Quantification in Tail-recursive Function Definitions

Sandip Ray
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712. USA
sandip@cs.utexas.edu

ABSTRACT

We investigate the logical issues behind axiomatizing equations that contain both recursive calls and quantifiers in ACL2. We identify a class of such equations, named *extended tail-recursive equations*, that can be uniformly introduced in the logic. We point out some potential benefits of this axiomatization, and discuss the logical impediments behind introducing more general quantified formulas.

Categories and Subject Descriptors

F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Assertions, Invariants, Mechanical Verification*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Computational Logic, Mechanical Theorem Proving*

General Terms

Theory, Verification

Keywords

Formal methods, Logic, ACL2, Skolemization, Conservativity

1. INTRODUCTION

In this paper, we explore ways to introduce "quantified recursive predicates" in ACL2. In particular, consider introducing the predicate `true` with the following equation:[1]

```
(= (true x)
   (if (done x) T (forall i (true (st x i)))))
```

Here we assume that the function symbols `done` and `st` have been introduced in the current ACL2 theory. Function symbols are introduced using three extension principles, namely definition, encapsulation, and `defchoose`. Since the recursive call of `true` is enclosed within a quantifier, the extension principles cannot be directly used to introduce `true` axiomatized to satisfy the above equation. However, the equation, if introduced as an axiom, preserves the consistency of the resulting theory. In fact, `true` can be introduced via encapsulation by exhibiting a witness as follows:

```
(encapsulate (((true *) => *))
  (local (defun true (x) T))
  (defthm true-satisfies-defining-equation
    (= (true x)
       (if (done x) T (forall i (true (st x i)))))))
```

It is often useful to introduce axioms containing both recursive calls and quantifiers. For example, the natural definition of the semantics of LTL constitutes a quantified predicate that recurs down a path through a Kripke Structure [8]; therefore, this definition cannot be introduced in ACL2. While the restriction can sometimes be circumvented by alternative definitions — in the case of LTL by defining its semantics with eventually periodic paths — such alternatives are typically more complicated to reason about [10].

In this paper, we explore the logical issues behind introducing axioms containing both recursion and quantification in ACL2. We identify a class of equations, named *extended tail-recursive equations*, which involve both recursive calls and quantifiers but which can nevertheless can be axiomatized in the logic. The equations form a natural extension of tail-recursive equations. We discuss some of the potential practical consequences of this observation, and the logical limitations behind further extending the class.

2. QUANTIFICATION IN ACL2

The syntax of ACL2 is quantifier-free. However, ACL2 has a construct called `defchoose` for introducing quantified predicates. We start with a brief overview of `defchoose`. For a more thorough treatment, the reader is referred to the topics `defchoose` and `defun-sk` in the ACL2 user's manual [4].

Assume that `foo` is a binary function and we wish to define a predicate `E-foo` so that `(E-foo x)` holds if and only if there exists some `y` such that `(foo x y)` holds. To do so, we first introduce the function `E-foo-witness` as follows:

```
(defchoose E-foo-witness (y) (x) (foo x y))
```

The effect of the above form is to extend the current theory with the following axiom.

[1]The syntax of ACL2 allows quantified formulas only via Skolemization. In this paper we will often write formulas with explicit quantification for pedagogical reasons.

```
(implies (foo x y) (foo x (E-foo-witness x)))
```

Thus, if there exists some `y` such that `(foo x y)` holds, then `(E-foo-witness x)` returns such a `y`; the return value is unspecified if no such `y` exists. We can now define `E-foo`.

```
(defun E-foo (x) (foo x (E-foo-witness x)))
```

`(E-foo x)` holds if and only if there is some `y` such that `(foo x y)` holds, as desired, and `E-foo-witness` can be viewed as a *Skolem function* supplying a witness `y` when `(E-foo x)` holds. Universal quantifications can be introduced by reducing them to existential quantifications. Thus, a predicate `F-foo` such that `(F-foo x)` holds if and only if `(foo x y)` holds for all `y`, can be introduced as follows:

```
(defchoose F-foo-witness (y) (x) (not (foo x y)))
(defun F-foo (x) (foo x (F-foo-witness x)))
```

ACL2 provides a macro called `defun-sk` to conveniently introduce quantified predicates: the following two forms introduce `E-foo` and `F-foo` by expanding to the events above.

```
(defun-sk E-foo (x) (exists y (foo x y)))
(defun-sk F-foo (x) (forall y (foo x y)))
```

The `defchoose` construct essentially provides a way of introducing first-order quantified predicates in ACL2 by Skolemization. The ability to specify arbitrary quantified predicates is a powerful feature of the logic; recent research has made use of quantification to formalize many generic functions proof strategies and investigate relations between them [6, 9, 11]. However, there are restrictions on the form of quantified predicates that can be introduced directly as above. For instance, `defchoose` can only introduce *non-recursive* quantified formulas: to extend an ACL2 theory with `E-foo` or `F-foo`, the function `foo` must have been *already* introduced in the current theory. This disallows the definition of `true` we saw in Section 1. Furthermore, ACL2 disallows the use of `defchoose` in a mutual recursion clique.

3. RECURSION AND QUANTIFICATION

Although the `defchoose` construct cannot introduce axioms involving both recursion and quantification, we saw in Section 1 that such they can be introduced by encapsulation if we can exhibit an appropriate witness. The witness for the predicate `true` is the constant function that always returns `T`. In this section, we show how to uniformly define witnesses for more general equations by encapsulation.

Consider the predicate `F-iv1` with the following axiom:

```
(= (F-iv1 x)
   (if (done x) (base x) (forall i (F-iv1 (st1 x i)))))
```

Here `done`, `base`, and `st1` are functions in the current theory. The axiom then can be introduced by encapsulation with the following witness.

```
(defun sn1 (x ch)
 (if (endp ch) x (sn1 (st1 x (car ch)) (cdr ch))))
(defun n-done (x ch)
 (if (endp ch) (not (done x))
   (and (not (done x))
        (n-done (st1 x (car ch)) (cdr ch)))))
(defun rm-lst (ch)
  (if (endp ch) nil
    (if (endp (cdr ch)) nil
      (cons (car ch) (rm-lst (cdr ch))))))
(defun done-ch1 (x ch)
 (and (done (f-sn1 x ch))
      (implies (consp ch) (n-done x (rm-lst ch)))))
(defun-sk F-iv1 (x)
 (forall ch
   (implies (done-ch1 x ch) (base (f-sn1 x ch)))))
```

We now explain the intuition behind the definition of this witness. Given an `i`, call `(st x i)` a *successor* of `x` selected by `i`. We think of `i` as a non-deterministic selector that chooses a successor for `x`, and `st` as transforming `x` to its successor given the choice `i`. Our desired axiom thus postulates an invariant over this transformation: if `x` satisfies `done` then the invariant holds if and only if `base` holds; otherwise it holds for `x` if and only if it holds for each successor. Introducing `F-iv1` amounts to exhibiting such an invariant.

With this view, `sn1` is an iterated transformation function for `x` based on a *selector sequence* `ch`. The definition of the witness `F-iv1` thus says that for all sequences `ch`, the first descendant of `x` that satisfies `done` must also satisfy `base`. If this condition holds for some `x` which does not satisfy `done` then it also holds each successor of `x` and vice-versa, justifying our equation. The key observation is that the (universal) quantification in the equation can be transferred to a (universal) quantification over the sequence `ch`.

We now explore some variations of the equation above. Consider defining a predicate `E-iv1` as follows:

```
(= (E-iv1 x)
   (if (done x) (base x) (exists i (E-iv1 (st x i)))))
```

`E-iv1` can be introduced using the same approach as `F-iv1`. The corresponding witness, defined below assuming appropriate definitions of the auxiliary functions as above, now posits that *there exists* a selector sequence such that the first `done` descendant of `x` satisfies `base`. Thus the desired quantification is transferred to the selector choice.

```
(defun-sk E-iv1 (x)
  (exists ch (and (done-ch1 x ch) (base (f-sn1 x ch)))))
```

It is instructive to consider the relationship between the above equations tail-recursive ones. Tail-recursive equations have the following general form (where `iv0` is the new function symbol being introduced):

```
(= (iv0 x) (if (done x) (base x) (iv0 (st0 x))))
```

Manolios and Moore [6] show how to introduce `iv0` by encapsulation. We can view this axiom as a special case of invariant over transformations, where the transformation `st0` takes `x` to a *unique* successor. Thus, a witness for the equation is the formula that says that if a descendant of `x` satisfies `done` then the first such descendant must satisfy `base`; this is essentially the witness constructed by Manolios and Moore.

We now briefly discuss alternating quantifiers. Consider introducing the predicate `EF-iv2` with the axiom below:

```
(= (EF-iv2 x)
   (if (done x) (base x)
     (exists i (forall j (EF-iv2 (st2 x i j))))))
```

Viewing `st2` as a transformation function on `x` — this time with two selectors `i`, and `j` — the equation can be satisfied by the predicate that says that for each sequence of `i`-choices there exists a sequence of `j`-choices such that the first descendant of `x` that satisfies `done` also satisfies `base`. As above, since the quantification is necessary over the selector sequences, we can introduce `EF-iv2` with definitions analogous to `E-iv1` and `F-iv1`.

Generalizing our observations so far, we now characterize the class of *extended tail-recursive* equations. The equation introducing a predicate `Q-iv` *extended tail-recursive* if it satisfies the following conditions:

1. There is exactly one recursive branch.
2. The outermost function symbol in the recursive branch is `Q-iv`, possibly enclosed within a sequence of quantifiers.

It is easy to see that any extended tail-recursive equation can be introduced in ACL2 by introducing the corresponding selector sequences to be quantified over.

4. LOGICAL IMPEDIMENTS

The conditions for extended tail-recursive equations might appear unduly restrictive. In particular, note that the first condition restricts the number of recursive branches in the defining axiom to be exactly one. We now discuss logical reasons behind these restrictions.

ACL2 restricts the use of quantification in order to prevent the extended theory from violating *conservativity*. Roughly, conservativity implies that if `foo` is a function symbol introduced to extend an ACL2 theory τ, then every formula ϕ that does not involve the symbol `foo` is provable in the extended theory if and only if there exists a (first-order) proof of ϕ in τ. Conservativity is maintained by each extension principle [5] and is key to the proof of consistency of ACL2 theories: since the formula `nil` does not involve the introduced function symbol, `nil` is provable after extension if and only if it is provable before, reducing the consistency of the extended theory to that of the initial "ground-zero" theory.

We provide a sketch of an argument showing that recursion and quantification in concert can violate conservativity.[2] From Gödel's Incompleteness Theorem [2], it is known that a truth predicate over Peano Arithmetic formulas is not a conservative extension of Peano Arithmetic. However, with the ability to axiomatize arbitrary quantified predicates we can define such a predicate as follows. First, we define a function `(prenex phi)` that turns a quantified formula `phi` over Peano functions to the prenex form; this function operates purely on the syntactic structure of the formula and can be easily defined in ACL2 as a recursive function. Consider then the following "definition" of `true-formula` below, which would be admissible if arbitrary quantified recursive definitions were allowed.

```
(defun true-formula-aux (phi sigma)
 (cond
  ((exists-p phi) ;; formula is (E x phi')
   (exists val
    (true-formula-aux (qbody phi)
                      (acons (qvar phi) val sigma))))
  ((forall-p phi) ;; formula is (A x phi')
   (forall val
    (true-formula-aux (qbody phi)
                      (acons (qvar phi) val sigma))))
  (t ;; formula is quantifier-free
   (term-value phi sigma))))
(defun true-formula (phi sigma)
  (true-formula-aux (prenex phi) sigma))
```

Here `sigma` is an association list binding the free variables of `phi` to values. The function `term-value` defines an evaluator of quantifier-free terms composed of the function symbols in `phi`; such an evaluator is definable for a previously fixed collection of function symbols. We can now prove by induction that `true-formula` holds for every formula that is provable. Therefore, the formulation of this definition in Peano Arithmetic is not conservative. Finally, since any ACL2 theory can be viewed as a first order theory formed by a conservative extension of Peano arithmetic together with ϵ_0-induction, we conclude that the definition is not conservative with respect to ACL2 theories.

The axiom for `true-formula-aux` is tail-recursive (apart from quantifiers) with two recursive branches. Thus restricting the number of recursive branches to one in extended tail-recursive equations is critical. Note however, that in the trivial case where `base` is a constant, as for `true` in Section 1, we can have an arbitrary number of quantified tail-recursive branches. Furthermore, an arbitrary number of branches is possible if there is no quantifier (as in case of `iv0`) since they can be reduced to one by if-lifting.

5. PRACTICAL BENEFITS

In spite of the restrictive nature of extended tail-recursive equations, they are useful in some interesting cases. A trivial but entertaining consequence of their admissibility is the possibility of using the *inductive assertions method* [1] to reason about non-deterministic computing systems. In this method, the user annotates a program by attaching assertions on certain *cutpoints*, and the goal is to prove that whenever program control reaches a cutpoint, the corresponding assertions hold. Moore [7] shows how to use symbolic simulation to derive such proofs from an operational model of the system. An operational model is given by a function `next` that can be treated as a state transformation function: `(next s)` gives the state of the machine after one transition from `s`. Moore's method involves the definition of a predicate `inv0` with the following equation:

```
(= (inv0 s) (if (cut s) (assert s) (inv0 (next s))))
```

Here `cut` is a predicate recognizing the cutpoints. Attempting to prove the formula `(implies (inv0 s) (inv0 (next`

[2]The argument has been adapted by the author from an example provided by Kaufmann.

`s)))` causes symbolic simulation of the machine from each cutpoint s that satisfies `assert` until the next subsequent cutpoint s' is reached. However, the method could previously be used only for *deterministic systems*, that is, in which the next state of the machine is uniquely determined by the current state. A non-deterministic system can transit from a state `s` to one of a number of possible next states and is modeled in ACL2 by defining `next` as a binary function such that for a state `s` and an external input `i`, `(next s i)` returns the corresponding next state. The analogue of Moore's predicate for non-deterministic systems is the following, which is extended tail-recursive.

```
(= (inv1 s)
   (if (cut s) (assert s) (forall i (inv1 (next s i)))))
```

As with Moore's approach, the proof of `(implies (inv1 s) (inv1 (next s i)))` would cause symbolic simulation from each cutpoint, this time for every possible input sequence. Note that actually configuring the ACL2 simplifier to perform the symbolic simulation might be non-trivial, because of its limited support in rewriting quantified expressions. Nevertheless, it is gratifying that there is no *logical* limitation in applying inductive assertions to non-deterministic systems. Indeed, exploring the logical issues behind this applicability provided the key motivation for this work.

Another potential application involves formalizing programming language metatheories. In a posting to the `acl2-help` mailing list on March 28, 2006, Swords wanted to use recursion and quantification in order to formalize a certain normalization property of simply-typed λ-calculus. A version of this property turned out to be definable with extended tail-recursive equations. Swords feels [private communication] that such equations can be used for formalizing many similar properties, although they might be insufficient in some cases.

6. CONCLUSION

We have identified a class of equations, called extended tail-recursive equations, which involve recursion and quantifiers but can nevertheless be axiomatized in ACL2. We have also discussed some logical impediments to generalizing the class.

It is important to underline the significance of the last remark. Although *some* interesting predicates can be defined with extended tail-recursive equations, a majority of others cannot be. For instance the semantics of LTL we mentioned in Section 1 cannot be represented as an extended tail-recursive equation. We believe it is important to extend the expressive power of ACL2 to facilitate the use of recursion and quantification. Recent research by Gordon *et al* integrating HOL with ACL2 [3] might provide a long-term solution; with this integration it could be possible to axiomatize such formulas in the more expressive logic of HOL, and export their first-order consequences to ACL2. Since the method is currently is under development, it is not clear how complex it would be to reason about such formulas via the integrated environment. In the mean time, we believe that extended tail-recursive equations provides some facility for axiomatizing rich quantified formulas in ACL2.

Finally, it may be possible to use a similar approach to integrate a richer class of formulas with recursion and quantification, by restricting only to well-founded recursions. In this work we did not impose such restriction. Of course, even if the recursive calls are well-founded, one cannot allow arbitrary recursion with quantification in ACL2: note that in the "definition" of `true-formula-aux`, the size of the formula structure decreases in each recursive call. Nevertheless it will be interesting to explore the forms of equations that can be introduced by quantified well-founded recursions.

7. ACKNOWLEDGEMENTS

J Strother Moore provided the initial impetus for this work by challenging the author to extend inductive assertions to work with non-determinstic systems. We also thank Matt Kaufmann for numerous conversations on conservativity in ACL2, and John Matthews for helpful discussions. This material is based upon work supported by DARPA and the National Science Foundation under Grant No. CNS-0429591.

8. REFERENCES

[1] R. Floyd. Assigning Meanings to Programs. In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematcs*, volume XIX, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[2] K. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematic und Physik*, 38:173–198, 1931.

[3] M. J. C. Gordon, W. A. Hunt, Jr., M. Kaufmann, and J. Reynolds. An Embedding of the ACL2 Logic in HOL. In P. Manolios and M. Wilding, editors, *Proceedings of the 6th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2006)*, Seattle, WA, 2006.

[4] M. Kaufmann and J. S. Moore. ACL2 home page. See URL `http://www.cs.utexas.edu/users/moore/acl2`.

[5] M. Kaufmann and J. S. Moore. Structured Theory Development for a Mechanized Logic. *Journal of Automated Reasoning*, 26(2):161–203, 2001.

[6] P. Manolios and J. S. Moore. Partial Functions in ACL2. *Journal of Automated Reasoning*, 31(2):107–127, 2003.

[7] J. S. Moore. Inductive Assertions and Operational Semantics. In D. Geist, editor, *Proceedings of the 12th International Conference on Correct Hardware Design and Verification Methods*, volume 2860 of *LNCS*, pages 289–303. Springer-Verlag, Oct. 2003.

[8] A. Pnueli. Linear and Branching Structures in the Semantics and Logics of Reactive Systems. In W. Brauer, editor, *Proceedings of the 12th International Colloquium on Automata, Languages, and Programming (ICALP 1985)*, volume 194 of *LNCS*, pages 15–32. Springer-Verlag, 1985.

[9] S. Ray and W. A. Hunt, Jr. Deductive Verification of Pipelined Machines Using First-Order Quantification. In R. Alur and D. A. Peled, editors, *Proceedings of the 16th International Conference on Computer-Aided Verification (CAV 2004)*, volume 3114 of *LNCS*, pages 31–43, Boston, MA, July 2004. Springer-Verlag.

[10] S. Ray, J. Matthews, and M. Tuttle. Certifying Compositional Model Checking Algorithms in ACL2. In W. A. Hunt, Jr., M. Kaufmann, and J. S. Moore, editors, *4th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2003)*, Boulder, CO, July 2003.

[11] S. Ray and J. S. Moore. Proof Styles in Operational Semantics. In A. J. Hu and A. K. Martin, editors, *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2004)*, volume 3312 of *LNCS*, pages 67–81, Austin, TX, Nov. 2004. Springer-Verlag.