X))
           (IMPLIES (NOT (LESSP X** J))
                    (p X** I J Y X))).
Linear arithmetic can be used to show that the measure:
      (DIFFERENCE (ADD1 J) X**)
decreases according to the well-founded relation LESSP in each induction step
of the scheme.  The above induction scheme generates four new conjectures:
```

<<<... rest of proof omitted; simplification is all that's needed >>>

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


[ 0.1 1.0 0.7 ]

FORALL-P4-MODEL-AUX-NECC
^L

(PROVE-LEMMA FORALL-P4-MODEL-NECC NIL
             (IMPLIES (NOT (IMPLIES (AND (NUMBERP X)
                                         (NOT (LESSP X I))
                                         (LESSP X J))
                                    (P4 X I J Y)))
                      (NOT (FORALL-P4-MODEL I J Y)))
             ((USE (FORALL-P4-MODEL-AUX-NECC (X** (FIX I))))))

This simplifies, opening up the functions FIX, NOT, AND, IMPLIES, LESSP, and
FORALL-P4-MODEL, to:

      (IMPLIES (AND (NOT (NUMBERP I))
                    (LESSP X 0)
                    (NUMBERP X)
                    (LESSP X J)
                    (NOT (P4 X I J Y)))
               (NOT (FORALL-P4-MODEL-AUX 0 I J Y))).

But this again simplifies, using linear arithmetic, to:

      T.

Q.E.D.


[ 0.1 0.8 0.1 ]

FORALL-P4-MODEL-NECC
```

```
^L

(PROVE-LEMMA FORALL-P4-MODEL-AUX-SUFF NIL
             (IMPLIES (AND (NUMBERP X)
                           (NOT (LESSP X I))
                           (LET ((X (FORALL-P4-SK-MODEL-AUX X I J Y)))
                                (IMPLIES (AND (NUMBERP X)
                                              (NOT (LESSP X I))
                                              (LESSP X J))
                                         (P4 X I J Y))))
                      (FORALL-P4-MODEL-AUX X I J Y))
             ((INDUCT (FORALL-P4-MODEL-AUX X I J Y))))

This formula can be simplified, using the abbreviations IMPLIES, NOT, OR, and
AND, to the following three new goals:
```

<<<... rest of proof omitted; simplification is all that's needed >>>

```
Q.E.D.


[ 0.1 7.2 0.9 ]

FORALL-P4-MODEL-AUX-SUFF
^L

(PROVE-LEMMA FORALL-P4-MODEL-SUFF NIL
             (LET ((X (FORALL-P4-SK-MODEL I J Y)))
                  (IMPLIES (IMPLIES (AND (NUMBERP X)
                                         (NOT (LESSP X I))
                                         (LESSP X J))
                                    (P4 X I J Y))
                           (FORALL-P4-MODEL I J Y)))
             ((USE (FORALL-P4-MODEL-AUX-SUFF (X (FIX I))))))

This simplifies, unfolding the definitions of FIX, NOT, AND, IMPLIES,
FORALL-P4-SK-MODEL, LESSP, and FORALL-P4-MODEL, to seven new conjectures:
```

<<<... rest of proof omitted; simplification is all that's needed >>>

```
Q.E.D.


[ 0.1 4.4 0.5 ]

FORALL-P4-MODEL-SUFF
^L

(DEFN-SK-CONSTRUCTIVE FORALL-P4
                      (I J Y)
                      (FORALL X
                              (IMPLIES (AND (NUMBERP X)
                                            (NOT (LESSP X I))
                                            (LESSP X J))
                                       (P4 X I J Y)))
                      ((X FORALL-P4-SK-MODEL)
                       (FORALL-P4 FORALL-P4-MODEL))
                      ((USE (FORALL-P4-MODEL-NECC)
                            (FORALL-P4-MODEL-SUFF))
                       (DISABLE-THEORY T)
                       (ENABLE-THEORY GROUND-ZERO)))

WARNING:  Note that FORALL-P4-INTRO contains the free variable X which will be
chosen by instantiating the hypothesis:
      (NOT (IMPLIES (AND (NUMBERP X)
                         (NOT (LESSP X I))
                         (LESSP X J))
                    (P4 X I J Y))).
```

```
We will verify the consistency and the conservative nature of this constraint
by attempting to prove:
      (AND (IMPLIES (IMPLIES (AND (NUMBERP (FORALL-P4-SK-MODEL I J Y))
                                  (NOT (LESSP (FORALL-P4-SK-MODEL I J Y) I))
                                  (LESSP (FORALL-P4-SK-MODEL I J Y) J))
                             (P4 (FORALL-P4-SK-MODEL I J Y) I J Y))
                    (FORALL-P4-MODEL I J Y))
           (IMPLIES (NOT (IMPLIES (AND (NUMBERP X)
                                       (NOT (LESSP X I))
                                       (LESSP X J))
                                  (P4 X I J Y)))
                    (NOT (FORALL-P4-MODEL I J Y)))).


This formula simplifies, unfolding the definitions of NOT, AND, IMPLIES, and
LESSP, to:

      T.

Q.E.D.


[ 0.5 3.7 0.0 ]

FORALL-P4-INTRO
T

>(check-quant forall-p4 (i j y)
              (forall x (implies (and (numberp x)
                                      (not (lessp x i))
                                      (lessp x j))
                                 (p4 x i j y)))
              forall-p4-intro)
T

>
```

Appendix B
The Common Lisp code

```
(defun defn-sk-constructive-fn (name args body witness-alist hints defn-sk-hints)
  (let (skolemized-body new-functions-and-formals)
       (match! (chk-acceptable-defn-sk name args body defn-sk-hints)
               (list name args body defn-sk-hints
                     skolemized-body new-functions-and-formals))
       (nconc (list 'constrain (pack (list name '-intro)) '(rewrite) skolemized-body
                    witness-alist)
              (if hints (list hints) nil))))

(defmacro defn-sk-constructive
  (name args body witness-alist &optional hints defn-sk-hints)
  (defn-sk-constructive-fn name args body witness-alist hints defn-sk-hints))

(defmacro defn-sk-constructive-help (name args body)
  (let (skolemized-body new-functions-and-formals)
    (match! (chk-acceptable-defn-sk name args body hints)
            (list name args body hints
                  skolemized-body new-functions-and-formals))
    (PRINEVAL (PQUOTE (PROGN CR CR
                             |The| |Skolemized| |formula| |will| |be| |:|))
              nil
              0 PROVE-FILE)
    (PRINEVAL (PQUOTE (PROGN CR (!term skolemized-body (quote |.|)) CR
                             |The| |witnessing| |function| |alist|
                             |should| |have| |roughly| |the| |form| |:|))
              `((skolemized-body . ,skolemized-body))
              0 PROVE-FILE)
    (PRINEVAL (PQUOTE (PROGN CR (!ppr x (quote |.|)) CR CR))
              `((X . ,(iterate for fn-formals in new-functions-and-formals
                               collect (list (car fn-formals)
                                             '???))))
              0 PROVE-FILE)
    t))

(defun defn-sk-nonconstructive-fn (name args body defn-sk-hints)
  (let (skolemized-body new-functions-and-formals)
    (match! (chk-acceptable-defn-sk name args body defn-sk-hints)
            (list name args body defn-sk-hints
                  skolemized-body new-functions-and-formals))
    `(let (undone-events)
       (do-events
        ',(iterate for fn-args in new-functions-and-formals
                   with temp
                   collect
                   `(dcl ,(car fn-args) ,(cdr fn-args))
                   into temp
                   finally
                   (return
                    (append temp
                            `((add-axiom ,(pack (list name '-intro))
                                         (rewrite)
                                         ,(untranslate skolemized-body))
                              (add-axiom ,(pack (list name '-boolean))
                                         (rewrite)
                                         (or (truep (,name ,@args))
                                             (falsep (,name ,@args)))))))))))))

(defmacro defn-sk-nonconstructive (name args body &optional defn-sk-hints)
```

```
  (defn-sk-nonconstructive-fn name args body defn-sk-hints))

(defun functionally-one-way-unify-rec (pat term fns acc &aux pair)
  ;; Returns a functional substitution fs with domain contained in
  ;; fns, no member of which is a constructor, such that
  ;; pat \ (fs U acc) = term; or else returns 'fail.  Here we're doing very
  ;; simple one-way unification -- no fancy higher-order stuff,
  ;; and every function symbol that's bound is bound to a symbol.
  ;; Let's say that the "free" function symbols are those in fns
  ;; which aren't bound in acc.
  ;; We assume that no function symbol of fns occurs in term.
  (cond
   ((variablep pat)
    (if (eq pat term)
        acc
      'fail))
   ((fquotep pat)
    (if (equal pat term)
        acc
      'fail))
   ((variablep term)
    'fail)
   ((eq (ffn-symb pat) (fn-symb term))
    ;; then by assumption, (ffn-symb pat) is not in fns, and hence
    ;; we'll never replace it
    (functionally-one-way-unify-lst (fargs pat) (sargs term) fns acc))
   ((setq pair (assoc-eq (ffn-symb pat) acc))
    (if (eq (cdr pair) (fn-symb term))
        (functionally-one-way-unify-lst (fargs pat) (sargs term) fns acc)
      'fail))
   ((not (member-eq (ffn-symb pat) fns))
    'fail)
   ((= (arity (ffn-symb pat)) (arity (fn-symb term)))
    ;; otherwise we are allowed to bind the function symbol of pat
    (functionally-one-way-unify-lst
     (fargs pat)
     (sargs term)
     fns
     (cons (cons (ffn-symb pat) (fn-symb term)) acc)))
   (t
    'fail)))

(defun functionally-one-way-unify-lst (patlist termlist fns acc)
  (if patlist
      (let ((new-acc (functionally-one-way-unify-rec
                      (car patlist) (car termlist) fns acc)))
        (if (eq new-acc 'fail)
            'fail
          (functionally-one-way-unify-lst
           (cdr patlist) (cdr termlist) fns new-acc)))
    acc))

(defun functionally-one-way-unify (patlist termlist fns)
  (functionally-one-way-unify-rec patlist termlist fns nil))

(defun fnnamep-formula (fn x &aux (args (formula-args x)) (op (formula-op x)))
  (cond
   ((eq op 'term)
    (fnnamep fn args))
   ((member-eq op '(forall exists))
    (fnnamep-formula fn (cadr args)))
   (t (iterate for arg in args
               thereis (fnnamep-formula fn arg)))))

(defun is-skolemization (term formula &optional translate-flg)
  ;; adapted from CHK-ACCEPTABLE-DEFN-SK
  (let ((translated-formula
         (if translate-flg
             (translate-to-formula formula)
```

```
          formula))
       (untranslated-formula
        (if translate-flg
            formula
          (untranslate-formula formula))))
    (when translate-flg (setq term (translate term)))
    (let ((x (skolemize translated-formula t))
          (arity-alist arity-alist))
      (iterate for pair in *new-functions-and-formals*
               do
               (setq arity-alist
                     (cons (cons (car pair) (length (cdr pair)))
                           arity-alist)))
      (setq unify-subst (functionally-one-way-unify
                         x
                         term
                         (mapcar 'car *new-functions-and-formals*)))
      (if (eq unify-subst 'fail)
          (progn (PRINEVAL (PQUOTE (PROGN CR |The| |Skolemization| |of| CR
                                          (!ppr untranslated-formula nil)
                                          |is| CR (!term x nil) |,| |but| |the| |term|
                                          (!term term nil) |is| |not| |a|
                                          |functional| |instance|
                                          |of| |that| |term| |.| CR CR))
                           `((untranslated-formula . ,untranslated-formula)
                             (x . ,x)
                             (term . ,term))
                           0 PROVE-FILE)
                 nil)
        t))))

(defun check-quant-with-fs (name witness-alist)
  ;; This is adapted from Nqthm code for CHK-ACCEPTABLE-CONSTRAIN.
  ;; Remember, the basic idea is that to show that a quantified
  ;; notion is correctly represented, we need to show that a
  ;; functional instantiation of a Skolemization is a theorem.
  ;; I'm probably could allow lambda expressions, but I won't
  ;; at this time.

;   It is important that the function symbols being witnessed, i.e.,
;   the new function symbols, not occur in the terms in the range of
;   the substitution.  This is insured by our not binding ARITY-ALIST
;   until after the TRANSLATE on the range terms.

  (MATCH! (CHK-ACCEPTABLE-FUNCTIONAL-SUBSTITUTION WITNESS-ALIST T)
          (LIST WITNESS-ALIST))
  (OR (NO-DUPLICATESP (CONS NAME (ITERATE FOR DOUBLET IN WITNESS-ALIST
                                          COLLECT (CAR DOUBLET))))
      (ER SOFT NIL
          |it| |is| |illegal| |to| |use| |the| |name| |of| |a|
          |Skolem| |function|
          |for| |the| |name| |of| |the| |function| |that| |is|
          |representing| |the| |quantified| |notion| |,|
          |or| |to| |duplicate| |a| |function| |in| |the| |domain| |of| |the|
          |witnessing| |alist| |of| |the| |constraint| |.|))
  (or (iterate for doublet in witness-alist always (symbolp (cadr doublet)))
      (er soft nil |We| |do| |not| |currently| |allow| |lambda| |expressions|
          |in| |the| |witnessing| |alist| |for| |CHECK-QUANT| |.|))
  WITNESS-ALIST)

(defun check-quant-fn (NAME ARGS BODY event-name-or-witness-alist hints &aux formula)
  ;; Checks that an existing notion NAME defines the notion in BODY
  ;; in the sense that (IFF (NAME . ARGS) BODY).
  (or (get name 'type-prescription-lst)
      (er soft (name) (!ppr name nil) |is| |not| |a| |function|
          |known| |in| |the| |current| |history| |.|))
  (chk-arglist name args)
  (setq formula (translate-to-formula body))
  (if (fnnamep-formula name formula)
```

```
    (progn (PRINEVAL (PQUOTE (PROGN CR |The| |function| |symbol| (!ppr name nil)
                                    |occurs| |in| |the| |formula| (!ppr body nil) |,|
                                    |and| |hence| (!ppr x nil) |is| |not| |what|
                                    |we| |call| |a|
                                    |representation| |of| |the| |indicated|
                                    |quantified| |notion| |.| CR CR))
                     `((name . ,name)
                       (body . ,body)
                       (x . (,name ,@args)))
                     0 PROVE-FILE)
           (return-from check-quant-fn nil)))
;; could probably omit the following
(free-var-chk-formula name args formula)
(cond
 ((symbolp event-name-or-witness-alist)
  (if (formula-of event-name-or-witness-alist)
      (is-skolemization (formula-of event-name-or-witness-alist)
                        (make-defn-sk-formula name args formula))
    (progn (PRINEVAL (PQUOTE (PROGN CR |The| |symbol|
                                    (!ppr event-name-or-witness-alist nil)
                                    |is| |not| |the| |name| |of| |an| |event|
                                    |with| |a| |formula| |.| CR CR))
                     `((event-name-or-witness-alist . ,event-name-or-witness-alist))
                     0 PROVE-FILE)
           nil)))
 (t
  ;; Otherwise, we check that the result of functionally substituting
  ;; the witness-alist into the Skolemization is really a theorem, even
  ;; though it would in fact be legal to further substitute and get a
  ;; theorem then instead.
  (let ((term (skolemize (make-defn-sk-formula name args formula) t))
        (arity-alist arity-alist))
    (iterate for pair in *new-functions-and-formals*
             do
             (setq arity-alist
                   (cons (cons (car pair) (length (cdr pair)))
                         arity-alist)))
    (iterate for pair in event-name-or-witness-alist
             do (or (assoc-eq (car pair) *new-functions-and-formals*)
                    (er soft ((fn (car pair))) |The| |function| |symbol|
                        (!ppr fn nil) |appears| |in| |the| |domain| |of|
                        |the| |witnessing| |alist| |but| |is| |not| |a|
                        |Skolem| |function| |generated| |by| |the|
                        |Skolemizer| |in| |the| |current| |context| |.|)))
    (iterate for pair in *new-functions-and-formals*
             do (or (assoc-eq (car pair) event-name-or-witness-alist)
                    (er soft ((fn (car pair)) term) |The| |function| |symbol|
                        (!ppr fn nil) |was| |generated| |as| |a|
                        |Skolem| |function| |but| |does| |not| |appear|
                        |in| |the| |witnessing| |alist| |that|
                        |was| |supplied| |.| |The| |Skolemization|
                        |is| |:| CR (!term term '|.|))))
    (let ((term-to-prove
           (sublis-fn (iterate for doublet
                               in (check-quant-with-fs
                                   name event-name-or-witness-alist)
                               collect
                               (cons (car doublet)
                                     (cadr doublet)))
                      term)))
      (PRINEVAL (PQUOTE (PROGN CR |Our| |proof| |obligation| |is| |:|
                               (!ppr term-to-prove '|.|) CR CR))
                `((term-to-prove . ,term-to-prove))
                0 PROVE-FILE)
      ;; The following is adapted from the code for PROVE-LEMMA
      (LET ((IN-PROVE-LEMMA-FLG T) PROVE-ANS)
           (MATCH! (CHK-ACCEPTABLE-HINTS HINTS)
                   (LIST HINTS))
           (UNWIND-PROTECT
```

```
              (progn
                ;; Call translate first so that if arities is wrong, we
                ;; find out now.  Kind of a shame, since PROVE calls translate
                ;; too, but that’s OK.
                (or (error1-set (setq term-to-prove
                                      (translate (APPLY-HINTS HINTS TERM-to-prove))))
                    (er soft nil |Aborting| |,| |since| |our| |proof| |obligation|
                        |is| |not| |well-formed| |.|))
                (and (PROVE term-to-prove)
                     t))
              (PROGN (ITERATE FOR X IN HINT-VARIABLE-ALIST
                              DO (SET (CADR X) (CADDDR X)))
                     (SETQ LOCAL-DISABLEDP-HASH-FLAG NIL))))))))

(defmacro check-quant (&optional NAME ARGS BODY event-name hints)
  (if (or (null name)
          (null body)
          (null event-name))
      (er soft nil CHECK-QUANT |has| |four| |required| |arguments| |.|)
    (if (and event-name (symbolp event-name) hints)
        (er soft (event-name hints) |Since| |a| |previous| |event|
            |name| |was| |specified| |,| |namely| (!ppr event-name nil)
            |,| |it| |is| |not| |sensible| |or| |legal| |to| |supply|
            |hints| |here| |.| |The| |final| |argument| (!ppr hints nil)
            |to| CHECK-QUANT |is| |thus| |illegal| |.|)
      ‘(check-quant-fn ’,name ’,args ’,body ’,event-name ’,hints))))

(defun check-quant-help-fn (NAME ARGS BODY &aux formula)
  ;; Checks that an existing notion NAME defines the notion in BODY
  ;; in the sense that (IFF (NAME . ARGS) BODY).
  (or (arity name)
      (er soft (name) (!ppr name nil) |is| |not| |a| |function|
          |known| |in| |the| |current| |history| |.|))
  (chk-arglist name args)
  (setq formula (translate-to-formula body))
  ;; could probably omit the following
  (free-var-chk-formula name args formula)
  (let ((x (skolemize (make-defn-sk-formula name args formula) t)))
    (PRINEVAL (PQUOTE (PROGN CR |The| |new| |function|
                             (plural?
                              y
                              (progn |symbols| |with| |their| |argument| |lists| |are|)
                              (progn |symbol| |with| |its| |argument| |list| |is|))
                             (!ppr-list y (quote |.|)) CR CR))
              ‘((y . ,*new-functions-and-formals*))
              0 PROVE-FILE)
    (untranslate x)))

(defmacro check-quant-help (name args body)
  ‘(check-quant-help-fn ’,name ’,args ’,body))

(defun bdd-forall-sk-model-aux (bdd-forall-sk-model-aux-name qvar upper args body)
  ‘(defn ,bdd-forall-sk-model-aux-name ,(cons qvar args)
     (if (lessp ,qvar ,upper)
         (if (not ,body)
             ,qvar
           (,bdd-forall-sk-model-aux-name (add1 ,qvar) ,@args))
       ,upper)
     ((lessp (difference (add1 ,upper) ,qvar)))))

(defun bdd-forall-sk-model
  (bdd-forall-sk-model-aux-name bdd-forall-sk-model-name lower args)
  ‘(defn ,bdd-forall-sk-model-name ,args
     (,bdd-forall-sk-model-aux-name (fix ,lower) ,@args)))

(defun bdd-forall-model-aux (bdd-forall-model-aux-name qvar upper args body)
  ‘(defn ,bdd-forall-model-aux-name ,(cons qvar args)
     (if (lessp ,qvar ,upper)
         (if ,body
```

```
            (,bdd-forall-model-aux-name (add1 ,qvar) ,@args)
          f)
      t)
    ((lessp (difference (add1 ,upper) ,qvar)))))

(defun bdd-forall-model
  (bdd-forall-model-aux-name bdd-forall-model-name lower args)
  '(defn ,bdd-forall-model-name ,args
     (,bdd-forall-model-aux-name (fix ,lower) ,@args)))

(defun bdd-forall-model-aux-necc (bdd-forall-model-aux-necc-name
                                  bdd-forall-model-aux-name
                                  qvar qvar** upper args body)
  '(prove-lemma
    ,bdd-forall-model-aux-necc-name
    ()
    (implies (not (implies (and (numberp ,qvar)
                                (numberp ,qvar**)
                                (not (lessp ,qvar ,qvar**))
                                (lessp ,qvar ,upper))
                           ,body))
             (not (,bdd-forall-model-aux-name ,qvar** ,@args)))))

(defun bdd-forall-model-necc (bdd-forall-model-necc-name
                              bdd-forall-model-name
                              bdd-forall-model-aux-necc-name
                              qvar qvar** lower upper args body)
  '(prove-lemma
    ,bdd-forall-model-necc-name
    ()
    (implies (not (implies (and (numberp ,qvar)
                                (not (lessp ,qvar ,lower))
                                (lessp ,qvar ,upper))
                           ,body))
             (not (,bdd-forall-model-name ,@args)))
    ((use (,bdd-forall-model-aux-necc-name (,qvar** (fix ,lower)))))))

(defun bdd-forall-model-aux-suff (bdd-forall-model-aux-suff-name
                                  bdd-forall-sk-model-aux-name
                                  bdd-forall-model-aux-name
                                  qvar lower upper args body)
  '(prove-lemma
    ,bdd-forall-model-aux-suff-name
    ()
    (implies (and (numberp ,qvar)
                  (not (lessp ,qvar i))
                  (let ((,qvar (,bdd-forall-sk-model-aux-name ,qvar ,@args)))
                    (implies (and (numberp ,qvar)
                                  (not (lessp ,qvar ,lower))
                                  (lessp ,qvar ,upper))
                             ,body)))
             (,bdd-forall-model-aux-name ,qvar ,@args))
    ((induct (,bdd-forall-model-aux-name ,qvar ,@args)))))

(defun bdd-forall-model-suff (bdd-forall-model-aux-suff-name
                              bdd-forall-model-suff-name
                              bdd-forall-sk-model-name
                              bdd-forall-model-name
                              qvar lower upper args body)
  '(prove-lemma
    ,bdd-forall-model-suff-name
    ()
    (let ((,qvar (,bdd-forall-sk-model-name ,@args)))
      (implies (implies (and (numberp ,qvar)
                             (not (lessp ,qvar ,lower))
                             (lessp ,qvar ,upper))
                        ,body)
               (,bdd-forall-model-name ,@args)))
    ((use (,bdd-forall-model-aux-suff-name (,qvar (fix ,lower)))))))
```

```
(defun bdd-forall-defn-sk-constructive
  (name bdd-forall-sk-model-name bdd-forall-model-necc-name
        bdd-forall-model-suff-name bdd-forall-model-name
        qvar lower upper args body)
  '(defn-sk-constructive ,name ,args
     (forall ,qvar (implies (and (numberp ,qvar)
                                 (not (lessp ,qvar ,lower))
                                 (lessp ,qvar ,upper))
                            ,body))
     ((,(make-new-skolem-fn-1 qvar 0 (list (cons name args)))
       ,bdd-forall-sk-model-name)
      (,name ,bdd-forall-model-name))
     ((use (,bdd-forall-model-necc-name)
           (,bdd-forall-model-suff-name))
      (disable-theory t)
      (enable-theory ground-zero))))

(defun bdd-forall-events (name qvar lower upper args body)
  ;; defines (name .args) to be
  ;; (forall qvar (implies (and (numberp qvar)
  ;;                            (not (lessp qvar lower))
  ;;                            (lessp qvar upper))
  ;;                       body))
  (chk-new-name name t)
  (chk-arglist name args)
  (if (or (nvariablep qvar)
          (member-eq qvar args))
      (er soft (qvar args) |The| |quantified| |variable| (!ppr qvar nil)
          |is| |supposed| |to| |be| |a| |variable| |that| |is| |not| |a|
          |member| |of| |the| |argument| |list| (!ppr args (quote |.|))))
  ;; let's make sure these all translate before we even start
  (translate lower)
  (translate upper)
  (translate body)
  (let ((bdd-forall-sk-model-aux-name (pack (list name '-sk-model-aux)))
        (bdd-forall-sk-model-name (pack (list name '-sk-model)))
        (bdd-forall-model-aux-name (pack (list name '-model-aux)))
        (bdd-forall-model-name (pack (list name '-model)))
        (bdd-forall-model-aux-necc-name (pack (list name '-model-aux-necc)))
        (qvar** (pack (list qvar '**)))
        (bdd-forall-model-necc-name (pack (list name '-model-necc)))
        (bdd-forall-model-aux-suff-name (pack (list name '-model-aux-suff)))
        (bdd-forall-model-suff-name (pack (list name '-model-suff))))
    (list (bdd-forall-sk-model-aux
           bdd-forall-sk-model-aux-name qvar upper args body)
          (bdd-forall-sk-model
           bdd-forall-sk-model-aux-name bdd-forall-sk-model-name lower args)
          (bdd-forall-model-aux bdd-forall-model-aux-name qvar upper args body)
          (bdd-forall-model bdd-forall-model-aux-name bdd-forall-model-name lower args)
          (bdd-forall-model-aux-necc
           bdd-forall-model-aux-necc-name bdd-forall-model-aux-name
           qvar qvar** upper args body)
          (bdd-forall-model-necc
           bdd-forall-model-necc-name bdd-forall-model-name
           bdd-forall-model-aux-necc-name
           qvar qvar** lower upper args body)
          (bdd-forall-model-aux-suff
           bdd-forall-model-aux-suff-name
           bdd-forall-sk-model-aux-name
           bdd-forall-model-aux-name
           qvar lower upper args body)
          (bdd-forall-model-suff
           bdd-forall-model-aux-suff-name
           bdd-forall-model-suff-name
           bdd-forall-sk-model-name
           bdd-forall-model-name
           qvar lower upper args body)
          (bdd-forall-defn-sk-constructive
           name bdd-forall-sk-model-name bdd-forall-model-necc-name
```

```
          bdd-forall-model-suff-name bdd-forall-model-name
          qvar lower upper args body))))

(defmacro bdd-forall (name qvar lower upper args body)
  `(do-events (bdd-forall-events
              ',name ',qvar ',lower ',upper ',args
              ',body)))
```

References

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

2. R. S. Boyer and J S. Moore, *A Computational Logic Handbook*, Academic Press, Boston, 1988.

3. Matt Kaufmann, ''A User's Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover'', Technical Report 19, Computational Logic, Inc., May 1988.

4. M. Kaufmann, ''Addition of Free Variables to an Interactive Enchancement of the Boyer-Moore Theorem Prover'', Tech. report 42, Computational Logic, Inc., 1717 West Sixth Street, Suite 290 Austin, TX 78703, 1990.

5. Matt Kaufmann, ''An Extension of the Boyer-Moore Theorem Prover to Support First-Order Quantification'', *Journal of Automated Reasoning*, (to appear), See also CLI Technical Report 43, May, 1989 for an expanded version

6. R.S. Boyer, D. Goldschlag, M. Kaufmann, J S. Moore, ''Functional Instantiation in First Order Logic'', *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, Academic Press, 1991, pp. 7-26, Versions also published as CLI Technical Report 44 and in proceedings of the 1989 Workshop on Programming Logic, Programming Methodology Group, University of Goteborg, West Germany.

7. Matt Kaufmann, ''Skolemization Explained Simply'', Internal Note 27, Computational Logic, Inc., November 1987.

8. R. S. Boyer and J S. Moore, ''The Addition of Bounded Quantification and Partial Functions to A Computational Logic and Its Theorem Prover'', *Journal of Automated Reasoning*, Vol. 4, No. 2, 1988, pp. 117-172.

9. R.S. Boyer, J S. Moore, ''A Theorem Prover for a Computational Logic'', Technical Report 54, Computational Logic, Inc., 1990, Published in proceedings of 10th International Conference on Automated Deduction, Kaiserslautern, West Germany, July 1990

Table of Contents