Addition of Free Variables to the PC-NQTHM Interactive Enhancement of the Boyer-Moore Theorem Prover

Matt Kaufmann

Technical Report #42 May, 1989 (revised March, 1990)

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

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

Acknowledgements

I'd like to thank my colleagues at Computational Logic, Inc. for useful conversations and suggestions during the course of this work. I'd especially like to thank Bob Boyer for a suggestion which led directly to this approach and Matt Wilding, David Goldschlag, and Bishop Brock for helpful comments on drafts of this report.

1. Introduction

The PC-NQTHM system, also referred to below as ‘‘the proof-checker’’, is an interactive enhancement to the Boyer-Moore Theorem Prover (NQTHM). CLI Technical Report 19 [1] is a user’s manual for PC-NQTHM as it existed in May 1988. Since that time we have, however, added a notion of ‘‘free variables’’ to that system. The present report documents changes since May 1988 that involve the addition of free variables to PC-NQTHM. However, other changes from the version reported in [1], besides those relating to free variables, can be found in the appendix at the end of this report. We assume that the reader has some familiarity with PC-NQTHM, either by way of having used it a little or by way of having looked briefly at the user’s manual [1].

The primary motivation behind this extension of PC-NQTHM was a desire to give support to an experimental modification of the Boyer-Moore Theorem Prover that allows first-order quantifiers; see [2]. In fact, some appropriate new macro commands have been created and used successfully (also reported in [2]) in the presence of first-order quantifiers; see 3.2 below. However, this extension of PC-NQTHM documented herein should prove useful even without involving quantifiers, since it allows the user to defer the choice of substitution when using the built-in commands CLAIM, USE-LEMMA, and REWRITE; see Subsection 3.1 below.

With a few minor exceptions[1], the new version is ‘‘upward-compatible’’ with the previous version in the following sense: proofs that replay in the previous system still replay in the new version. However, the new version includes a notion of *free variable* in the proof state. In the previous version, when one wished to use a lemma (either by way of the proof-checker’s REWRITE or USE-LEMMA commands), one had to specify the instance of that lemma at the time it was used (by way of an optional substitution argument). In the new version, new variables are simply labeled as ‘‘free’’, and three new built-in commands have been introduced to handle free variables. Free variables may be instantiated by way of the PUT command. The COPY command conveniently copies hypotheses so that, roughly speaking, the user can perform multiple PUTs. Finally, the help command SHOW-FREE-VARIABLES displays the current free variables. These commands are documented by the on-line help facility and in this report, as are some convenient macro commands.

Section 2 explains our notion of ‘‘free variables’’ and documents the new built-in commands COPY,

[1] GENERALIZE lemmas are now used in the new proof-checker’s GENERALIZE command; a very minor change has been made to SPLIT for better efficiency; SUBV can only be used when at the top of the conclusion. Also see the appendix.

PUT, and SHOW-FREE-VARIABLES. A quick scan of that section should suffice to prepare the user to start using free variables.

Section 3 documents other changes in PC-NQTHM arising from the addition of the notion of free variables. It explains the changes in existing built-in commands and also documents some new and modified macro commands.

Finally, Section 4 deals with soundness. It provides examples demonstrating the necessity for some restrictions on the system's handling of free variables, reviews some important technical notions, and sketches a proof of soundness of the approach. That section is included more for theoreticians than for PC-NQTHM users. It contains some details about the exact interaction of free variables with various commands, notably GENERALIZE, that were not so fully given in previous sections.

Here's a bit of background. A version of the PUT command appeared in an early natural deduction prover of Bledsoe [3]. This prover was incorporated into an early program verification system [4], whose descendent is the current prover for the Gypsy Verification Environment (GVE), [5]. PUT and COPY commands both exist in the current GVE prover, the latter having existed for some 5 years. Our inspiration for the proof-checker's version of these commands came from the GVE prover, although we make no claims about similarity of implementation. In addition, we are unaware of any soundness arguments for similar provers such as those given in Subsection 4.3 of this report.

2. Free variables and the new built-in commands

2.1 Free variables

Our notion of *free variables* is similar to the notion described for the Boyer-Moore Theorem Prover in [6] on pages 235 and 325. Roughly speaking, a *free variable* is one that may be instantiated any way one likes.

Let's formalize this notion as follows. Recall from [1] that a *proof state* has the following fields -- except that we now add a new field, called FREE-VARIABLES:

```
A state consists of:
INSTRUCTION, CURRENT-TERM, GOVERNORS, CURRENT-ADDR-R, GOAL,
   OTHER-GOALS, CUMULATIVE-LEMMAS-USED, FREWRITE-DISABLED-RULES,
   ABBREVIATIONS, and FREE-VARIABLES.
```

Recall also our basic paradigm, namely that each successful invocation of a built-in *change* command pushes a new proof state on top of the so-called **`state-stack`**. The changes to the proof-checker described in this report mostly involve the three new commands mentioned above as well as some new macro commands, together with the effects that the existing commands now have on the FREE-VARIABLES component of a state.

The correctness of the approach is stated formally, and then proved, in Subsection 4.3 of this report. For now, a rough description of the semantics of the approach is that a proof state is *provable* if there is some instantiation for its free variables such that the conjunction of the state's goals, under that instantiation, is a theorem. The commands should thus have the property that if a state is provable after the issuing of a command, then the pre-existing state is provable. Thus, if the final state has the property that every goal is **`T`** (true), then it must follow that the original goal is a theorem.

Let us turn now to the new built-in commands.

2.2 PUT, COPY, and SHOW-FREE-VARIABLES

These new commands are the commands that most directly involve free variables. Here are examples of how they are used:

`(PUT (V$ (PLUS X Y)) (W (TIMES A B)))`
{Substitute **`(PLUS X Y)`** *for* **`V$`** *and* **`(TIMES A B)`** *for* **`W`***} in the current proof state*

`(COPY 3)`
{Copy the third hypothesis, renaming free variables in that hypothesis to obtain the new hypothesis whenever such renaming is known to be sound.}

`SHOW-FREE-VARIABLES`
{Show the free variables together with the names of the goals in which they appear.}

Notice that PUT substitutes for the indicated free variables in the entire proof state, not just in the current goal.

The on-line help facility provides the following descriptions of these commands.

```
->: (help put copy show-free-variables)

     (PUT (V1 term1) ... (Vn termn)):  Apply the indicated substitution to the
entire proof state, where each Vi is a free variable in the current proof
state.
EXAMPLE:  (PUT (V$ (PLUS X Y)))
----------------------------------------------------
     (COPY n1 ... nk):  Copy the indicated active hypotheses, renaming all
free variables in the copies that do not appear in any other goal.
EXAMPLE:  (COPY 3)

     OTHER FORM (see HELP-LONG):  COPY.
----------------------------------------------------
     SHOW-FREE-VARIABLES:  Show the free variables of the current proof state,
each listed together with the goals in which it appears (if any).

     (SHOW-FREE-VARIABLES T):  As above, except that printing of the goal
information is suppressed.
->:
```

Documentation for the command COPY (with no arguments) is found using (HELP-LONG COPY):

```
     COPY:  Same as (COPY n1 ... nk) where (n1 ... nk) is the list of all
active hypotheses, except that a hypothesis is only copied when the resulting
copy contains at least one renamed variable.
```

More details on all these commands may be found by using HELP-LONG.

3. Additional and modified commands

3.1 Modifications to built-in commands

In this section we describe modifications to the built-in ‘‘change’’ commands GENERALIZE, INDUCT, REWRITE, USE-LEMMA, and CLAIM that have been made in order to deal appropriately with free variables. Some of these modifications are relevant to soundness, as shown in the final section. Others are useful for deferring the choice of substitution when using a lemma or a claim.

Let us begin with GENERALIZE. The first example of the final section shows the necessity, for soundness, of having some restriction on how GENERALIZE interacts with the set of free variables of the proof state. Actually it would suffice, when generalizing with a substitution that replaces terms $\mathtt{t_i}$ with corresponding new variables $\mathtt{v_i}$, to remove from the list of free variables any variable that occurs in that substitution. However, we can do a little better. Our precise conditions are rather subtle for when a free variable must become non-free because of GENERALIZE. Therefore, we’ll postpone further discussion of this aspect of GENERALIZE until the soundness proof in Subsection 4.3.

Next let us consider the INDUCT command. Here are two new points from the documentation provided by (HELP-LONG INDUCT).

```
(2) If there are any free variables of the state that occur in the
current goal, these will be removed from the free variables list.  This is
necessary for soundness.  (3) If there are any free variables of the state
occurring in the variables of a command of the form (INDUCT (g v1 ... vn)),
these will be removed from the free variables list.  This too is necessary for
soundness.
```

REWRITE, USE-LEMMA, and CLAIM have been modified so that newly-introduced variables are considered to be free. Here is the appropriate documentation. First, from (HELP-LONG REWRITE) we excerpt the following:

```
     NOTE on free variables of the proof state:  All variables introduced into
the proof state by REWRITE are added to the list of free variables.
```

And, from (HELP-LONG USE-LEMMA) we excerpt the following:

```
     NOTE on free variables of the proof state:  All variables introduced into
the proof state by USE-LEMMA are added to the list of free variables.
```

Finally, here is some new information provided by (HELP-LONG CLAIM), where `exp` is the expression being CLAIMed.

```
     NOTE:  Any variables in  exp  that do not occur anywhere else in the
current proof state will be added to the list of free variables of the proof
state.
```

3.2 Some new and some improved macro commands

Several macro commands have been newly provided or strengthened with the new system, some of which deal particularly with free variables. The most interesting ones are documented in this section. Some new commands that are not related to free variables are described briefly in the appendix. Special attention is given to the command SK* at the end of this subsection.

First we give a very brief summary of the commands that we will document below. ANDSPLIT+ is like ANDSPLIT, except that it copies hypotheses to avoid unwanted sharing of free variables among goals. BACKCHAIN has been enhanced, primarily to perform unification (i.e. to do PUTs when helpful). R$ is like REWRITE except that new free variables are suffixed with ‘$’ (possibly followed by an index). SK* removes all functions in the current goal that were defined using DEFN-SK, as described in some detail at the end of this subsection, while SK is a more selective version. UNIFY unifies the current subterm with a hypothesis. USE-LEMMA$ is like USE-LEMMA but treats all variables not bound by the supplied substitution as free variables. FORWARD$ is like FORWARD in that it does forward-chaining, but FORWARD$ also does unification, and the user does not have to supply a hypothesis number. GENERALIZE-SKOLEMS generalizes away calls of Skolem functions introduced by DEFN-SK (mentioned in the discussion of SK* above), and it can be especially useful for preparing a goal for a call to the prover in which metalemmas are expected to be used.

Now for the details, as provided by the help facility.

```
->: (help andsplit+ backchain r$ sk sk* unify use-lemma$ forward$ generalize-skolems)

Macro Command  [Use HELP-LONG to see the definition]
(ANDSPLIT+)
As with the ANDSPLIT command, if the current subterm is of the form
(AND x y ...), then a subgoal is created for each conjunct, and the current
goal becomes proved.  However, unlike ANDSPLIT, hypotheses are copied before
each subgoal is created, whenever the COPY command can succeed.  This way
there is less chance that one will find, later in a proof, that a variable to
be instantiated occurs in a goal other than the current one.
----------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(BACKCHAIN &OPTIONAL N)
(BACKCHAIN n):  The idea is that if hypothesis number n is active and of the
form (IMPLIES P Q), where Q is the current conclusion, this results in proving
the current goal but leaving one at a subgoal to prove P under the current
hypotheses.  However, BACKCHAIN is more powerful than that, in four ways.
     First of all, it suffices that Q unify with the current conclusion, i.e.,
it suffices that there be a PUT command that will make Q and the current
conclusion identical.
     Second, the indicated hypothesis can be of the form
(IMPLIES P1 (IMPLIES P2 (... (IMPLIES Pk Q)))), for any non-negative k.  In
particular, it can simply be Q.
     Third, the conditions above hold as well if Q is a conjunction of which
the current conclusion is one of the conjuncts (in the general sense that if Q
is (AND Q1 Q2), then everything above holds if it holds for either Q1 or Q2 in
place of Q).
     Finally, n need not be supplied, i.e. it is permissible to submit the
command BACKCHAIN with no arguments.  In that case, an attempt will be made to
find a suitable n, by looking through the hypotheses in reverse order until a
suitable one is found.
     NOTE:  one must be at the top of the current goal to run this command.
Also note that "bookmarks" will be placed around the instructions generated:
the first command generated will be (BOOKMARK (BEGIN (BACKCHAIN n k))), where
n is the index of the hypothesis used and k is as shown above, and the final
command will be (BOOKMARK (END (BACKCHAIN n k))).
----------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(R$ &REST ARGS)
Same as REWRITE, except that new free variables are suffixed with $ signs when
they don't already end in $ signs or $ signs followed by indices.
----------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(SK)
Assuming that the current term is a call of a DEFN-SK function FOO, and that
there is a lemma called FOO-SUFF or FOO-NECC (according to parity determined
by position of the current term), we want to replace the current term by the
appropriate Skolem axiom stored in FOO-SUFF or FOO-NECC.
----------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(SK* &OPTIONAL NOISY)
Applies SK to everything possible in the current goal (including subterms of
active hypotheses). With a non-NIL optional argument it prints out what it is
doing.
----------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(UNIFY &OPTIONAL N)
UNIFY:  Creates a PUT command that unifies some active top-level hypothesis
with the current subterm, if possible; fails otherwise.  (UNIFY n) specifies
that one should only try this with the nth hypothesis.
     See also BACKCHAIN.
----------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(USE-LEMMA$ LEMMA &OPTIONAL SUBST)
Like USE-LEMMA, except that variables in the lemma which are not explicitly
substituted for are brought in as free variables in the proof state.
```

```
---------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(FORWARD$)
Like forward, except no argument is required and instead, the system tries to
find (starting with the last hypothesis) an active hypothesis of the form
(IMPLIES p q) such that p unifies with some active hypothesis.  If the system
finds such hypotheses then it calls the FORWARD command appropriately after
doing any necessary unification.
---------------------------------------------------
Macro Command  [Use HELP-LONG to see the definition]
(GENERALIZE-SKOLEMS)
Generalizes away all terms whose function symbol was introduced as a Skolem
function by a DEFN-SK event.  This is probably a useful thing to do just
before entering the theorem prover if you want it to be able to take maximal
advantage of metafunctions.
->:
```

Finally, let us give a little extra attention to the command SK*, which is very convenient to use in conjunction with DEFN-SK (the new Boyer-Moore event currently available in some experimental enhancements, cf. [2]). In order for SK* to work, it is necessary that appropriately named lemmas have been proved. For example, after submitting the definition

```
(defn-sk arb-large-p (x)
  (forall big (exists bigger
                      (and (lessp big bigger)
                           (p x bigger)))))
```

one should prove the following two trivial lemmas in order for SK* to work on calls of ARB-LARGE-P. They are obtained by inspection of the formula printed out by the system in response to the DEFN-SK event. One of these is the ‘‘sufficiency’’ lemma, whose name is obtained by suffixing ‘‘-SUFF’’ to the end of the new function name, and which corresponds to the first conjunct of the formula printed out in response to the DEFN-SK command. The other is the ‘‘necessity lemma’’, whose name is obtained by suffixing ‘‘-NECC’’ to the end of the new function name, and which corresponds to the second conjunct of the formula printed out in response to the DEFN-SK command.[2]

[2]The INSTRUCTION hints shown below will always work for the -suff and -necc lemmas. In fact a macro DEFN-SK+ is in the CLI core image of pc-nqthm, and is available upon request, which creates these lemmas automatically.

```
(prove-lemma arb-large-p-suff
             (rewrite)
             (implies (and (lessp (big x) bigger)
                           (p x bigger))
                      (arb-large-p x))
             ((instructions promote (rewrite 2))))

(prove-lemma arb-large-p-necc
             (rewrite)
             (implies (not (and (lessp big (bigger big x))
                                (p x (bigger big x))))
                      (not (arb-large-p x)))
             ((instructions promote
                            (dive 1)
                            (rewrite 2)
                            top s)))
```

Having proved such lemmas for each function introduced by DEFN-SK, the SK* macro command has the following effect. Each call of such a function, in either an active hypothesis or the conclusion of the current goal, is replaced by a term in which that function has been eliminated by ‘‘opening it up’’ using the lemmas above (and paying attention to parity). Here is an example use. Notice that the -SUFF lemma above is used for *proving* that ARB-LARGE-P holds, while the -NECC lemma is appropriate for *using* the fact that ARB-LARGE-P holds. The variables suffixed with ‘$’ are newly-introduced free variables.

```
>(verify (implies (and (q x) (arb-large-p x))
                  (arb-large-p (add1 x))))

->: sk*

->: th

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

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

The current subterm is:
(IMPLIES (AND (Q X)
              (LESSP BIG$ (BIGGER BIG$ X))
              (P X (BIGGER BIG$ X)))
         (AND (LESSP (BIG (ADD1 X)) BIGGER$)
              (P (ADD1 X) BIGGER$)))
->:
```

4. Soundness

In this final section we start by showing why certain restrictions are necessary on the system’s use of free variables. We then review certain fundamental notions, especially about substitutions, and conclude by demonstrating soundness of the approach.

4.1 Examples illustrating restrictions on free variables

This subsection demonstrates the necessity of having some restrictions on the handling of free variables. All input in the examples below appears on lines following the prompt, '**->:** '; the rest is output. Let us consider various commands in turn.

GENERALIZE.

Consider the following ''proof'' of **(EQUAL T F)**. What's going on here is that we're trying to prove a falsehood by generalizing away an expression with a free variable and then later instantiating that variable with PUT. As the ''proof'' below shows, it is important not to allow such a sequence of steps.

```
->: p

(EQUAL T F)
->: show-rewrites

1. SILLY
     New term: T
     Hypotheses: ((LESSP (ADD1 $Z) $Z))
->: (rewrite 1 (($z z)))

Rewriting with SILLY.

NOTE:  The variable Z is being introduced into the proof state and is
therefore being added to the current list of free variables.

Creating 1 new subgoal, (MAIN . 1).

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

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

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

The current subterm is:
(LESSP (ADD1 Z) Z)
->: (generalize (((add1 z) a)))

The goal (MAIN . 1) has been generalized to the new goal ((MAIN . 1) . 1),
which is now the current goal.
```
{Here's the message indicating the removal of **Z** *from the free variables of the proof state.}*
```
     WARNING:  The variable Z is being removed from the list of free variables
of the current proof state, for soundness reasons, because it has been
generalized away but still "interacts" with free variables in the resulting
goal.  (Use (HELP-LONG GENERALIZE) for a further explanation.)

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

((MAIN . 1) . 1)
```

```
->: th

*** Active top-level hypotheses:
H1.  (NUMBERP A)

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

The current subterm is:
(LESSP A Z)
->: (put (z (add1 a)))
```
{This had better not work, or we'd be able to finish the proof!!}

```
**NO CHANGE** -- The variable Z is not in the list of free variables of the
current state.
->:
```

INDUCT

Let's see now why having certain restrictions on how the set of free variables interacts with the INDUCT command are necessary for the soundness of the system. One such restriction is labeled (2) in the help printed in response to (HELP-LONG INDUCT):

```
     (2) If there are any free variables of the state that occur in the
current goal, these will be removed from the free variables list.  This is
necessary for soundness.
```

Now suppose that we have defined the notion IS-SQUARE of a perfect square (for natural numbers), and that we have the following rewrite rule IS-SQUARE-SUFFICIENCY:

```
(IMPLIES (EQUAL (TIMES Y Y) X)
         (IS-SQUARE X))
```

Also suppose that we want to prove that every natural number is a perfect square (which is obviously absurd). If we enter the proof-checker with the command

```
(verify (implies (numberp n) (is-square n)))
```

and proceed with the commands PROMOTE and REWRITE, we have a single goal and a single free variable, **Y**:

```
*** Active top-level hypotheses:
H1.  (NUMBERP N)

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

The current subterm is:
(EQUAL (TIMES Y Y) N)
```

If we then apply the command (INDUCT (PLUS N Q)), or any similar induction on N, then we find ourselves with the following two goals:

```
(IMPLIES (ZEROP N)
         (IMPLIES (NUMBERP N)
                  (EQUAL (TIMES Y Y) N)))

(IMPLIES (AND (NOT (ZEROP N))
              (IMPLIES (NUMBERP (SUB1 N))
                       (EQUAL (TIMES Y Y) (SUB1 N))))
         (IMPLIES (NUMBERP N)
                  (EQUAL (TIMES Y Y) N)))
```

Our specification of the INDUCT command requires the removal of **Y** from the list of free variables at this point. Suppose however that we did not have this restriction. Then the substitution replacing **Y** by **(IF (ZEROP N) 0 N)** would result in each of the two goals above being provable, thus completing the proof!

Here is an example showing the necessity of another restriction for the INDUCT command printed in response to (HELP-LONG INDUCT):

```
     (3) If there are any free variables of the state
occurring in the variables of a command of the form (INDUCT (g v1 ... vn)),
these will be removed from the free variables list.
```

Here is how to prove a non-theorem if that restriction is entirely removed. Consider the following valid but silly rewrite rule:

```
(prove-lemma funny (rewrite)
  (implies (and f (not (zerop n)))
           (equal (equal x f) t)))
```

Suppose we attempt to verify **(EQUAL T F)**, with our first step be to REWRITE using the rule FUNNY above. Then we obtain three subgoals:

```
->: print-all-goals


(MAIN . 1)
F

(MAIN . 2)
(NUMBERP N)

(MAIN . 3)
(NOT (EQUAL N 0))
->:
```

Suppose we work on the first of these by issuing the command (INDUCT (PLUS N Q)). Then our proof state looks as follows:

```
->: print-all-goals

((MAIN . 1) . 1)
(IMPLIES (ZEROP N) F)

((MAIN . 1) . 2)
(IMPLIES (AND (NOT (ZEROP N)) F) F)

(MAIN . 2)
(NUMBERP N)

(MAIN . 3)
(NOT (EQUAL N 0))
```

If **N** is not removed from the list of free variables, then we can instantiate **N** to be 1 via the command (PUT (N 1)), at which point all of the goals become provable!

Notice, by the way, that the first of these two INDUCT examples does not violate restriction (3), while the second doesn't violate restriction (2). That is, neither restriction alone is enough to guarantee soundness, which seems to leave us with no obvious way to weaken the requirements for INDUCT.

4.2 A Review of Some Technical Notions

In the following discussion, **|-** denotes provability with respect to the current Boyer-Moore history (chronology). It is handy to introduce some notation and standard terminology.

Review of facts about substitutions.

A *substitution* is simply a function mapping terms to terms. For a term **t** and substitution **s**, we write **t/s** to denote the result of substituting **s** into **t** (which is defined in the usual way). It also makes sense to substitute a substitution s_2 into a given substitution s_1 by substituting s_2 into every element of the range of s_1:

$$s_1 \mathbin{//} s_2 = \{\langle x, y/s_2\rangle : \langle x,y\rangle \in s_1\}$$

Next let us introduce notation for the (more or less standard) notion of the *composition* of substitutions s_1 and s_2, thought of as the substitution obtained by first applying s_1 and then applying s_2:

$$s_1 \mathbin{@} s_2 = (s_1 \mathbin{//} s_2) \cup \{\langle x,y\rangle \in s_2 : x \notin \mathrm{domain}(s_1)\}.$$

We will use the following composition rule for substitutions, which relates the above notion of composition to the usual functional notion. Notice that we write **t/s/s'** as shorthand for **(t/s)/s'** .

LEMMA (*Composition rule for substitutions*). For any term **t** and substitutions **s** and **s'**, we have:

`t/s/s' = t/(s @ s').` -|

Let us conclude our review of substitutions with one more simple observation.

LEMMA (*restricting substitutions*). Let **t** be a term and let **s** be a substitution whose domain is a set of variables. Suppose that **A** is a subset of the domain of **s** such that every variable occurring in **t** which belongs to the domain of **s** in fact belongs to **A**, and let s_0 be the restriction of **s** to **A**. Then **t/s** is identical to to t/s_0. -|

Let us move on now to some notions related to PC-NQTHM in particular.

DEFINITION. A *provable state* is a proof state for which there is a substitution **s**, with domain equal to (equivalently, contained in) the set of free variables in that state, such that if **P** is the conjunction of the goals of that state, then **P/s** is a theorem of the current history.

Notice that an equivalent definition results if we allow the domain of **s** to be any *subset* of the set of free variables, since we can extend such a substitution by {**<v,v>:** **v** is a free variable not in **domain(s)**}.

''Soundness'' means simply that any state that can be brought to completion by proof-checker commands, in the sense that the final goals all have true conclusions, is ''provable'' in this sense. To make this notion of soundness a bit more precise, let us recall the following definition from Appendix B of [1]. (Disclaimers regarding the sense in which certain issues are avoided may be found in that appendix. They apply here but will not be repeated here.)

DEFINITION. A *valid state stack* is a state stack that can be produced from an interactive session that begins with a call of the form **(VERIFY <term>)** and then results from the execution of a sequence of change commands.

4.3 Soundness Theorem

We wish to prove the following theorem, which extends the theorem in Appendix B of [1] to the setting of the new proof-checker. It implies that in particular, if all the goals have true conclusions at the end of a proof, then the original goal (which of course never has free variables) is a theorem. That is, the proof-checker only certifies theorems of the Boyer-Moore logic.

THEOREM. If the top (most recent) proof state in a valid state stack is a provable state, then so is the bottom (original) state in that state stack.

We will prove this theorem by induction on the length of the valid state stacks. First let us note (without proof) that all commands except INDUCT, PUT, COPY, and GENERALIZE have the following property (which is stronger than the property (*) on p. 56 of the manual, [1]):

(+) A goal `G` is replaced by a goal `G'` (which may equal `G`) and zero or more new goals `GG` such that `|- GG & G' -> G`.

In order to prove the theorem by induction, then, suppose that we have a valid state stack and that the theorem holds for all shorter state stacks. Assume that the top state is provable. If the state stack has length 1, then we're done, so assume that there are at least two states in the stack. By the inductive hypothesis, it suffices to prove that the next-to-top state is provable. That state is of the form `(G & R)`, where `G` is its current goal and `R` is the conjunction of the rest of its goals. Then the top state may be written as `(GG & G' & R)`, where here `G'` is the new version of `G` in that state and `GG` is the conjunction of the newly-created subgoals of `G`, and `R` is the conjunction of the remaining goals. Since the top state is provable (by hypothesis), there is a substitution `s` on its set of free variables such that `|- (GG & G' & R)/s`. Hence if the final instruction is other than INDUCT, PUT, COPY, or GENERALIZE, then by (+) we have `|- (G & R)/s`. This concludes the proof (except for these four commands), except that we should observe that the commands REWRITE, USE-LEMMA, and CLAIM can introduce new free variables into a state, and hence the domain of `s` may properly include the set of free variables of the next-to-top state; but in that case, we simply observe that `|- (G & R)/s'`, where `s'` is the restriction of `s` to the set of free variables of the next-to-top state. It remains then to handle the cases where the final instruction is INDUCT, PUT, COPY, or GENERALIZE.

INDUCT. This case is clear -- we may take `s'` to equal `s` -- since the restriction on INDUCT guarantees that no free variable occurs in `G` or in `GG`. For that condition implies that `GG/s` equals `GG` and `G/s` equals `G`. Now the Boyer-Moore logic supports INDUCT in the sense that from `|- GG` we get `|- G`, and the conclusion follows. (In practice the INDUCT command allows free variables in `G`, but first removes them from the free variables list before completing the command.)

PUT. The PUT command takes a substitution `s0` whose domain is a subset of the free variables FRV of the current state, and creates a new state with a set FRV1 of free variables. Let `P` be the conjunction of the goals in the next-to-top state, i.e. `P` is `(G & R)`. By hypothesis, there is a substitution `s` with domain FRV1 such that `|- (P/s0)/s`. It suffices to show that there is a substitution `s'` with domain FRV such that `|- P/s'`. Let `s1` be the composition of `s0` with `s`. Then `P/s1` = `P/s0/s` by the composition rule, and that's almost all there is to it. The problem is that we need a substitution with domain FRV, yet `s1` has domain equal to the union of the domains of `s` and `s0`, i.e. the union of FRV1 with the domain of `s0`. However, since the domain of `s0` is contained in FRV, and since the only variables in FRV1 which are not in FRV are variables that do not occur in `P`, then the restriction `s2` of `s1` to FRV has the property that `P/s2` = `P/s1`. Thus we may let `s'` be `s2`.

COPY. Consider a version of COPY in which one specifies a single variable to be copied (rather than requiring all possible variables to be renamed in the copy and perhaps specifying a hypothesis). We note that it suffices to prove soundness for this version of COPY. For if we can do so, and then we want to prove soundness with respect the official version of COPY (again by induction on the state stack), we simply note that every valid state stack in the official sense is a subsequence of a valid state stack in this modified sense. To see this, note that if `P` is the hypothesis to be copied, then by a sub-induction hypothesis we may copy `P` with respect to all but one variable; call the copy `P'`. Now copy `(OTHER-HYPS & P & P')` with respect to the remaining variable to get `(OTHER-HYPS & OTHER-HYPS' & P & P' & P'' & P''')`, and then drop `OTHER-HYPS'`, `P'`, and `P''`. (If the new variable doesn't have the desired name, that's an easy matter to fix using PUT.) It remains then only to prove the following theorem. Notice that the restriction on `v` corresponds to the restriction on COPY regarding which variables may be renamed.

COPY SOUNDNESS THEOREM. Let `P`, `C`, and `R` be terms, and suppose that `v'` is a variable not occurring in any of these. Let `P'` be `P/{<v,v'>}`, where `v` is a variable other than `v'` that does not occur in `C` or `R`. Also suppose that `s` is a substitution with domain `D ∪ {v,v'}`, for a set `D` not containing `v` or `v'`, such that `|- ([(P & P') -> C] & R)/s`. Then there is a substitution `s1` with domain `D ∪ {v}` such that `|- ([P -> C] & R)/s1`.

Proof. Since `v` does not occur in `C` or `R`, it suffices to prove the following lemma. Think of `s0` below as being the restriction of `s` above to `D`, of `s1` as being `s1`, of `s2` as being `s`, of `u` as being `s(v)`, and of `u'` as being `s(v')`. `-|`

LEMMA. Let `P` be a term, let `s0` be a substitution with domain not including either of the distinct variables `v`, `v'`, and suppose that `v'` does not occur in `P`. Let `u` and `u'` be terms, let `P0` be `P/(s0 ∪`

`{<v,u>})`, let `s1` be `s0 ∪ {<v, (IF P0 u' u)>}`, let `s2` be `s0 ∪ {<v,u>,<v',u'>}`, and let `P'` be `P/{<v,v'>}`. Then the following is a theorem:

(A) `|- P/s1 <-> [P & P']/s2`.

Proof. First note that by the lemma on restricting substitutions, since `v'` does not occur in `P` we have

(1) `P/s2 = P0`

Also, a simple computation using the lemma on restricting substitutions and the composition rule for substitutions shows:

(2) `P'/s2 = P/(s0 ∪ {<v,u'>})`.

Consider what happens when we split into cases according to `P0`. When `P0` holds, `P/s1` reduces to `P/(s0 ∪ {<v,u'>})`, and (1) and (2) immediately imply (A) in this case. When `P0` fails, `P/s1` reduces to `P0`, which by this case hypothesis reduces to `F` (false), and hence by (1) we again have (A). `-|`

GENERALIZE. This part of the soundness proof is necessarily rather technical. A corresponding mechanically-checked proof has been performed using (pc-)nqthm and will appear in [7].

First, let us point out a precondition on the success of the GENERALIZE command, as reported by (HELP GENERALIZE):

```
(GENERALIZE ((term1 V1) ... (termn Vn))):  Replace each of the given terms by
the indicated corresponding new variable, which must not occur anywhere in the
current proof state.
```

Now, let us specify in some detail what happens when one has a proof state `ps` and applies the GENERALIZE command with substitution `sg` to obtain a new proof state `ps'`. We will pay particular attention to how this command relates to free variables.

- Fix a proof state `ps`.
- Let `sg` be a one-to-one substitution mapping variables to terms.[3]
- Let `ps'` be the result of applying the GENERALIZE command, with substitution `sg` mapping new variables to terms. Thus, the new current goal is the result of substituting the inverse `sg`$^{-1}$ of `sg` into the current goal of `ps`.

---

[3] In the implementation, the user may specify a substitution in which the same term is generalized to more than one variable. However, in such a case one should think of `sg` as noticing only the first occurrence of that term in the substitution.

- Let **FREE** and **FREE'** be the respective sets of free variables of **ps** and **ps'**.
- Consider the binary relation $\mathbf{R_0}$ defined on **FREE** as follows: $\mathbf{R_0}(\mathbf{v},\mathbf{w})$ if and only if **v** and **w** occur in a common goal of **ps'**.
- Let **R** be the transitive closure of $\mathbf{R_0}$.
- Let **C** be the range of **R** on the intersection of **FREE** with the variables of the current goal in **ps'**.
- Let **V** be the set of variables that occur in the range of **sg**.

Loosely speaking, we want to remove from **FREE** the set $(\mathbf{C} \cap \mathbf{V})$ consisting of all variables from **FREE** that both occur in somewhere in the terms being generalized away *and* also have ''anything to do with'' the new current goal (where ''anything to do with'' is defined in terms of the equivalence relation **R**). We also want to make sure the none of the new variables is considered free.[4] The precise relationship specified between **FREE** and **FREE'** is as follows.

$$\mathbf{FREE'} = (\mathbf{FREE} \setminus (\mathbf{C} \cap \mathbf{V})) \setminus (\mathbf{domain\ sg})$$

Here then, finally, is what we need to prove, using the notation in Subsection 4.2 above.

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

Proof. We continue using the notation above, as well as the notation introduced in Subsection 4.2 above. Let $\mathbf{s_1}$ be the restriction of **s'** to **C** (recall that **C** is defined above); let $\mathbf{s_2^0}$ be the restriction of **s'** to the complement of **C**; let $\mathbf{s_{triv}}$ be any substitution with domain equal to the domain of **sg** such that its range has no occurrences of variables; and let $\mathbf{s_2} = \mathbf{s_2^0}\ //\ \mathbf{s_{triv}}$. Finally we can define the desired substitution **s** as follows:

$$\mathbf{s} = (\mathbf{s_1} \cup \mathbf{s_2})\ //\ (\mathbf{sg}\ //\ \mathbf{s_2})$$

[4]This would be automatic if we were to require not only that **domain(sg)** to be disjoint from the set of variables occurring in the current proof state, which we do, but also that every variable in **FREE** occurs in the current proof state. But we do not bother to enforce the latter.

Notice that **s** and **s′** have the same domain, and hence the domain of **s** is a subset of **FREE**. It remains to show **|- G/s** and **|- P/s**.

Let us first show **|- G/s**. Since we know **|- G′/s′** by hypothesis, then by the rule of instantiation it suffices to prove the following two facts.

(1) $\mathbf{G/s = G'/(s_1 \cup s_2)/(sg\ //\ s_2)}$

(2) $\mathbf{G'/(s_1 \cup s_2) = G'/s'}$

Equation (1) follows by structural induction on **G** and the definition of **s**. We omit the proof here, as the details are very similar to those in various arguments below. For (2), let us first observe that by definition of **C**, every variable of **FREE** that occurs in **G′** is a member of **C**. Therefore, every member of **domain(s′)** that occurs in **G′** is a member of $\mathbf{domain(s_1)}$. It follows from the lemma on restricting substitutions (in Subsection 4.2 above) that both sides of equation (2) are equal to $\mathbf{G'/s_1}$.

It remains to show **|- P/s**. Fix an arbitrary conjunct **Q** of **P**. First, we claim:

(3) $\mathbf{Q/(s_1 \cup s_2)/(sg\ //\ s2) = Q/s}$

Here is a proof of (3).

$\mathbf{Q/(s_1 \cup s_2)/(sg\ //\ s2)}$
= *{by the composition rule for substitutions}*
$\mathbf{Q/((s_1 \cup s_2)\ @\ (sg\ //\ s2))}$
= *{definition of* **@***, since* **sg** *and* $\mathbf{(s_1 \cup s_2)}$ *have disjoint domains}*
$\mathbf{Q/(((s_1 \cup s_2)\ //\ (sg\ //\ s2)) \cup (sg\ //\ s2))}$
= *{by the lemma on restricting substitutions and the hypothesis that no variable in* **domain(sg)** *occurs in the proof state}*
$\mathbf{Q/((s_1 \cup s_2)\ //\ (sg\ //\ s2))}$
= *{definition of* **s***}*
Q/s

Finally, we claim that

(4) $\mathbf{Q/(s_1 \cup s_2)}$ is an instance of **Q/s′**.

Assume this claim for the moment. Now the composition rule for substitutions implies that the ‘‘is an instance of’’ relation is transitive on terms. It therefore follows from (3) and (4) that

(5) **Q/s** is an instance of **Q/s′**.

Since we know $\texttt{|- P/s'}$ (by hypothesis), and since $\mathbf{Q}$ is a conjunct of $\mathbf{P}$, it follows that $\mathtt{Q/s'}$, and hence by the rule of instantiation and (5) we have that $\texttt{|- Q/s}$. Since $\mathbf{Q}$ is an arbitrary conjunct of $\mathbf{P}$, we have shown that $\texttt{|- P/s}$. Hence, it remains only to prove (4).

There are two cases. First suppose that $\mathbf{Q}$ contains a variable of $\mathbf{FREE}$ that is $\mathbf{R}$-equivalent to a variable of $\mathbf{FREE}$ occurring in $\mathbf{G'}$. In that case, every variable of $\mathbf{FREE}$ occurring in $\mathbf{Q}$ is in $\mathbf{C}$ (since $\mathbf{R}$ is an equivalence relation), and hence is in the domain of s_2. It follows from the lemma on restricting substitutions that $Q/(s_1 \cup s_2)$ and Q/s' are both equal to Q/s_1. The other case is where $\mathbf{Q}$ does not contain any variable of $\mathbf{FREE}$ that is $\mathbf{R}$-equivalent to a variable occurring in $\mathbf{G}$. Since every member of the domain of s_1 is $\mathbf{R}$-equivalent to a variable occurring in $\mathbf{G}$, it follows that:

(6) No variable of $\mathbf{Q}$ is in the domain of s_1.

Therefore $Q/(s_1 \cup s_2) = Q/s_2$. Recall the notation above defining s_2 to be $s_2^0 \ // \ s_{triv}$. It suffices to show that $Q/s_2 = Q/s'/s_{triv}$, which we do now.

$Q/s'/s_{triv}$
= *{by the composition rule for substitutions, and since the domain of* s_{triv} *equals the domain of* $\mathtt{sg}$*, which is disjoint from the domain of* s'*}*
$Q/((s' \ // \ s_{triv}) \cup s_{triv})$
= *{by the lemma on restricting substitutions, since by hypothesis, no variable of* $\mathbf{Q}$ *is in the domain of* $\mathtt{sg}$*, i.e. in the domain of* s_{triv}*}*
$Q/(s' \ // \ s_{triv})$
= *{by the lemma on restricting substitutions and (6)}*
$Q/(s_2^0 \ // \ s_{triv})$
= *{by definition of* s_2*}*
Q/s_2

-|

Appendix A
Other Changes

In this appendix we document some of the main changes in the current version of PC-NQTHM from the version documented in [1], other than the changes pertaining to free variables (which are already discussed above). The list below provides brief summaries only. The help facility should be used to get more information, e.g. submit (HELP FORWARD) to PC-NQTHM to obtain more information about FORWARD. These are in no particular order.

1. There is now a global variable **`*pc-nqthm-version-number*`**. As of March 7, 1990, this version is 1.1. (However, in some Lisps this number could be slightly different, e.g. 1.1000000000000001 -- !!)
2. FORWARD has been improved from an earlier version so that it tries to prove the new goals generated and reports which new goals are unproved.
3. The GENERALIZE command now uses GENERALIZE lemmas (see page 248 of [6]).[5] (Use the help facility for details.)
4. REPEAT has been changed so that if all goals are proved then it's considered a ''success'' (in the sense of ''success'' and ''failure'' for commands, cf. [1]).
5. USE has been modified so that it warns the user when uninstantiated variables remain.
6. IFSPLIT is a new command that can be used to case split on the test of an IF expression.
7. BASH no longer automatically prints out the new goals created. However, the macro command PGBASH (mnemonic for print-goals-bash) does print out the new goals, in case you really want that feature. BASH now does one thing it didn't do before, namely inform you which rules and definitions were used.
8. The command (CASESPLIT **`exp`**) does a case split according to a user-supplied term **`exp`**.
9. ELIM (destructor elimination) now allows, but does not require, the new variable names for the generalized terms to be supplied by the user.
10. ADD-ABBREVIATION can now be given a single argument, a variable. Then the term to be abbreviated is the current subterm.
11. The command (S NIL) is now legal, meaning simplify without opening up any function or using any rewrite rules. (It always was legal in principle, but a bug prohibited its use.)
12. COMM prints out commands in a way that hides commands set between bookmarks, and thus hides uninteresting commands generated by many macro commands.
13. UNDO! is very handy for undoing top-level macro commands in a way that corresponds closely to how commands are displayed using COMM (described just above).
14. The SUBV command (which hardly anyone ever uses) now requires you to be at the top of the conclusion, and is smarter about which hypotheses to use. It also now prints out the substitution that it finds. Details may (as usual) be found using the help facility.
15. The EXIT command can be given an extra non-NIL argument, meaning ''do not prompt upon exit but just go ahead and create the event.''
16. The extension of the syntax to include COND, CASE, and LET now also includes LIST*, where $\mathtt{(LIST*}\ x_1\ x_2\ \ldots\ x_n\mathtt{)}$ abbreviates $\mathtt{(cons}\ x_1\ \mathtt{(cons}\ x_2\ \ldots\ \mathtt{(cons}\ x_{n-1}\ x_n\mathtt{)}\ \ldots\ \mathtt{))}$.

[5] The utility of this change was brought to our attention by Matt Wilding.

Therefore, you can use backquotes with ‘‘,@’’ in your event forms in certain Lisps (e.g. akcl). However, in analogy to how LET is handled, LIST* will never appear in terms pretty-printed by the system.[6]

17. The proof-checker’s INDUCT command formerly differed from the Boyer-Moore theorem prover’s heuristics as follows: the theorem prover ignored induction schemes corresponding to disabled functions, but the proof-checker’s INDUCT did not. That discrepancy has been eliminated.
18. SHOW-REWRITES now informs you if a rewrite rule is disabled.
19. The macro command PRO is a ‘‘smarter’’ version of PROMOTE.
20. PRUNE is similar to UNDO, except that some attempt is made to save that part of the proof which doesn’t ‘‘depend on’’ the command that was undone.
21. SHRINK attempts to shorten the proof by compressing commands. For example, the sequence consisting of (DIVE 2) followed by UP would be canceled by SHRINK, and consecutive CHANGE-GOAL commands would be combined.
22. EX is like EXIT, but runs SHRINK first.
23. The variable RESTART-STACK-DEPTH is set to 0 if the underlying Lisp is not a version of kcl (e.g. akcl). What this means is that you’ll need to submit (VERIFY) to go back into the proof checker after an interrupt. It was discovered that Symbolics Lisp machines can throw you into the cold load stream otherwise. (But akcl makes special provisions, which is why I’ve left the value of this variable at 8 for kcl.) One may feel free to set RESTART-STACK-DEPTH to another integer value, of course; see [1] for documentation.
24. The SPLIT command has been made more efficient at the cost of a very slight weakening of the heuristics for checking if the goal is already valid. I’d be surprised if anyone notices any difference here.
25. The CASE patch to TRANSLATE has been fixed so that OTHERWISE behaves properly (thanks to Larry Smith).
26. The equality substitution macro commands EQSUB and EQSUB-R now substitute into the hypotheses (in addition to the conclusion) and then drop the equality that has been used.

[6] For nqthm hackers: that is, although TRANSLATE has been modified to support LIST*, UNTRANSLATE has not.

References

1. Matt Kaufmann, ‘‘A User’s Manual for an Interactive Enhancement to the Boyer-Moore Theorem Prover’’, Tech. report 19, Computational Logic, Inc., May 1988.

2. Matt Kaufmann, ‘‘DEFN-SK: An Extension of the Boyer-Moore Theorem Prover to Handle First-Order Quantifiers’’, Tech. report 43, Computational Logic, Inc., June 1989.

3. W.W. Bledsoe, P. Bruell, ‘‘A Man-Machine Theorem-Proving System’’, *Advance Papers of Third International Joint Conference on Artificial Intelligence*, W.W. Bledsoe, 5-1 (Spring) 1974.

4. D.I. Good, R.L. London, W.W. Bledsoe, ‘‘An Interactive Program Verification System’’, *Proceedings of 1975 International Conference on Reliable Software*, D.I. Good, 1975.

5. Bill Young, ‘‘Using the GVE: Examples of Proof Commands’’, Tech. report 8, Computational Logic, Inc., June 1987.

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

7. Matt Kaufmann, ‘‘A Mechanically-checked Correctness Proof for Generalization in the Presence of Free Variables’’, Tech. report 53, Computational Logic, Inc., to appear.

Table of Contents