Automated Reasoning With Quantified Formulae

David Greve
Rockwell Collins Advanced Technology Center
Cedar Rapids, IA
dagreve@rockwellcollins.com

ABSTRACT

ACL2 allows users to define predicates whose logical behavior mimics that of universally or existentially quantified formulae. Proof support for such quantification, however, is quite limited. We present an ACL2 framework that employs tables, computed hints and clause processing to identify quantified formulae and to skolemize or instantiate them when possible. We demonstrate how the framework can be used to prove automatically the `forall-p-append` example presented in the ACL2 documentation.

Categories and Subject Descriptors

F.4.1 [**Mathematical Logic**]: Mechanical Theorem Proving

General Terms

Quantification

Keywords

ACL2, Skolemization, Instantiation

1. QUANTIFICATION IN ACL2

Quantification in ACL2 is a second class citizen in a first order world. It suffers from deficiencies in both expression and reasoning support. ACL2 is commonly referred to as being "quantifier-free" in deference to the fact that the logic of ACL2 does not include explicit quantification constructs. Rather, ACL2 exploits that fact that it is possible to use a choice principle in conjunction with the definitional principle to introduce a predicate whose logical behavior mimics that of a quantified formula. This approach, however, precludes the introduction of locally quantified formulae during function or theorem definition and makes it difficult to mix quantification and recursion[4]. There is no support for automated reasoning about quantified formulae built in to ACL2, a condition exacerbated by the lack of explicit quantification constructs. Because quantification is necessarily hidden in the properties of various function symbols, it is very difficult for a user to examine a failed proof attempt and discern how quantified formulae might be manipulated to achieve a desired result. Over the years such factors have conspired to help ensure that the majority of ACL2 proof efforts remain quantifier-free.

ACL2 does, however, provide a number of means by which sufficiently motivated users may soundly extend the reasoning power of the system without modifying the trusted core. In this paper we discuss how tables, computed hints and clause processing have been combined to construct a reasoning infrastructure that enables ACL2 to identify and automatically skolemize or instantiate quantified formulae appearing in the goal of a proof effort. We demonstrate how the framework can be used to prove automatically the `forall-p-append` example presented in the ACL2 documentation and suggest some possible future enhancements.

2. INTRODUCTION

The macro `defun-sk` may be used to introduce a quantified formula in ACL2. Note that, as in this case, quantified formulae in ACL2 may have formal parameters in addition to quantified variables.

```
(defun-sk forall-p (x)
 (forall (a) (implies (member a x) (p a))))
```

Under the hood this macro generates a sequence of ACL2 events. The first is a `defchoose` event that employs a choice principle to witness the function `(forall-p-witness x)` with the property that, if there is a value for `a` that makes the predicate `(implies (member a x) (p a))` false, this function will too. This function symbol is then employed as the quantified variable in the definition of the `(forall-p x)` predicate. The final theorem, in this case `forall-p-necc`, is the *quantification theorem* which captures the logical behavior of the quantified formula.

```
(defchoose forall-p-witness (a) (x)
  (not (implies (member a x) (p a))))

(defun forall-p (x)
  (let ((a (forall-p-witness x)))
    (implies (member a x) (p a))))

(defthm forall-p-necc
 (implies (forall-p x) (implies (member a x) (p a))))
```

The procedure for existential quantifiers is similar except that the hypothesis and conclusion of the resulting quantification theorem are swapped. Note that because `defun-sk` is a top-level event (actually a sequence of events) it cannot be used to introduce quantification in the body of a function or theorem definition.

Instantiation is the name given to the process of deriving a new formula from a universally quantified formula by replacing each occurrence of a quantified variable with a specific instance. In ACL2, instantiation is performed by way of a `:use` hint applied to the quantification theorem governing the behavior of the terms involved.

```
(defthm instantiation-example
 (implies
   (and (forall-p y) (member b y)) (p b))
  :hints (("Goal" :use (:instance forall-p-necc
           (x y)     ;; Formal Parameter
           (a b))))) ;; Quantifier Instantiation
```

Skolemization, as implemented in this library, is the process of deriving a new formula from a quantified formula by replacing each occurrence of the quantified variable with a new variable that is free in the current context. Since quantified variables already appear as function symbols in ACL2, what we call skolemization is actually just generalization. Our implementation of skolemization is sound for any quantified formula but it is typically used only to simplify existentially quantified formulae.

The sense in which a quantified formula appears in the goal, either true or negated, is important. Negated existentially quantified formulae in the hypothesis act like universally quantified formulae in that they may be soundly instantiated. Negated universally quantified formulae in the hypothesis, however, are susceptible only to skolemization. The converse is true for quantified formulae in the conclusion.

3. IDENTIFICATION

Consider the results of a failed proof in which the universally quantified formula `forall-p` is a hypothesis.

```
ACL2 !> (defthm forall-p-proof
  (implies (forall-p y) (implies (member b y) (p b))))

Subgoal 2
(IMPLIES
  (AND (NOT (MEMBER (FORALL-P-WITNESS Y) Y))
       (MEMBER B Y)) (P B))

Subgoal 1
(IMPLIES
  (AND (P (FORALL-P-WITNESS Y))
       (MEMBER B Y)) (P B))
```

Evident in this example are several of the challenges that must be overcome in order to automate reasoning about quantification in ACL2. The first challenge is recognizing specific quantified predicates. Each quantified predicate is associated with a unique witness function that is used in place of the quantified variable. In `forall-p` the function `(forall-p-witness Y)` replaces the quantified variable `A`. Then quantified predicates must be recognized from their constituent components. The structure of a quantified formula is likely to change dramatically under simplification. In this case, the `(member a x)` portion of the quantified formula appears in Subgoal 2 while the `(p a)` portion appears in Subgoal 1. The fact that each subgoal implicitly contains `(forall (a) (implies (member a x) (p a)))` as a hypothesis is not at all obvious from casual observation. Finally, the polarity (either true or negated form) of the quantified predicate must be established. While the appearance of a quantification witness is often an indication of the presence of a quantified formula, the witness itself does not establish the polarity in which the quantified formula appears. This is important because the polarity of the formula determines whether it is susceptible to instantiation or skolemization.

Overcoming each of these challenges requires intimate knowledge of the various quantified predicates that might be present in a goal. Automating this process requires that this knowledge be available dynamically at proof time. To solve this problem we employ ACL2 tables. Tables allow us to store information in the ACL2 logical world from the time the quantification is introduced to the time it is used in a subsequent proof effort. Additionally, tables are available to computed hints so the information can be queried dynamically as the proof progresses.

The framework that we have developed records information about quantified formulae at the time of their creation through a macro called `def::un-sk` which is a wrapper around ACL2's `defun-sk`. The macro is capable of processing any `defun-sk` compliant quantification and honors all of the `defun-sk` keywords. The information we record about a quantified formula includes the name of the quantified predicate, the kind of quantification (existential or universal), the quantified variable list, the formal parameter list, the name of the quantification theorem (`-necc` or `-suff`), the name of a skolemization rule, the body of the quantified formula, and the witness function name. A list is used to aggregate this data and each time a new quantifier is introduced using `def::un-sk` the information about that quantified predicate is added to the list. The macros `TABLE::GET` and `TABLE::-SET` simplify access and updating of table entries.

```
(TABLE::SET
  'QUANT::QUANT-LIST
 (CONS (NEW-QUANT
          :NAME 'FORALL-P
          :TYPE ':FORALL
          :BOUND '(A)
          :FORMALS '(X)
          :INSTANTIATE 'FORALL-P-NECC
          :SKOLEMIZE 'FORALL-P-SKOLEMIZATION
          :BODY '(IMPLIES (MEMBER A X) (P A))
          :WITNESS '(FORALL-P-WITNESS X))
   (TABLE::GET 'QUANT::QUANT-LIST)))
```

Quantification specific computed hints query this table at proof time to secure a list of candidate quantified formulae.

The list is used to search for quantified formulae in the current goal. A goal is said to contain a quantified formula if it contains a logical instance of that formula. In the case of `forall-p` we search for logical instances of `(forall-p x)`. The failed proof above contained the instance `(forall-p y)`, but `(forall-p y)` was rewritten into `(implies (member (forall-p-witness y) y) (p (forall-p-witness y)))))` and the result was simplified propositionally to produce two subgoals. Consequently the process of searching for formulae instances involves more than just pattern matching against the goal. The process must, at a minimum, take into consideration propositional simplification and in practice it must also consider the impact of rewriting. Bash[2] is used to simplify formulae before and during pattern matching. This method is strictly more powerful than that employed by Davis' pick-a-point strategy[1], though perhaps similar to the method employed by Moore in his `:consider` hints library[3].

4. INSTANTIATION

Discovering an instance of a quantified expression is equivalent to discovering a unifying binding for its formal parameters[5]. Instantiation requires that suitable bindings also be found for the free (quantified) variables appearing in the quantification theorem. The quantification theorem for the `forall-p` predicate appears as follows:

```
(implies (forall-p x) (implies (member a x) (p a)))
```

Given a binding for the formal `x`, a suitable instance of `a` must also be identified. This may be done by searching the goal for the term of the form `(member a x)` for the given value of `x`. However, an instance of the pattern `(not (p a))` would also satisfy the (contrapositive form of the) formula:

```
(implies (forall-p x)
         (implies (not (p a)) (not (member a x))))
```

Since the form of quantified formulae is unconstrained we are compelled to search for every such logical permutation of a formula. This in addition to the issues of simplification and rewriting discussed above. Finally, to ensure that each instantiation adds information to the goal, we drop any instantiations in which the conclusion of the fully instantiated formula already appears the goal.

It is possible that this process will identify several viable instantiations. To avoid overwhelming ACL2, instantiations are introduced one at a time, with time for the goal to stabilize between each one. It is further possible that instantiation will be needed multiple times within a single proof. Such re-entrant hints are made possible thru the use of the computed hint replacement feature which can be used to ensure that the process continues for the duration of the proof effort. Access to this automated instantiation infrastructure is made available to the user through the computed hint macro: `(quant::inst?)`[1].

[1]Inspired by and named for a PVS[6] proof command

5. SKOLEMIZATION

As mentioned above, our implementation of skolemization is nothing more than the generalization of quantification function symbols. While it is not possible to extend ACL2's built-in generalization facility, ACL2 does support the construction of verified clause processors which can be used to perform arbitrary generalizations. We have developed a clause processor that generalizes expressions of the form `(gensym::generalize ..)`. To induce the generalization of the function symbol `(p a x)` it is sufficient to rewrite it into `(gensym::generalize (p a x))` which the clause processor will then replace with a new symbol.

Support for skolemization in our framework involves three components. First, the construction of appropriate skolemization rewrite rules to wrap quantification witnesses with `(gensym::generalize ..)`. Such rules are constructed automatically by the `def::un-sk` macro. Next, a clause processor that recognizes generalization instances in the goal and replaces them with unique symbols. Finally, a computed hint capable of detecting existentially quantified formulae instances in a goal and enabling the skolemization rules for those specific instances.

It is possible for skolemization to prepare the way for induction by replacing a function call with a symbol. Usually, however, skolemization is useful only as a means of reducing proof clutter by eliminating irrelevant details. Access to automated skolemization is provided to the user through the use of the computed hint macro: `(quant::skosimp)`[2].

6. APPLICATION

The theorem `forall-p-append`[3] can be proven automatically using our infrastructure assuming `forall-p` is introduced using `def::un-sk`. The complete automated proof of the theorem contains 15 Subgoals, 15 skolemizations (not required but retained to demonstrate the feature), and 20 instantiations. The abridged output from one particularly illustrative subgoal is presented here. The proof itself begins as follows, with hints suggesting skolemization and instantiation.

```
(defthm forall-p-append
   (equal (forall-p (append x1 x2))
          (and (forall-p x1) (forall-p x2)))
  :hints ((quant::skosimp) (quant::inst?)))
```

The `(quant::skosimp)` hint fires first, finds a skolemizable formula in the goal and reports this to the user. Note that the skolemizable formula is actually a universally quantified formula that appears in the conclusion (in effect, negated). This is indicated in the comment window by square brackets around the name of the formula. A hint is constructed that enables the skolemization rule that will rewrite the specific instance `(forall-p-witness X2)` into `(gensym::generalize (hide (forall-p-witness X2)))`, where the `hide` term is included to inhibit recursive applications of the rule.

[2]Also named in honor of a PVS proof command

[3]See the ACL2 documentation topic QUANTIFIERS-USING-DEFUN-SK-EXTENDED

```
Subgoal 10
(IMPLIES
 (AND
  (MEMBER (FORALL-P-WITNESS (APPEND X1 X2)) X2)
  (P (FORALL-P-WITNESS (APPEND X1 X2)))
  (MEMBER (FORALL-P-WITNESS X2) X2))
 (P (FORALL-P-WITNESS X2))).

Skolemizable Formula In Goal:
[FORALL-P]: (EXISTS (A)
 (NOT (IMPLIES (MEMBER A X2) (P A))))

Computed Hint:
  (:DO-NOT '(PREPROCESS)
   :IN-THEORY (ENABLE FORALL-P-SKOLEMIZATION)
   :RESTRICT ((FORALL-P-SKOLEMIZATION ((X X2)))))
```

The skolemization rewrite rule fires and the appearance of `(gensym::generalize ..)` in the goal triggers the invocation of the generalization clause processor which replaces `(gensym::generalize ..)` with the new symbol `HIDE10`.

```
This simplifies, using the :rewrite rule
FORALL-P-SKOLEMIZATION, to

Subgoal 10'
(IMPLIES
 (AND
  (MEMBER (FORALL-P-WITNESS (APPEND X1 X2)) X2)
  (P (FORALL-P-WITNESS (APPEND X1 X2)))
  (MEMBER (GENSYM::GENERALIZE
            (HIDE (FORALL-P-WITNESS X2))) X2))
  (P (GENSYM::GENERALIZE
       (HIDE (FORALL-P-WITNESS X2))))).

We now apply the verified :CLAUSE-PROCESSOR
function GENERALIZE-CLAUSE-PROCESSOR-WRAPPER to
produce one new subgoal.

Subgoal 10''
(IMPLIES
 (AND
  (MEMBER (FORALL-P-WITNESS (APPEND X1 X2)) X2)
  (P (FORALL-P-WITNESS (APPEND X1 X2)))
  (MEMBER HIDE10 X2))
 (P HIDE10)).
```

The `(quant::inst?)` hint now detects the presence of an instantiable formula in the goal and reports it to the user. An instance of this formula is also found and a `:use` hint is generated. Note, however, that the goal does not actually contain the expression instance `(member HIDE10 (binary--append X1 X2))` as implied by the instantiation. Rather, it contains the more specific `(member HIDE10 X2)` which was generated by ACL2 from that expression via the application of the rewrite rule `member-append`. Our ability to detect such instances requires a tight interaction between pattern matching and simplification by rewriting.

```
Instantiable Formula In Goal:
FORALL-P : (FORALL (A)
 (IMPLIES (MEMBER A (BINARY-APPEND X1 X2)) (P A)))

Computed Hint:
(:USE (:INSTANCE FORALL-P-NECC
        (A HIDE10) (X (BINARY-APPEND X1 X2))))
```

Simplification of the instantiated formula is now sufficient to prove the goal.

```
We augment the goal with the hypothesis provided
by the :USE hint.  The hypothesis can be derived
from FORALL-P-NECC via instantiation. We are
left with the following subgoal.

Subgoal 10'''
(IMPLIES
 (AND (IMPLIES
   (FORALL-P (APPEND X1 X2))
   (IMPLIES (MEMBER HIDE10 (APPEND X1 X2))
            (P HIDE10)))
  (MEMBER (FORALL-P-WITNESS (APPEND X1 X2)) X2)
  (P (FORALL-P-WITNESS (APPEND X1 X2)))
  (MEMBER HIDE10 X2))
(P HIDE10)).

But simplification reduces this to T using the
:rewrite rules MEMBER-APPEND ...
```

7. CONCLUSION

While ACL2 does not provide automated support for reasoning about quantified formulae, it does provide mechanisms that enable suitably motivated users to add such functionality in a sound fashion. An ACL2 framework based on tables, computed hints and clause processing has been described that identifies quantified formulae and skolemizes or instantiates them automatically when possible. This framework has been employed to prove automatically the `forall-p-append` example presented in the ACL2 documentation.

8. REFERENCES

[1] Jared Davis. Finite set theory based on fully ordered lists. In *Fifth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 '04)*, November 2004.

[2] Matt Kaufmann. Bash facility. ACL2 System Book /misc/bash.lisp, 2006.

[3] J Moore. Essay on the design of consider hints. ACL2 System Book /hints/consider-hints.lisp, 2007.

[4] S. Ray. Quantification in Tail-recursive Function Definitions. In P. Manolios and M. Wilding, editors, *Proceedings of the* 6*th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2006)*, volume 205 of *ACM International Conference Series*, pages 95–98, Seattle, WA, August 2006. ACM.

[5] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.

[6] N. Shankar, S. Owre, and J. M. Rushby. *The PVS Proof Checker: A Reference Manual (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.