Adding APPLY to ACL2 (Part 3)

Matt Kaufmann
J Strother Moore

Department of Computer Science
University of Texas at Austin

January, 2016

Rules about User-Defined f

We've explained `apply$` for lambda-expressions and primitives.

But what about user-defined functions?

If f is a user-defined function,

(`apply$` f `args`)
= (`apply$-nonprim` f `args`),

where apply$-nonprim is undefined (a `defstub`).

How Do We Prove Anything?

So how do you prove

```
(equal (sumlist '(1 2 3) 'sq)
       14)
```

How Do We Prove Anything?

So how do you prove

```
(implies [∀ args : (apply$ 'sq args) = (sq (car args))]
         (equal (sumlist '(1 2 3) 'sq)
                14))
```

How Do We Prove Anything?

So how do you prove

```
(implies [∀ args : (apply$ 'sq args) = (sq (car args))]
         (equal (sumlist '(1 2 3) 'sq)
                14))
```

Note that this solves the `LOCAL` problem because now the theorem mentions the function `sq`.

How Do We Prove Anything?

So how do you prove

```
(implies [∀ args : (apply$ 'sq args) = (sq (car args))]
         (equal (sumlist '(1 2 3) 'sq)
                14))
```

But we can't write ∀ so we use `defun-sk` to introduce a `Applicablep-SQ` to express that quantified hypothesis.

How Do We Prove Anything?

So how do you prove

```
(implies (Applicablep-SQ)
         (equal (sumlist '(1 2 3) 'sq)
                14))
```

But we can't write $\forall$ so we use `defun-sk` to introduce a `Applicablep-SQ` to express that quantified hypothesis.

How Do We Prove Anything?

So how do you prove

```
(implies (Applicablep SQ)
         (equal (sumlist '(1 2 3) 'sq)
                14))
```

But we can't write ∀ so we use `defun-sk` to introduce a `Applicablep-sq` to express that quantified hypothesis.

`(Applicablep SQ)` is just an abbreviation for `(Applicablep-SQ)`.

Background

The (`Applicablep-`f) hypotheses cannot be proved because they concern undefined functions, e.g., `f-classes-nonprim` and `apply$-nonprim`.

Can we produce a model of these undefined functions that makes the hypotheses provable?

For a Mapping Function

```
(make-applicable SUMLIST)
```

$\Longrightarrow$

```
(defun-sk Applicablep-SUMLIST ()
 (forall (x)
  (and (equal (f-classes-nonprim 'SUMLIST) '(NIL :FN))
       (implies (tamep-functionp (cadr x))
                (equal (apply$-nonprim 'SUMLIST x)
                       (sumlist (car x) (cadr x)))))))
```

(Applicablep-SUMLIST)

$\leftrightarrow$ [(f-classes 'SUMLIST) $=$ '(NIL :FN)

$\wedge$ ($\forall$ x :

(tamep-functionp (cadr x))

$\rightarrow$ (apply$ 'SUMLIST x)

$=$(SUMLIST (car x)(cadr x)))]

Immediate Goal

For a given chronology (sequence of user events) define the stubs of the `apply` book,

- f-classes-nonprim
- apply$-nonprim

in a way that makes all the `Applicablep-`f hypotheses of the chronology provably true.

Eventual Goal

Prove (by hand) that we can always model any admissible chronology.

This requires that we precisely describe how to do it.

Remember: We don't actually have to implement this process. We just have to be sure we could and that it would produce a certifiable file!

In this talk we'll focus on a few representative functions.

```
ap                              ; Ordinary
rev                             ; (independent of apply$)
flatten
sq
fact
gcd
```

```
collect                        ; Mapping Fns
sumlist                        ; (having at least one
sumlist-with-params            ; :FN or :EXPR formal)
filter
all
collect-on
collect-tips
apply$2
russell
foldr
collect-from-to
collect*
collect2
```

```
collect-rev                         ; Tame Instances
                                    ; (uses mapping functions
                                    ; but with (QUOTEd)
                                    ; tame args)
```

Examples

```
(defun$ rev (x)                                        ; Ordinary
  (if (consp x)
      (ap (rev (cdr x)) (cons (car x) nil))
      nil))

(defun$ collect (lst fn)                               ; Mapping fn
  (cond ((endp lst) nil)
        (t (cons (apply$ fn (list (car lst)))
                 (collect (cdr lst) fn)))))

(defun$ collect-rev (lst)                              ; Tame Instance
  (collect lst 'REV))
```

More Precise Immediate Goal

Given certified book "chronology":

```
(in-package "ACL2")
(include-book "apply")
...
(defun rev ...)
...
(defun collect ...)
...
(defun collect-rev ...)
...
(defthm ...)
...
```

create and certify books:

- "`apply!`"
- "`chronology!`"
- "`applicablep!`"

where:

"chronology!":

```
(in-package "ACL2")
(include-book "apply")
...
(defun rev ...)
...
(defun collect ...)
...
(defun collect-rev ...)
...
(defthm ...)
...
```

"chronology!":

```
(in-package "ACL2")
(include-book "apply!")
...
(defun rev ...)
...
(defun collect ...)
...
(defun collect-rev ...)
...
(defthm ...)
...
```

"applicablep!":

```
(in-package "ACL2")
(include-book "chronology!")
(defthm applicable-rev-true
 (Applicablep-REV))
...
(defthm applicable-collect-true
 (Applicablep-COLLECT))
...
(defthm applicable-collect-rev-true
 (Applicablep-COLLECT-REV))
```

A Thought Experiment

Suppose $f_1, \ldots f_n$ are the user's functions.

How would we define `apply$`?

A Thought Experiment

Suppose $f_1, \ldots f_n$ are the user's functions.

How would we define `apply$` . . . and `ev$` and `ev$-list` (since they're mutually recursive)?

But we'll focus just on `apply$`.

Since some f_i call `apply$`, we must define `apply$` *before* $f_1, \ldots, f_n$.

A Thought Experiment

```
(defun apply$ (fn args)
  (cond
   ((consp fn)
    (ev$ (caddr fn) (pairlis$ (cadr fn) args)))
   ((apply$-primp fn) (apply$-prim fn args))
   ((eq fn 'f1) (f1 (car args) ... (cad...dr args)))
   ...
   ((eq fn 'fn) (fn (car args) ... (cad...dr args)))
   ((eq fn 'APPLY$)
    (if (tamep-functionp (car args))
        (apply$ (car args) (cadr args))
        nil))
   (t nil)))
```

A Thought Experiment

```
(defun apply$ (fn args)
  (cond
   ((consp fn)
    (ev$ (caddr fn) (pairlis$ (cadr fn) args)))
   ((apply$-primp fn) (apply$-prim fn args))
   ((eq fn 'f1) (f1 (car args) ... (cad...dr args)))
   ...
   ((eq fn 'fn) (fn (car args) ... (cad...dr args)))
   ((eq fn 'APPLY$)
    (if (tamep-functionp (car args))
        (apply$ (car args) (cadr args))
        nil))
   (t nil)))
```

A Thought Experiment

```
(defun apply$ (fn args)
  (cond



    ((eq fn 'f1) (f1 (car args) ... (cad...dr args)))
    ...
    ((eq fn 'fn) (fn (car args) ... (cad...dr args)))
    ((eq fn 'APPLY$)
     (if (tamep-functionp (car args))
         (apply$ (car args) (cadr args))
         nil))
    (t nil)))
```

A Thought Experiment

```
(defun apply$ (fn args)
  (cond



   ...
   ((eq fn 'COLLECT) (collect (car args) (cadr args)))
   ...
   ((eq fn 'APPLY$)
    (if (tamep-functionp (car args))
        (apply$ (car args) (cadr args))
        nil))
   (t nil)))
```

A Thought Experiment

```
(defun apply$ (fn args)
  (cond  ...
   ((eq fn 'COLLECT)
    (collect (car args) (cadr args)))
   ...
   ((eq fn 'APPLY$)
    (if (tamep-functionp (car args))
        (apply$ (car args) (cadr args))
        nil))
   (t nil)))
```

A Thought Experiment

```
(defun apply$ (fn args)
  (cond ...
   ((eq fn 'COLLECT)
    (collect (car args) (cadr args))) ;  Undefined!
   ...
   ((eq fn 'APPLY$)
    (if (tamep-functionp (car args))
        (apply$ (car args) (cadr args))
        nil))
   (t nil)))
```

A Thought Experiment

```
(defun apply$ (fn args)
  (cond ...
   ((eq fn 'COLLECT)
    (collect (car args) (cadr args)))
   ...
   ((eq fn 'APPLY$)
    (if (tamep-functionp (car args))
        (apply$ (car args) (cadr args))
        nil))
   (t nil)))
(defun collect (lst fn)
  (cond ((endp lst) nil)
        (t (cons (apply$ fn (list (car lst)))
                 (collect (cdr lst) fn)))))
```

A Thought Experiment

```
(mutual-recursion
 (defun apply$ ...)
 (defun ev$ ...)
 (defun ev$-list ...)
 (defun collect ...)     ; all user mapping fns
 (defun sumlist ...)
 ...
 (defun foldr ...)
 (defun russell ...)
 ...)
```

Lesson 1: We must find a measure that justifies this clique!

A Thought Experiment

But if `apply!` contains:

```
(mutual-recursion
 (defun apply$ ...)
 ...
 (defun collect ...)
 ...)
```

we can't certify `chronology!` where `chronology` contains:

```
(defun collect (lst fn)  ; Error: Name in use!
  (cond ((endp lst) nil)
        (t (cons (apply$ fn (list (car lst)))
                 (collect (cdr lst) fn)))))
```

A Thought Experiment

Lesson 2: `apply!` should use different names and prove equivalence.

If the user introduces f_i then we'll introduce f_i!.

We call f_i! the *doppleganger* of f_i.

```
(defun collect (lst fn)
  (cond ((endp lst) nil)
        (t (cons (apply$ fn (list (car lst)))
                 (collect (cdr lst) fn)))))
```

A Thought Experiment

Lesson 2: `apply!` should use different names and prove equivalence.

If the user introduces f_i then we'll introduce f_i!.

We call f_i! the *doppleganger* of f_i.

```
(defun collect! (lst fn)
  (cond ((endp lst) nil)
        (t (cons (apply! fn (list (car lst)))
                 (collect! (cdr lst) fn)))))
```

A Thought Experiment

```
(defun apply! (fn args)
  (cond ...
   ((eq fn 'REV) (rev! (car args)))
   ((eq fn 'COLLECT) (collect! (car args) (cadr args)))
   ...
   ((eq fn 'APPLY$)
    (if (tamep-functionp (car args))
        (apply! (car args) (cadr args))
        nil))
   (t nil)))
(defun collect! (lst fn)
  (cond ((endp lst) nil)
        (t (cons (apply! fn (list (car lst)))
                 (collect! (cdr lst) fn)))))
```

A Thought Experiment

```
(defun apply! (fn args)
  (cond ...
   ((eq fn 'REV) (rev! (car args)))
   ((eq fn 'COLLECT)
    (collect! (car args) (cadr args)))
   ...
   ((eq fn 'APPLY$)
    (if (tamep-functionp (car args))
        (apply! (car args) (cadr args))
        nil))
   (t nil)))
```

A Thought Experiment

```
(defun apply! (fn args)
  (cond ...
   ((eq fn 'REV) (rev! (car args)))
   ((eq fn 'COLLECT)
    (if (tamep-functionp (cadr args))
        (collect! (car args) (cadr args))
        nil))
   ...
   ((eq fn 'APPLY$)
    (if (tamep-functionp (car args))
        (apply! (car args) (cadr args))
        nil))
   (t nil)))
```

Lesson 3: Check tameness!

A Thought Experiment

```
(defun apply! (fn args)
  (cond ...
   ((eq fn 'REV) (rev! (car args)))
   ((eq fn 'COLLECT)
    (if (tamep-functionp (cadr args))
        (collect! (car args) (cadr args))
        nil))
   ...
   ((eq fn 'APPLY$)
    (if (tamep-functionp (car args))
        (apply! (car args) (cadr args))
        nil))
   (t nil)))
```

What about `collect-rev`?

A Thought Experiment

```
(defun$ rev (x)                                 ; Ordinary
  (if (consp x)
      (ap (rev (cdr x)) (cons (car x) nil))
      nil))
(defun$ collect (lst fn)                        ; Mapping fn
  (cond ((endp lst) nil)
        (t (cons (apply$ fn (list (car lst)))
                 (collect (cdr lst) fn)))))
(defun$ collect-rev (lst)                       ; Tame Instance
  (collect lst 'REV))
```

A Thought Experiment

```
(defun rev! (x)                                      ; Ordinary
  (if (consp x)
      (ap! (rev! (cdr x)) (cons (car x) nil))
      nil))
(defun collect! (lst fn)                             ; Mapping fn
  (cond ((endp lst) nil)
        (t (cons (apply! fn (list (car lst)))
                 (collect! (cdr lst) fn)))))
(defun collect-rev! (lst)                            ; Tame Instance
  (cond ((endp lst) nil)
        (t (cons (rev! (car lst))
                 (collect-rev! (cdr lst))))))
```

A Thought Experiment

```
(defun rev! (x)                                ; Ordinary
  (if (consp x)
      (ap! (rev! (cdr x)) (cons (car x) nil))
      nil))
(defun collect! (lst fn)                       ; Mapping fn
  (cond ((endp lst) nil)
        (t (cons (apply! fn (list (car lst)))
                 (collect! (cdr lst) fn)))))
(defun collect-rev! (lst)                      ; Ordinary
  (cond ((endp lst) nil)
        (t (cons (rev! (car lst))
                 (collect-rev! (cdr lst))))))
```

A Thought Experiment

Lesson 4: The dopplegangers of tame instances are ordinary and should be treated as such in `apply!`.

The Construction

1. Include the book apply-prim.lisp to define `apply$-primp` and `apply$-prim`.

2. Define `f-classes-nonprim` to return the f-classes of all non-primitive functions f_i as computed by `chronology.lisp`.

```
(defun f-classes-nonprim (fn)
  (case fn
    ...
    (REV t)
    ...
    (COLLECT '(NIL :FN))
    ...
    (RUSSELL '(FN :NIL))
    ...))
```

3. Define `f-classes` and the `tamep` clique as in `apply.lisp`.

4. Partition the user's functions into three groups:

- ordinary functions – those independent of `apply$`, `ev$`, and `ev$-list`.
- mapping functions – those having at least one :FN or :EXPR argument
- tame instances – functions defined by calling mapping functions on quoted tame functions and expressions

5. Define dopplegangers for all ordinary functions and for all tame instances.

6. Define the dopplegangers of `apply$`, `ev$`, `ev$-list` and all mapping functions in a mutually recursive clique.

7. Define `apply$-nonprim` to be the part of `apply!` that handles the user's functions (looking for 'f_i and calling $f_i!$).

```
(defun apply$-nonprim (fn args)
  (case fn
    ...
    (REV (rev! (car args)))
    ...
    (COLLECT
     (if (tamep-functionp (cadr args))
         (collect! (car args) (cadr args))
         nil))
    ...
    (COLLECT-REV
     (collect-rev! (car args) (cadr args)))
    ...))
```

8. Copy down the rest of `apply.lisp`.

9. Copy down all of the user's functions (ordinary, mapping, and tame instances) *exactly* as they are defined in `chronology.lisp`.

10. Prove that the dopplegangers of `apply$`, `ev$`, `ev$-list` and all the mapping functions are equal to their correspondents.

```
(defthm doppleganger-equiv-for-mapping-fns
  (and (equal (apply! fn args)
              (apply$ fn args))
       (equal (ev! x a)
              (ev$ x a))
       ...
       (equal (collect! lst fn)
              (collect lst fn))
       ...
       (equal (russell! fn lst)
              (russell fn lst))
       ...))
```

11. Prove that the dopplegangers of the ordinary functions and tame instances are equal to their correspondents.

```
(defthm ap!-is-ap
  (equal (ap! x y) (ap x y)))
(defthm rev!-is-rev
  (equal (rev! x) (rev x)))
...
(defthm collect-rev!-is-collect-rev
  (equal (collect-rev! lst) (collect-rev lst)))
```

The Challenge

How do you invent a measure to explain a mutually recursive clique containing:

- `apply!`
- `ev!`
- `collect!`
- `foldr!`
- ...

My Current Answer

A lexicographic combination of:

1. `apply$`, `ev$`, and `ev$-list` have measure 0; all user mapping fns have measure
   `(if (tamep-functionp fn) 0 1)`

2. combined sizes of fn and the :FN and/or :EXPR arguments

3. the mapping function's "native" measure

4. maximal distance to `apply$`

Examples

```
(apply$2 fn x y)                ⟨1, *, *, *⟩
⇓                                 ≻
(apply$ fn (list x y))          ⟨0, *, *, *⟩



(apply$ 'collect args)          ⟨0, 1+|(cadr args)|, *, *⟩
⇓                                 ≻
(collect (car args)             ⟨0, |(cadr args)|, *, *⟩
         (cadr args))
```

`(collect lst fn)`	$\langle$0, \|fn\|, \|lst\|, *$\rangle$
$\Downarrow$	$\succ$
`(collect (cdr lst) fn)`	$\langle$0, \|fn\|, \|(cdr lst)\|, *$\rangle$

`(collect-tips x fn)`	$\langle$0, \|fn\|, \|x\|, 1$\rangle$
$\Downarrow$	$\succ$
`(apply$ fn (list x))`	$\langle$0, \|fn\|, 0, 0$\rangle$

Successes

This lexicographic measure justifies

```
collect
collect2
collect*
collect-on
collect-tips
collect-from-to
sumlist
sumlist-with-params
filter
all
foldr
foldl
apply$2
apply$2x
apply$2xx
russell
recur-by-collect
prow
prow*
```

If we find a mapping function that `make-applicable` accepts but for which the above construction fails, we must either

- restrict `make-applicable` so that it rejects the mapping function, or
- find a more elaborate construction!

If we find a mapping function that `make-applicable` accepts but for which the above construction fails, we must either

- restrict `make-applicable` so that it rejects the mapping function, or
- find a more elaborate construction!

An alternative: Prohibit user defined mapping functions. Just supply the ones we can justify now and call it done.

Other Issues

`Make-applicable` incorrectly accepts `foo` as a tame instance!

```
(defun foo (x) (apply$ 'foo (list (cons x x))))
```

`Make-applicable` should check that the measure is a bounded ordinal.

`Make-applicable` should check that the mapping function is not mutually recursive.

Prove that the construction works for all functions admited by `make-applicable`!