Adding APPLY to ACL2 (Part 1)

Matt Kaufmann
J Strother Moore

Department of Computer Science
University of Texas at Austin

January, 2016

Motivation

Iterative constructs are common in all programming languages — except ACL2.

$$\sum_{x \in '(1\ 2\ 3)} x^2$$

```
(loop for x in '(1 2 3) sum (sq x))
```

```
(sumlist '(1 2 3) 'sq)
```

Motivation

Iterative constructs are common in all programming languages — except ACL2.

$$\sum_{x \in '(1\ 2\ 3)} x^2$$

```
(loop for x in '(1 2 3) sum (sq x))
```

=

```
(Sumlist '(1 2 3) 'sq)
```

Note that if we had mapping functions we could define a LOOP macro.

But in ACL2...

```
(defun sum-sq (lst)
  (if (endp lst)
      0
      (+ (sq (car lst))
         (sum-sq (cdr lst)))))

(sum-sq '(1 2 3))
```

Now Write These in ACL2

$$\sum_{x \in '(1\ 2\ 3)} x^3$$

$$\sum_{x \in '(1\ 2\ 3)} x^2 + x$$

$$\sum_{x \in '(1\ 2\ 3)} x^2 + 2x + 1$$

Each requires a different ACL2 function,

`sum-sq`,

`sum-cubes`,

`sum-sq+x`,

`sum-yet-another-poly`.

Two Beautiful Things about Iterative Notation

Succinct: Many different computations can be described with the same control structure.

General: Lemmas can be proved about the control structure independent of the particulars.

$$\sum_{x \in (\texttt{append}\ a\ b)} \gamma = \sum_{x \in a} \gamma + \sum_{x \in b} \gamma$$

```
(sum-sq (append a b))
 = (+ (sum-sq a) (sum-sq b))

(sum-cubes (append a b))
 = (+ (sum-cubes a) (sum-cubes b))

(sum-sq+x (append a b))
 = (+ (sum-sq+x a) (sum-sq+x b))

(sum-yet-another-poly (append a b))
 = (+ (sum-yet-another-poly a)
      (sum-yet-another-poly b))
```

Goals

Make it possible to define such functions as:

```
(defun sumlist (lst fn)
  (if (endp lst)
      0
      (+ (apply fn (list (car lst)))
         (sumlist (cdr lst) fn))))
```

to prove and use such lemmas as:

```
(defthm sumlist-append
  (equal (sumlist (append a b) fn)
         (+ (sumlist a fn)
            (sumlist b fn))))
```

and to reason about and execute such terms as

```
(sumlist lst 'sq)
(sumlist lst 'cube)
(sumlist lst '(lambda (x) (+ (* x x) x)))
(sumlist lst '(lambda (x) (+ (* x x) (* 2 x) 1)))
```

Key: Add Apply

```
(defun sumlist (lst fn)
  (if (endp lst)
      nil
      (+ (apply fn (list (car lst)))
         (sumlist (cdr lst) fn))))
```

Caveats

This is a work in progress.

`APPLY` is a CLTL function that we cannot formalize in ACL2's logic.

We formalize `APPLY$`.

We will pronouce `APPLY$` as though it were "*apply*" because "*apply dollar*" is tedious.

We confuse macros and functions: e.g., we might pass `'+` as a function when we should pass `'binary-+`.

Related Work

To see how something similar was done in Nqthm see

"The Addition of Bounded Quantification and Partial Functions to a Computational Logic and Its Theorem Prover," R. S. Boyer and J S. Moore. **Journal of Automated Reasoning**, Kluwer Academic Publishers, **4**(2), 1988, pp. 117-172.

Tech Report version:

http://www.cs.utexas.edu/users/moore/publications/quant.pdf

Naive Second Order Axiom Scheme

```
(apply fn args)
=
(fn (car args)
    (cadr args)
    ...
    (cad...dr args))
```

Thus,

```
(apply '* (list 3 7))
= 21

(apply 'append (list '(1 2 3) '(a b c)))
= (1 2 3 A B C)
```

Naive Second Order Axiom Scheme

```
(apply fn args)
=
(fn (car args)
    (cadr args)
    ...
    (cad...dr args))
```

Thus,

```
(apply 'binary-* (list 3 7))
= 21

(apply 'binary-append (list '(1 2 3) '(a b c)))
= (1 2 3 A B C)
```

A Problem with Apply

```
(defun russell (fn)                    ; benign nonrec def
  (not (apply fn (list fn))))
```

Thus

```
(russell 'russell)
=                                      ; {def russell}
(not (apply 'russell (list 'russell)))
=                                      ; {naive axiom}
(not (russell 'russell))
```

Contradiction!

Taming Apply

It is easy to *define* a restricted `apply$` that is sound:

```
(defun apply$ (fn args)
  (cond
   ((eq fn 'CAR)  (car (car args)))
   ((eq fn 'CDR)  (cdr (car args)))
   ((eq fn 'CONS) (cons (car args) (cadr args)))
   ...
   ))
```

We could so handle any function that does not ancestrally depend on `apply$`.

But we couldn't apply `sumlist`!

$\sum_{y \in \ a} \ (\sum_{e \ \in \ y} \ e^2)$
$=$

```
(sumlist a '(lambda (y) (sumlist y 'sq)))
```

But we couldn't apply `sumlist`!

$$\sum_{y \in\ a} \left(\sum_{e\ \in\ y}\ e^2\right)$$
$=$

```
(sumlist a '(lambda (y) (sumlist y 'sq)))
```

Taming Apply

Our approach to adding `apply$` is to "tame" the naive axiom so that we have the naive axiom only for "tame" applications.

We will classify some functions as *mapping functions* depending on how they use their arguments.

Roughly speaking, we define tameness so that

- functions ancestrally independent of `apply$` are tame, and
- applications of mapping functions are tame if their arguments are suitably tame.

Avoiding (Non-Definitional) Axioms

We have formalized a prototype `apply$`.

Instead of adding axioms about it we force the user to provide *applicability* hypotheses which

> *explicitly allow* `(apply$ '`f `...)` *to be rewritten to* `(`f `...)` *when the arguments are suitably tame*.

Make-Applicable

We have also prototyped an analysis tool for identifying mapping functions based on how a function uses its arguments.

Caveat About Make-Applicable

We suspect our current `make-applicable` is inadequate: it might admit mapping functions that are not in fact applicable.

This does not imperil soundness but might make some theorems vacuously true.

We foresee strengthening `make-applicable` and constructing a hand proof that it is ok. Stay tuned!

But Wait!

There is another major problem with apply that does imperil soundness!

The LOCAL Problem

```
(encapsulate nil
  (local (defun foo (x) (nfix x)))

  (defthm lemma1
    (equal (sumlist '(1 2 3) 'foo) 6)))

(defun foo (x) (+ 1 (nfix x)))

(defthm lemma2
  (equal (sumlist '(1 2 3) 'foo) 9))

(defthm oops nil
 :hints (("Goal" :use (lemma1 lemma2))))
```

The LOCAL Problem

```
(encapsulate nil
  (local (defun foo (x) (nfix x)))

  (defthm lemma1
    (equal (sumlist '(1 2 3) 'foo) 6)))

(defun foo (x) (+ 1 (nfix x)))

(defthm lemma2
  (equal (sumlist '(1 2 3) 'foo) 9))

(defthm oops nil
 :hints (("Goal" :use (lemma1 lemma2))))
```

The LOCAL Problem

```
(encapsulate nil                                ; Pass 2
  (local (defun foo (x) (nfix x)))

  (defthm lemma1
    (equal (sumlist '(1 2 3) 'foo) 6)))

(defun foo (x) (+ 1 (nfix x)))

(defthm lemma2
  (equal (sumlist '(1 2 3) 'foo) 9))

(defthm oops nil
 :hints (("Goal" :use (lemma1 lemma2))))
```

The LOCAL Problem

```
(encapsulate nil
  (local (defun foo (x) (nfix x)))

  (defthm lemma1
    (equal (sumlist '(1 2 3) (bar x)) 6)))

(defun foo (x) (+ 1 (nfix x)))

(defthm lemma2
  (equal (sumlist '(1 2 3) (bar x)) 9))

(defthm oops nil
 :hints (("Goal" :use (lemma1 lemma2))))
```

provided

```
(thm (equal (bar x) 'foo))
```

Challenges

Tame the naive apply axiom in a pragmatically adequate way.

Solve the LOCAL problem.

Show that our solution is not *vacuous* (to be explained later).

Think about execution, guards, attachments, lambda equivalence, . . .

Pragmatic Adequacy

A demonstration of a certified book providing `apply$` and `make-applicable`.

But first, **Upcoming Seminars** and **Future Work**.

Upcoming Seminars

How `apply$` is defined in the apply book.

What we mean by vacuity . . .
and an ACL2-checked construction showing that a wide variety of mapping function schemes are non-vacuous.

The construction shows that we *could* limit `make-applicable` to the schemes in our "pragmatic adequacy" demonstration, thus achieving *soundness* and *non-vacuity*.

But we *hope* to find a construction that admits more schemes.

To facilitate experimentation we have, for now, made `make-applicable` "too loose."

Future Work

Further confirming our "pragmatic adequacy" hypothesis

Proving that our solution is not vacuous

Addressing the executability problem

Simplifying `lambda` expressions

The Demo