Using Quantification in ACL2

Nathan Wetzler
nwetzler@cs.utexas.edu

University of Texas, Austin
March 25, 2012

Introduction

- ACL2 is described as a "quantifier-free" first-order logic of recursive functions

- David Greve: "[quantification in ACL2 is a] second-class citizen in a first-order world."

- ACL2 does provide a construct that mimics quantification, but automated reasoning is not supported.

Outline

- Quantification (Preliminaries)
- Quantification in ACL2
- Automated Reasoning for Quantification in ACL2

Quantification in Logic

- Quantifiers help distinguish first-order logic from propositional logic
- Quantification occurs over a “domain of discourse” or “universe”
- Universal quantification
  - Traditional notation: $\forall\ x \in D\ P(x)$
    - Variants: $\forall x\ P(x)$ (forall x : (P x))
  - For all elements x in domain D, P is true of x
- Existential quantification
  - Traditional notation: $\exists\ x \in D\ P(x)$
    - Variants: $\exists x\ P(x)$ (exists x : (P x))
  - There exists an element x in domain D such that P is true of x.

Proof Strategies

- Universal (forall) as hypothesis
- Universal (forall) as conclusion
- Existential (exists) as hypothesis
- Existential (exists) as conclusion

Proof Strategies

- Universal (forall) as hypothesis

```
Suppose we want to prove:
(implies (forall x (P x))
         (Q y))

Then we can choose some object "a" and add (P a) to our
hypotheses.
(implies (and (forall x (P x))
              (P a))
         (Q y))
```

Proof Strategies

- Universal (forall) as conclusion

```
Suppose we want to prove:
(implies (Q y)
         (forall x (P x)))

Then we must prove (P a) for an arbitrary "a".
(implies (Q y)
         (P a))
```

Proof Strategies

- Existential (exists) as hypothesis

```
Suppose we want to prove:
(implies (exists x (P x))
         (Q y))

Then we can add (P a) for an arbitrary "a" to our
hypotheses.
(implies (and (exists x (P x))
              (P a))
         (Q y))
```

Proof Strategies

- Existential (exists) as conclusion

```
Suppose we want to prove:
(implies (Q y)
         (exists x (P x)))

Then must choose some object "a" and prove:
(implies (Q y)
         (P a))
```

Definition:

```
(subset x y) =
(forall e : (member e x)
            --> (member e y))
```

Prove:

```
(subset x y)
& (subset y z)
--> (subset x z)
```

<--> definition of subset

```
(subset x y)
& (subset y z)
--> (forall e : (member e x)
                --> (member e z))
```

forall conclusion, e is not free

```
(subset x y)
& (subset y z)
--> ((member e x) --> (member e z))
```

<--> promote

```
(subset x y)
& (subset y z)
& (member e x)
--> (member e z)
```

<--> definition of subset

```
(forall e : (member e x)
            --> (member e y))
& (forall e : (member e y)
              --> (member e z))
& (member e x)
--> (member e z)
```

forall hypothesis twice, e/e

```
(member e x) --> (member e y)
& (member e y) --> (member e z)
& (member e x)
--> (member e z)
```

<--> forward chaining twice, hypothesis

```
true
```

Why Use Quantifiers in ACL2?

Pros:

- Sometimes we can avoid writing a complicated witnessing function
- Makes a cleaner specification that resembles classical logic
- Can help modularize proof by hiding witnessing function

Cons:

- Limited reasoning support
- May still have to write witnessing function
- Usually do the same thing with recursion
- Non-executability

Quantification in ACL2

- Syntax of ACL2 does not allow the use of quantifiers
- Quantification in ACL2 can be achieved through the construct `defun-sk`
- Syntax of `defun-sk`

```
(defun-sk function-name (formal-parameters)
  (quantifier (quantified-variables) body))
```

  - `quantifier` must be either `forall` or `exists`
  - All variables in `body` must be either formal parameters or quantified variables (no free variables).
  - A nice naming convention is to use the prefix `forall-` or `exists-`

Example

```
Logic definition:
(subset x y) =
(forall e : (member e x)
            --> (member e y))

Logic theorem:
(subset x y)
& (subset y z)
--> (subset x z)
```

```
ACL2 defun-sk:
(defun-sk forall-subset (x y)
  (forall e (implies (member e x)
                     (member e y))))

ACL2 theorem:
(defthm forall-subset-transitive
  (implies (and (forall-subset x y)
                (forall-subset y z))
           (forall-subset x z)))
```

`defun-sk` expansion

- `defun-sk` is implemented as a macro
- This macro translates to an encapsulate that does three* things:
  - `defchoose` event to establish a witness function
  - `defun` event to establish predicate
  - `defthm` event to establish quantification theorem

defun-sk expansion

```
(defun-sk forall-subset (x y)
  (forall e (implies (member e x)
                     (member e y))))

Translates to:

(encapsulate
 ((forall-subset-witness (x y) e))
 (local (in-theory '(implies)))
 (local
  (defchoose forall-subset-witness (e) (x y)
    (not (implies (member e x) (member e y)))))
 (defun-nx forall-subset (x y)
   (declare (xargs :non-executable t))
   (let ((e (forall-subset-witness x y)))
        (implies (member e x) (member e y))))
 (in-theory (disable (forall-subset)))
 (defthm forall-subset-necc
   (implies (not (implies (member e x) (member e y)))
            (not (forall-subset x y)))
   :hints (("goal" :use (forall-subset-witness forall-subset)
            :in-theory (theory 'minimal-theory)))))
```

Quantification Predicate

- Second event in defun-sk macro is a definition:

```
(defun-nx function-name (formal-parameters)
  (let ((quantification-variables
          (witness-function formal-parameters)))
    body))
```

- Best way to think about the occurrence of this function in a proof is that it represents the quantified formula.

- The defun-nx is simply a non-executable defun

Quantification Theorem

- Third event in defun-sk macro is a theorem, referred to as the "quantification theorem":

```
(defthm function-name-suff   ;existential
  (implies body
           (function-name formal-parameters)))

(defthm function-name-necc   ;universal
  (implies (not body)
           (not (function-name formal-parameters))))
```

- The best way to think about this theorem is that it can be used to supply a witness in a proof.

Quantifier Proof in ACL2

```
(defun-sk forall-subset (x y)
  (forall e (implies (member e x)
                     (member e y))))

(defthm forall-subset-transitive
  (implies (and (forall-subset x y)
                (forall-subset y z))
           (forall-subset x z))
  :hints (("Goal"
           :use ((:instance (:definition forall-subset)
                            (x x)
                            (y z))
                 (:instance forall-subset-necc
                            (x x)
                            (y y)
                            (e (forall-subset-witness x z)))
                 (:instance forall-subset-necc
                            (x y)
                            (y z)
                            (e (forall-subset-witness x z)))))))
```

Quantification Versus Recursion

- Sometimes quantification may not be necessary:

```
(defun-sk forall-subset (x y)
  (forall e (implies (member e x)
                     (member e y))))

(defun subset-recursive (x y)
  (if (atom x)
      t
      (if (member (car x) y)
          (subset-recursive (cdr x) y)
        nil)))

(defthm subset-equal
  (equal (forall-subset x y)
         (subset-recursive x y)))
```

Why Use Quantifiers in ACL2?

Pros:

- Sometimes we can avoid writing a complicated witnessing function
- Makes a cleaner specification that resembles classical logic
- Can help modularize proof by hiding witnessing function

Cons:

- Limited reasoning support
- May still have to write witnessing function
- Usually do the same thing with recursion
- Non-executability

Automation

- David Greve worked on improving quantification reasoning in ACL2
- Paper: “Automated reasoning with quantified formulae” (2009)
- Work is distributed in the ACL2 books repository: “books/coi/quantification/quantification.lisp”

Motivation

- Greve was familiar with two tools from PVS called "`skosimp`" and "`inst?`"
- "`skosimp`" would identify quantified formulae and skolemize them (remove the quantifier and replace the quantified variable with a free variable)
- "`inst?`" would identify quantified formulae and attempt to instantiate them.

Usage

- Include the quantification book by adding:

```
(include-book “coi/quantification/quantification” :dir :system)
```

- Replace `defun-sk` with `def::un-sk`. Same syntax.
- Two computed hints: `(quant::skosimp)` and `(quant::inst?)`
  - Apply hints to theorems by adding:

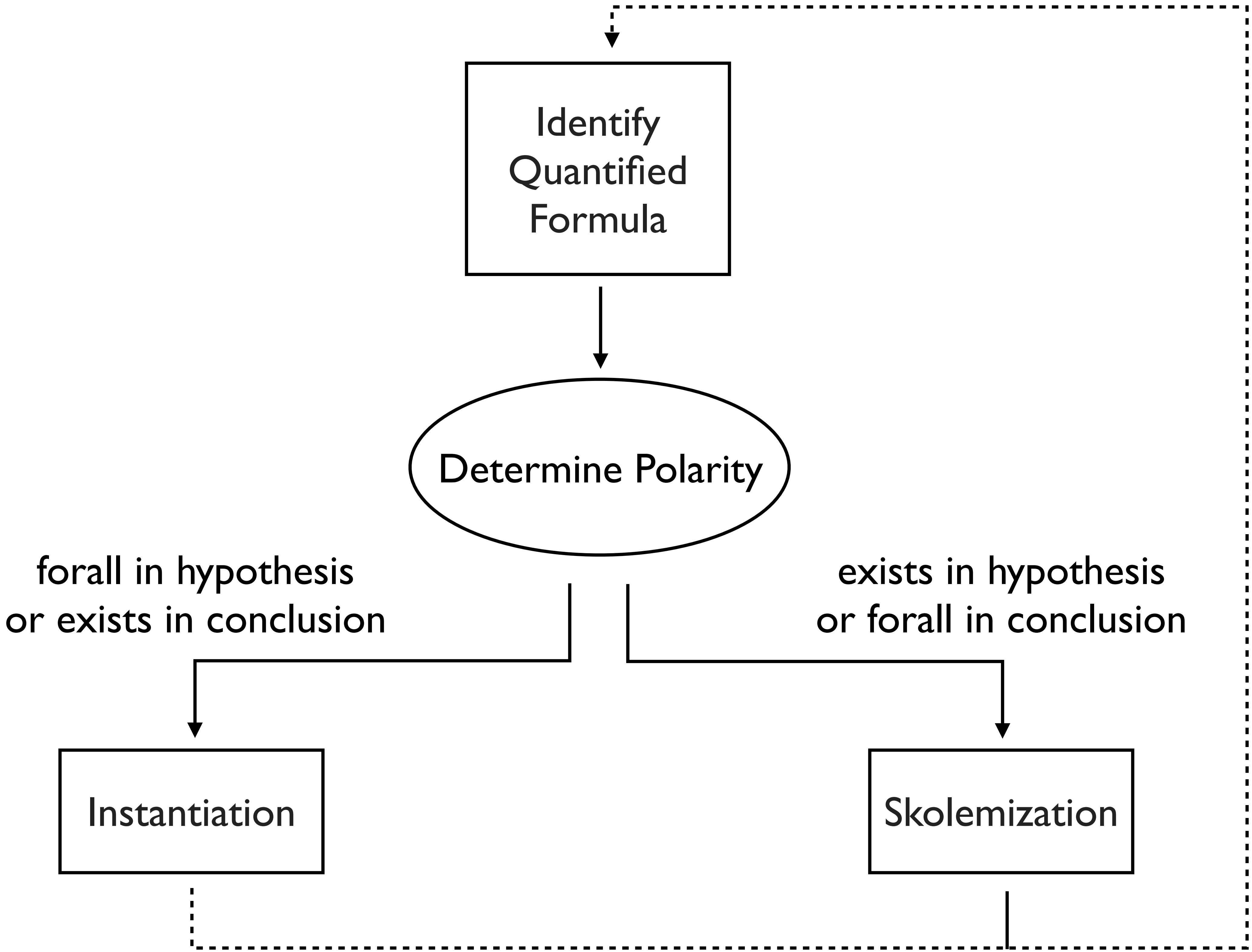
```
:hints ((quant::skosimp) (quant::inst?))
```

Quantification Proof

```
(include-book "coi/quantification/quantification" :dir :system)


(def::un-sk forall-subset (x y)
  (forall e (implies (member e x)
                     (member e y))))


(defthm forall-subset-transitive
  (implies (and (forall-subset x y)
                (forall-subset y z))
           (forall-subset x z))
  :hints ((quant::skosimp) (quant::inst?)))
```


Identify Quantified Formula
Determine Polarity
forall in hypothesis or exists in conclusion
exists in hypothesis or forall in conclusion
Instantiation
Skolemization

Identification

- Some of the information about quantified formulae is not available at proof time.
- To solve this, Greve defined def::un-sk which is a wrapper for defun-sk but also creates an ACL2 table with all the necessary information
  - Includes quantifier type, quantified variables, formal variables, lemma names, witness name, body, etc.
- With a stored list of all quantified formulae that might appear, we can search the goal for instances for the quantified formulae (which will appear as the witness function).

Instantiation

- After identifying a quantified formula that needs instantiation, we must search for subterms of the quantified formula in the goal
- If a match is found (that binds the formal parameters and quantified variables), then the quantification theorem is called with the appropriate binding
- Instantiations are done one at a time so that the prover is not overwhelmed

Skolemization

- Once we identify a quantified formula that needs skolemization, we need to generalize by creating a new variable representing the quantified formula.
- First, the witness term is flagged for generalization by wrapping it in `(gensym::generalize ...)`
- Second, a clause processor recognizes instances of the wrapper and replaces them with a new symbol.

Why Use Greve's Work?

Pros:

- Works very nicely on simple examples
- Very good with automatic instantiation when instance can be pattern-matched

Cons:

- Performs poorly with nested quantifiers
- Does not work when pattern matching is not possible
- Potential problem when the order of simplification matters

Evolution of Proofs

- Let's take a quick look again at the evolution of our subset proof

```
Definition:
(subset x y) =
(forall e : (member e x)
            --> (member e y))


Prove:
(subset x y)
& (subset y z)
--> (subset x z)

<--> definition of subset

(subset x y)
& (subset y z)
--> (forall e : (member e x)
                --> (member e z))

forall conclusion, e is not free

(subset x y)
& (subset y z)
--> ((member e x) --> (member e z))

<--> promote
```

```
(subset x y)
& (subset y z)
& (member e x)
--> (member e z)

<--> definition of subset

(forall e : (member e x)
            --> (member e y))
& (forall e : (member e y)
              --> (member e z))
& (member e x)
--> (member e z)

forall hypothesis twice, e/e

(member e x) --> (member e y)
& (member e y) --> (member e z)
& (member e x)
--> (member e z)

<--> forward chaining twice, hypothesis

true
```

```
(defun-sk forall-subset (x y)
  (forall e (implies (member e x)
                     (member e y))))

(defthm forall-subset-transitive
  (implies (and (forall-subset x y)
                (forall-subset y z))
           (forall-subset x z))
  :hints (("Goal"
           :use ((:instance (:definition forall-subset)
                            (x x)
                            (y z))
                 (:instance forall-subset-necc
                            (x x)
                            (y y)
                            (e (forall-subset-witness x z)))
                 (:instance forall-subset-necc
                            (x y)
                            (y z)
                            (e (forall-subset-witness x z)))))))
```

```
(include-book "coi/quantification/quantification" :dir :system)

(def::un-sk forall-subset (x y)
  (forall e (implies (member e x)
                     (member e y))))

(defthm forall-subset-transitive
  (implies (and (forall-subset x y)
                (forall-subset y z))
           (forall-subset x z))
  :hints ((quant::skosimp) (quant::inst?)))
```

Conclusion

- Quantification is possible in ACL2 through the construct `defun-sk`
- Automated reasoning about quantified formulae is not supported
- David Greve has contributed a library that helps automate quantification reasoning

Appendix

defchoose

- Syntax:

```
(defchoose fn (bound-vars) (free-vars)
  body)
```

- Simplest way to think about `defchoose` is that it produces a witnessing function generated by ACL2.
- A more (but not entirely) correct view is that `defchoose` acts like an encapsulate that exports the function name and has the following theorem/axiom:

```
(implies body
         (let ((bound-vars (fn free-vars)))
           body))
```

- With respect to `defun-sk`, universal quantification results in a negation of the body of the `defun-sk`.
- Also a `:strengthen` argument, but that's beyond the scope of this talk. (adds extra axioms about finding a canonical element)

Quantification Theorem

- Third event in defun-sk macro is a theorem, referred to as the "quantification theorem":

```
(defthm function-name-suff   ;existential
  (implies body
           (function-name formal-parameters)))

(defthm function-name-necc   ;universal
  (implies (not body)
           (not (function-name formal-parameters))))
```

- The best way to think about this theorem is that it can be used to supply a witness in a proof.

- Note the difference between the existential and universal forms. The universal form is somewhat hard to think about as is. Think about the contrapositive instead.

- The universal version isn't a great rewrite rule (because of the not in the conclusion). If you supply the option :rewrite :direct to defun-sk, then the contrapositive will be used instead:

```
(defthm function-name-necc ;universal with :rewrite :direct
  (implies (function-name formal-parameters)
           body))
```