Adding APPLY to ACL2 (Part 2)

Matt Kaufmann
J Strother Moore

Department of Computer Science
University of Texas at Austin

January, 2016

Background

The `apply` book defines `apply$` and related concepts and provides rules for manipulating them.

The rules allow convenient proof of theorems about such mapping functions as `sumlist`, `collect`, `foldr`, etc.

Sample Theorems from Part 1

```
(equal (sumlist (ap a b) fn)
       (+ (sumlist a fn)
          (sumlist b fn)))

(equal (sumlist a
                '(lambda (x) (binary-* '2 x)))
       (* 2 (sumlist a 'identity)))

(equal (foldr x 'cons y)
       (ap x y))
```

```
(implies (and (Applicablep sq)
              (natp n))
         (equal (sumlist (nats n) 'sq)
                (+ (/ (expt n 3) 3)
                   (/ (expt n 2) 2)
                   (/ n 6))))
```

Part 2

We focus on the rules made available by the `apply` book.

User Perspective

The important concepts are:

- `apply$` and `ev$` (and `ev$-list`)
- `tamep-functionp` and `tamep`
- `f-classes` (used to determine tameness)
- (`make-applicable` f), an event that introduces the (`Applicablep` f) notation.

Manageable Functions

`Apply$` can only handle a function if it has these properties:

- in `:logic` mode
- returns a single value
- does not use `state` or stobjs
- does not require trust tags or restricted syntax to be called

Primitives

There are 798 manageable functions in the Ground Zero theory.

All are built into `apply$`.

Those primitives are recognized by `apply$-primp` and applied by `apply$-prim`.

Classifying Formals

Let f be a user-defined function with formals $(v_1 \ldots v_n)$. v_i has classification:

- `:FN`, if v_i is used exclusively as a function (passed ancestrally to `apply$`)
- `:EXPR`, if v_i is used exclusively as an expression (passed ancestrally to `ev$`)
- `nil`, if v_i is never used as a function or expression, i.e., v_i is "vanilla"

F-Classes

(f-classes $'f$) $=$

- `nil`, if f is not manageable or has an unclassifiable formal

- `t`, if all formals are "vanilla"

- $(c_1 \ldots c_n)$, at least one formal is functional or expressional

Tamep-Functionp

f is a *tame function* iff f is

- a symbol and (`f-classes` f) $=$ `t` (i.e., f is manageable and no formal is used as a function or expression)

- (`lambda` ($v_1 \ldots v_n$) b) and b is a tame term

Tamep

x is *tame term* iff x is

- a variable or `QUOTE`d constant

- `((lambda` $(v_1 \ldots v_n)$ $b)$ $a_1 \ldots a_n)$ where b and the a_i are tame

- $(f\ a_1 \ldots a_n)$ where (`f-classes` f) $=(c_1 \ldots c_n)$ and if c_i is :FN, a_i is a `QUOTE`d tame fn, if c_i is :EXPR, a_i is a `QUOTE`d tame term, and else a_i is tame.

Positive and Negative Examples

```
(binary-+ '1 x)

(sumlist lst 'CAR)

(sumlist lst
        '(lambda (x)
           (binary-+ '1 x)))
(sumlist lst
         '(lambda (x)
            (sumlist x 'CAR)))
(sumlist lst
         '(lambda (x)
            (sumlist x (foo y))))
```

Rules about f-classes

```
(defthm f-classes-primitive
  (implies (apply$-primp f)
           (equal (f-classes f) t)))

(defthm f-classes-apply$
  (equal (f-classes 'APPLY$) '(:FN NIL)))

(defthm f-classes-ev$
  (equal (f-classes 'EV$) '(:EXPR NIL)))
```

Rules about ev$

```
(defthm ev$-def-fact
  (implies (tamep x)
           (equal (ev$ x a)
                  (cond
                   ((variablep x) (cdr (assoc x a)))
                   ((fquotep x) (cadr x))
                   ((eq (car x) 'IF)
                    (if (ev$ (cadr x) a)
                        (ev$ (caddr x) a)
                        (ev$ (cadddr x) a)))
                   (t (apply$ (car x)
                              (ev$-list (cdr x) a)))))))
```

This is stored as several `:rewrite` rules.

Rules about ev$-list

```
(defthm ev$-list-def
  (equal (ev$-list x a)
         (cond
          ((endp x) nil)
          (t (cons (ev$ (car x) a)
                   (ev$-list (cdr x) a))))))
```

Stored as a `:definition` rule.

Rules about apply$

```
(defthm apply$-primitive
  (implies (apply$-primp f)
           (equal (apply$ f args)
                  (apply$-prim f args))))

(defthm beta-reduction
  (equal (apply$ (list 'LAMBDA vars body) args)
         (ev$ body (pairlis$ vars args))))
```

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 $\forall$ 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 $\forall$ 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 $\forall$ so we use `defun-sk` to introduce a `Applicablep-sq` to express that quantified hypothesis.

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

Rules about User-Defined f

You must use (`make-applicable` f) to tell `apply$` about f.

If f is not manageable, (`make-applicable` f) causes an error.

Otherwise, it executes a `defun-sk` to introduce `Applicablep-`f.

Then `make-applicable` proves rules about `Applicablep-`f.

The forms of the `defun-sk` and the rules depend on whether f is tame.

(make-applicablep ap)

```
(defthm apply$-AP
  (implies (force (Applicablep-AP))
           (and (equal (f-classes 'AP) t)
                (equal (apply$ 'AP args)
                       (ap (car args)
                           (cadr args))))))
```

(make-applicablep sumlist)

```
(defthm apply$-SUMLIST
  (and

   (implies (force (Applicablep-SUMLIST))
            (equal (f-classes 'SUMLIST) '(NIL :FN)))

   (implies (and (force (Applicablep-SUMLIST))
                 (tamep-functionp (cadr args)))
            (equal (apply$ 'SUMLIST args)
                   (sumlist (car args)
                            (cadr args))))))
```

Defun-sk for Applicablep-SUMLIST

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

. . . From Which We Can Prove

```
(defthm apply$-SUMLIST
  (and

   (implies (force (Applicablep-SUMLIST))
            (equal (f-classes 'SUMLIST) '(NIL :FN)))

   (implies (and (force (Applicablep-SUMLIST))
                 (tamep-functionp (cadr args)))
            (equal (apply$ 'SUMLIST args)
                   (sumlist (car args)
                            (cadr args))))))
```

Because f-classes-nonprim and apply$-nonprim are undefined, it is impossible to evaluate, prove, or disprove (`Applicablep-SUMLIST`).

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

Vacuity

Can we be sure that there is *some way* to define f-classes-nonprim and apply$-nonprim so that

$$[\forall\ args:$$

```
(f-classes-nonprim 'SUMLIST) = '(NIL :FN)
∧
((tamep-functionp (cadr args))
 →
 (apply$-nonprim 'SUMLIST args)
   =
  (sumlist (car args)
           (cadr args)))]
```

More Precisely

Given any collection of non-erroneous `make-applicable` events can we define f-classes-nonprim and apply$-nonprim so that *all* the `Applicablep-`f hypotheses are true?

This is the subject of Part 3.