Iteration in ACL2

Matt Kaufmann and J Moore (speaker)

The University of Texas at Austin (retired)
Dept. of Computer Science

ACL2 Workshop, May, 2020

Outline

Outline

Introduction

Challenge: Sum the squares of the elements of a list.

- Recursion is natural for equational logic.

```
(defun sum-sq (lst)                    ; ACL2
  (if (endp lst)
      0
      (+ (* (car lst) (car lst))
         (sum-sq (cdr lst)))))
```

Introduction

Challenge: Sum the squares of the elements of a list.

- Recursion is natural for equational logic.

```
(defun sum-sq (lst)                    ; ACL2
  (if (endp lst)
      0
      (+ (* (car lst) (car lst))
         (sum-sq (cdr lst)))))
```

- Iteration is natural for programming.

```
(loop  for x in lst sum (* x x)) ; Common Lisp
```

Introduction

Challenge: Sum the squares of the elements of a list.

- Recursion is natural for equational logic.

```
(defun sum-sq (lst)                    ; ACL2
  (if (endp lst)
      0
      (+ (* (car lst) (car lst))
         (sum-sq (cdr lst)))))
```

- Iteration is natural for programming.

```
(loop  for x in lst sum (* x x)) ; Common Lisp

(loop$ for x in lst sum (* x x)) ; ACL2
```

But the `loop$` body must be *tame*!

TAMENESS

Tame terms must

- not use state or stobjs, and
- have a call tree that can be syntactically determined, *e.g.*,
  - every argument used as a "function" must be quoted, and
  - no "functional" argument may be modified or used in a way other than as a function.

See :DOC tame.

Why Loop$ Is Non-Trivial

The usual mathematical explanation of iteration involves *functionals*: functions that take functions as arguments, e.g., `map`, `foldr`, so that "sum the squares," "sum the cubes," "sum the counts," etc., all use "sum f." But. . .

Why Loop$ Is Non-Trivial

The usual mathematical explanation of iteration involves *functionals*: functions that take functions as arguments, e.g., `map`, `foldr`, so that "sum the squares," "sum the cubes," "sum the counts," etc., all use "sum f." But. . .

- ACL2 is first order (so usual solution is unavailable)

Why Loop$ Is Non-Trivial

The usual mathematical explanation of iteration involves *functionals*: functions that take functions as arguments, e.g., `map`, `foldr`, so that "sum the squares," "sum the cubes," "sum the counts," etc., all use "sum f." But. . .

- ACL2 is first order (so usual solution is unavailable)
- ACL2 requires that functions be total (limited)

Why Loop$ Is Non-Trivial

The usual mathematical explanation of iteration involves *functionals*: functions that take functions as arguments, e.g., `map`, `foldr`, so that "sum the squares," "sum the cubes," "sum the counts," etc., all use "sum f." But. . .

- ACL2 is first order (so usual solution is unavailable)
- ACL2 requires that functions be total (limited)
- ACL2 supports `local` events (???)

Why Loop$ Is Non-Trivial

The usual mathematical explanation of iteration involves *functionals*: functions that take functions as arguments, e.g., `map`, `foldr`, so that "sum the squares," "sum the cubes," "sum the counts," etc., all use "sum f." But. . .

- ACL2 is first order (so usual solution is unavailable)
- ACL2 requires that functions be total (limited)
- ACL2 supports `local` events (???)
- Common Lisp `loop` is *complicated* (hard to formalize)

Why Loop$ Is Non-Trivial

The usual mathematical explanation of iteration involves *functionals*: functions that take functions as arguments, e.g., `map`, `foldr`, so that "sum the squares," "sum the cubes," "sum the counts," etc., all use "sum f." But. . .

- ACL2 is first order (so usual solution is unavailable)
- ACL2 requires that functions be total (limited)
- ACL2 supports `local` events (???)
- Common Lisp `loop` is *complicated* (hard to formalize)
- ACL2 features are well-integrated (details take time)

WHY LOOP$ IS NON-TRIVIAL

The usual mathematical explanation of iteration involves *functionals*: functions that take functions as arguments, e.g., `map`, `foldr`, so that "sum the squares," "sum the cubes," "sum the counts," etc., all use "sum f." But...

- ACL2 is first order (so usual solution is unavailable)
- ACL2 requires that functions be total (limited)
- ACL2 supports `local` events (???)
- Common Lisp `loop` is *complicated* (hard to formalize)
- ACL2 features are well-integrated (details take time)

The breakthrough that made a limited `loop$` possible was `apply$`– which we assume you're now familiar with!

Introduction

Today I will discuss:

Introduction

Today I will discuss:

- how to **use** `loop$` ...

INTRODUCTION

Today I will discuss:

- how to **use** `loop$` ...
  - but see :DOC loop$ for details; and

Introduction

Today I will discuss:

- how to **use** `loop$` ...
  - but see :DOC loop$ for details; and
- a bit about the **implementation** of `loop$` ....

Introduction

Today I will discuss:

- how to **use** `loop$` ...
  - but see :DOC loop$ for details; and
- a bit about the **implementation** of `loop$` ....
  - but for more details see the paper or the ACL2 source code, notably the "Essay on LOOP$" and the "Essay on Evaluation of Apply$ and Loop$ Calls During Proofs".

Introduction

Today I will discuss:

- how to **use** `loop$` ...
  - but see :DOC loop$ for details; and
- a bit about the **implementation** of `loop$` ....
  - but for more details see the paper or the ACL2 source code, notably the "Essay on LOOP$" and the "Essay on Evaluation of Apply$ and Loop$ Calls During Proofs".

More examples may be found in community book `projects/apply/loop-tests.lisp`.

Introduction

Prior work: an Nqthm analogue to `loop$` is `FOR`.

Introduction

Prior work: an Nqthm analogue to `loop$` is `FOR`.

Much as `loop$` depends on `apply$`, `FOR` depended on an evaluator, `V&C$`.

INTRODUCTION

Prior work: an Nqthm analogue to `loop$` is `FOR`.

Much as `loop$` depends on `apply$`, `FOR` depended on an evaluator, `V&C$`.

That sort of universal evaluator isn't possible for ACL2 because of `local`.

THE LOCAL PROBLEM

```
(encapsulate ()
  (local (defun f (x) x))
  (defthm lemma-1 (equal (uni-eval '(f 3)) 3)))
```

BTW: `lemma-1` can be exported because it is not *ancestrally dependent* on `f`.

THE LOCAL PROBLEM

```
(encapsulate ()
  (local (defun f (x) x))
  (defthm lemma-1 (equal (uni-eval '(f 3)) 3)))
```

BTW: `lemma-1` can be exported because it is not *ancestrally dependent* on `f`.

```
(defun f (x) (1+ x))
```

THE LOCAL PROBLEM

```
(encapsulate ()
  (local (defun f (x) x))
  (defthm lemma-1 (equal (uni-eval '(f 3)) 3)))
```

BTW: `lemma-1` can be exported because it is not *ancestrally dependent* on `f`.

```
(defun f (x) (1+ x))

(defthm lemma-2 (equal (uni-eval '(f 3)) 4))
```

THE LOCAL PROBLEM

```
(encapsulate ()
  (local (defun f (x) x))
  (defthm lemma-1 (equal (uni-eval '(f 3)) 3)))
```

BTW: `lemma-1` can be exported because it is not *ancestrally dependent* on `f`.

```
(defun f (x) (1+ x))

(defthm lemma-2 (equal (uni-eval '(f 3)) 4))

(thm nil :hints (("Goal" :in-theory nil
                         :use (lemma-1 lemma-2))))
```

THE LOCAL PROBLEM

```
(encapsulate ()
  (local (defun f (x) x))
  (defthm lemma-1 (equal (uni-eval '(f 3)) 3)))
```

BTW: `lemma-1` can be exported because it is not *ancestrally dependent* on `f`.

```
(defun f (x) (1+ x))

(defthm lemma-2 (equal (uni-eval '(f 3)) 4))

(thm nil :hints (("Goal" :in-theory nil
                         :use (lemma-1 lemma-2))))
```

BTW: `'(f 3) = (cons (intern-in-package-of-symbol (coerce (list (code-char 70)) 'string) 'ACL2) '(3))`

Outline

Syntax and Semantics

Semantics are given by translating `loop$` expressions into the ACL2 logic.

Syntax and Semantics

Semantics are given by translating `loop$` expressions into the ACL2 logic. For example,

```
(loop$ for x in '(1 2 3 4) sum (* x x))
```

essentially translates to the term

```
(sum$ '(LAMBDA (X) (BINARY-* X X))
      '(1 2 3 4))
```

Syntax and Semantics

Semantics are given by translating `loop$` expressions into the ACL2 logic. For example,

```
(loop$ for x in '(1 2 3 4) sum (* x x))
```

essentially translates to the term

```
(sum$ '(LAMBDA (X) (BINARY-* X X))
      '(1 2 3 4))
```

where *essentially*

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

Syntax and Semantics

Here is a more complex example showing introduction of *loop$ scions* `collect$`, `when$`, and `until$`.

```
ACL2 !>(loop$ for i from 0 to 1000000 by 5
              until (> i 30)
              when (evenp i) collect (* i i))
(0 100 400 900)
ACL2 !>
```

Syntax and Semantics

Here is a more complex example showing introduction of *loop$ scions* `collect$`, `when$`, and `until$`.

```
ACL2 !>(loop$ for i from 0 to 1000000 by 5
              until (> i 30)
              when (evenp i) collect (* i i))
(0 100 400 900)
ACL2 !>
```

The translation of this `loop$` expression is *essentially*:

```
(COLLECT$
 '(LAMBDA (I) (BINARY-* I I))
 (WHEN$ '(LAMBDA (I) (EVENP I))
        (UNTIL$ '(LAMBDA (I) (< '30 I))
                (FROM-TO-BY '0 '1000000 '5))))
```

SYNTAX AND SEMANTICS

The *actual* translation using `:trans` (see the paper):

```
(RETURN-LAST
 'PROGN
 '(LOOP$ FOR I FROM 0 TO 1000000 BY 5 UNTIL (> I 30)
         WHEN (EVENP I)
         COLLECT (* I I))
 (COLLECT$ '(LAMBDA (I)
                    (DECLARE (IGNORABLE I))
                    (RETURN-LAST 'PROGN
                                 '(LAMBDA$ (I) (* I I))
                                 (BINARY-* I I)))
           (WHEN$ '(LAMBDA (I)
                           (DECLARE (IGNORABLE I))
                           (RETURN-LAST 'PROGN
                                        '(LAMBDA$ (I) (EVENP I))
                                        (EVENP I)))
                  (UNTIL$ '(LAMBDA (I)
                                   (DECLARE (IGNORABLE I))
                                   (RETURN-LAST 'PROGN
                                                '(LAMBDA$ (I)
                                                          (> I 30))
                                                (< '30 I)))
                          (FROM-TO-BY '0 '1000000 '5)))))
```

Syntax and Semantics

See the paper for more about syntax and semantics....

Syntax and Semantics

See the paper for more about syntax and semantics....

There are more syntactic constructs available:

- types and guards;
- iterating on tails of a list;
- loop operators `ALWAYS`, `THEREIS` and `APPEND` (in addition to `COLLECT`, and `SUM`); and
- *fancy loops*, which have

Syntax and Semantics

See the paper for more about syntax and semantics....

There are more syntactic constructs available:

- types and guards;
- iterating on tails of a list;
- loop operators `ALWAYS`, `THEREIS` and `APPEND` (in addition to `COLLECT`, and `SUM`); and
- *fancy loops*, which have
  - more than one iteration variable, OR
  - reference to variables other than the iteration variable.

Syntax and Semantics

See the paper for more about syntax and semantics....

There are more syntactic constructs available:

- types and guards;
- iterating on tails of a list;
- loop operators `ALWAYS`, `THEREIS` and `APPEND` (in addition to `COLLECT`, and `SUM`); and
- *fancy loops*, which have
  - more than one iteration variable, OR
  - reference to variables other than the iteration variable.

Syntactic restrictions are eased for theorems (as opposed to definitions), e.g.:

```
(thm (equal (loop\$ for x in '(A B C)
                    collect (mv x x))
            '((A A) (B B) (C C))))
```

Outline

Support for Generic Reasoning with Loop$

Loop$ supports not only concise programming but also concise *reasoning*. Here's an example.

Support for Generic Reasoning with Loop$

Loop$ supports not only concise programming but also concise *reasoning*. Here's an example.

```
(defun sum-lengths (lst)
  (loop$ for x in lst sum (length x)))

; Phase 2: Lemmas

(thm (equal (sum-lengths (reverse x))
            (sum-lengths x)))
```

Support for Generic Reasoning with Loop$

Loop$ supports not only concise programming but also concise *reasoning*. Here's an example.

```
(defun sum-lengths (lst)
  (loop$ for x in lst sum (length x)))
(defthm sum$-revappend ; need shown by checkpoint
  (equal (sum$ fn (revappend x y))
         (+ (sum$ fn x) (sum$ fn y))))
(thm (equal (sum-lengths (reverse x))
            (sum-lengths x)))
```

Support for Generic Reasoning with Loop$

Loop$ supports not only concise programming but also concise *reasoning*. Here's an example.

```
(defun sum-lengths (lst)
  (loop$ for x in lst sum (length x)))
(defthm sum$-revappend ; need shown by checkpoint
  (equal (sum$ fn (revappend x y))
         (+ (sum$ fn x) (sum$ fn y))))
(thm (equal (sum-lengths (reverse x))
            (sum-lengths x)))

(defun sum-acl2-counts (lst)
  (loop$ for x in lst sum (acl2-count x)))
; This is now automatic; no new lemma is required.
(thm (equal (sum-acl2-counts (reverse x))
            (sum-acl2-counts x)))
```

Support for Generic Reasoning with Loop$

Loop$ supports not only concise programming but also concise *reasoning*. Here's an example.

```
(defun sum-lengths (lst)
  (loop$ for x in lst sum (length x)))
(defthm sum$-revappend ; need shown by checkpoint
  (equal (sum$ fn (revappend x y))
         (+ (sum$ fn x) (sum$ fn y))))
(thm (equal (sum-lengths (reverse x))
            (sum-lengths x)))

(defun sum-acl2-counts (lst)
  (loop$ for x in lst sum (acl2-count x)))
; This is now automatic; no new lemma is required.
(thm (equal (sum-acl2-counts (reverse x))
            (sum-acl2-counts x)))
```

If the two functions were defined in the usual way, we would need a lemma about revappend for each one.

Outline

WARRANT HYPOTHESES

`Loop$` scions invoke `apply$`. But `apply$` is undefined on user-defined function symbols.

Warrant Hypotheses

Loop$ scions invoke apply$. But apply$ is undefined on user-defined function symbols.

Warrant hypotheses tell apply$ how to handle user functions.

```
(DEFTHM APPLY$-SQUARE
  (IMPLIES (FORCE (APPLY$-WARRANT-SQUARE))
           (AND (EQUAL (BADGE 'SQUARE)
                       '(APPLY$-BADGE 1 1 . T))
                (EQUAL (APPLY$ 'SQUARE ARGS)
                       (SQUARE (CAR ARGS)))))
  :HINTS ...)
```

See our JAR paper [1]. (Apply$-warrant-$f$) is ancestrally dependent on f! (Apply$-warrant-$f$) can be abbreviated (warrant f).

WARRANT HYPOTHESES

NOTE: use this `include-book` for `apply$` or `loop$` reasoning.

```
(include-book "projects/apply/top" :dir :system)
(defun$ square (n)
  (declare (xargs :guard (integerp n)))
  (* n n))
```

WARRANT HYPOTHESES

NOTE: use this `include-book` for `apply$` or `loop$` reasoning.

```
(include-book "projects/apply/top" :dir :system)
(defun$ square (n)
  (declare (xargs :guard (integerp n)))
  (* n n))
```

The `defun$` form above provides the `defun` and the warrant:

```
ACL2 !>:trans1 (defun$ square (n)
                 (declare (xargs :guard (integerp n)))
                 (* n n))
 (PROGN (DEFUN SQUARE (N)
              (DECLARE (XARGS :GUARD (INTEGERP N)))
              (* N N))

        (DEFWARRANT SQUARE)) ;define warrant
                             ;if possible
```

Warrant Hypotheses

```
(include-book "projects/apply/top" :dir :system)
(defun$ square (n)
  (declare (xargs :guard (integerp n)))
  (* n n))
```

Warrant Hypotheses

```
(include-book "projects/apply/top" :dir :system)
(defun$ square (n)
  (declare (xargs :guard (integerp n)))
  (* n n))

(thm (equal (apply$ 'square '(5)) 25)) ;FAILS!
```

You can't prove anything interesting about `apply$` on a user-defined symbol without the warrant!

Warrant Hypotheses

```
(include-book "projects/apply/top" :dir :system)
(defun$ square (n)
  (declare (xargs :guard (integerp n)))
  (* n n))

(thm (equal (apply$ 'square '(5)) 25)) ;FAILS!
```

You can't prove anything interesting about `apply$` on a user-defined symbol without the warrant!

```
(thm                                    ;SUCCEEDS!
 (implies (apply$-warrant-square)
          (equal (apply$ 'square '(5)) 25)))
```

There is *always* a model in which all your warrant hypotheses are true!

Outline

EVALUATION

In the ACL2 *evaluation theory*, all warrants have an attachment to the true function, so you can run `apply$` and `loop$` terms.

`Loop$` *essentially* turns into `loop` in raw Common Lisp, so Common Lisp `loop` is run when evaluating `loop$` expressions in guard-verified functions (provided attachments are allowed).

EVALUATION

In the ACL2 *evaluation theory*, all warrants have an attachment to the true function, so you can run `apply$` and `loop$` terms.

`Loop$` *essentially* turns into `loop` in raw Common Lisp, so Common Lisp `loop` is run when evaluating `loop$` expressions in guard-verified functions (provided attachments are allowed). Consider the following example.

```
(defun sum-acl2-counts (lst)
  (declare (xargs :guard (true-listp lst)
                  :verify-guards nil))
  (loop$ for x in lst sum (acl2-count x)))

(defconst *lst* (from-to-by 1 (expt 10 6) 1))
```

Evaluation

```
; Not in a function body:
(loop$ for x in *lst* sum (acl2-count x))
0.13 seconds (calls sum$)
```

Evaluation

```
; Not in a function body:
(loop$ for x in *lst* sum (acl2-count x))
0.13 seconds (calls sum$)

; In non-guard-verified function body:
(sum-acl2-counts *lst*)
0.13 seconds (calls sum$)
```

Evaluation

```
; Not in a function body:
(loop$ for x in *lst* sum (acl2-count x))
0.13 seconds (calls sum$)

; In non-guard-verified function body:
(sum-acl2-counts *lst*)
0.13 seconds (calls sum$)

(verify-guards sum-acl2-counts)
```

Evaluation

```
; Not in a function body:
(loop$ for x in *lst* sum (acl2-count x))
0.13 seconds (calls sum$)

; In non-guard-verified function body:
(sum-acl2-counts *lst*)
0.13 seconds (calls sum$)

(verify-guards sum-acl2-counts)

; In guard-verified function body:
(sum-acl2-counts *lst*)
0.01 seconds (uses CL loop)
```

Evaluation

```
; Not in a function body:
(loop$ for x in *lst* sum (acl2-count x))
0.13 seconds (calls sum$)

; In non-guard-verified function body:
(sum-acl2-counts *lst*)
0.13 seconds (calls sum$)

(verify-guards sum-acl2-counts)

; In guard-verified function body:
(sum-acl2-counts *lst*)
0.01 seconds (uses CL loop)

; In a proof:
(thm (equal (sum-acl2-counts *lst*) 500000500000))
1.58 seconds (calls sum$)
```

EVALUATION

There is a subtlety for evaluation during proofs:

EVALUATION

There is a subtlety for evaluation during proofs:

Warrant hypotheses may be required!
(Attachments aren't allowed during proofs.)

EVALUATION

There is a subtlety for evaluation during proofs:

Warrant hypotheses may be required!
(Attachments aren't allowed during proofs.)

The solution involves tracking the required warrants and then forcing them when necessary.

EVALUATION (4)

Time permitting, I may say a few words about the implementation.

```
#-acl2-loop-only
(defmacro loop\$ (&whole loop$-form &rest args)
  (let ((term
         (or (loop$-alist-term
              loop$-form
              *hcomp-loop$-alist*)
             (loop$-alist-term
              loop$-form
              (global-val 'loop$-alist
                          (w *the-live-state*))))))
    `(cond (*aokp*
            (loop ,@(remove-loop$-guards args)))
           (t ,(or term
                   '(error "...."))))))
```

Outline

Limitations and Future Work

- Apply$ restrictions
  - Logic mode, tame functions

```
(defun foo (x)  ; illegal:  foo isn't yet tame
  (if (atom x)
      (list x)
    (loop$ for y in x append (foo y))))
```

  - No state or stobjs

Limitations and Future Work

- Apply$ restrictions
  - Logic mode, tame functions

```
(defun foo (x)  ; illegal:  foo isn't yet tame
  (if (atom x)
      (list x)
    (loop$ for y in x append (foo y))))
```

  - No state or stobjs
- Common Lisp loop supports more general forms than loop$, e.g.:

```
? (loop for x in '(2 20 5 50 3 30) by #'cddr
        maximize x)
5
? (loop for i from 11/2 downto 1 by 2 collect i)
(11/2 7/2 3/2)
?
```

Limitations and Future Work

- Top-level evaluation does not use Common Lisp `loop`; maybe insist on the use of `top-level`?

Limitations and Future Work

- Top-level evaluation does not use Common Lisp `loop`; maybe insist on the use of `top-level`?

```
ACL2 !>(time$ (loop$ for i from 1 to 10000000 sum i))
; (EV-REC *RETURN-LAST-ARG3* ...) took
; 1.33 seconds realtime, 1.34 seconds runtime
; (320,039,824 bytes allocated).
50000005000000
ACL2 !>(time$
        (top-level (loop$ for i from 1 to 10000000 sum i))
50000005000000
; (EV-REC *RETURN-LAST-ARG3* ...) took
; 0.05 seconds realtime, 0.05 seconds runtime
; (235,648 bytes allocated).
ACL2 !>
```

Limitations and Future Work

- Top-level evaluation does not use Common Lisp `loop`; maybe insist on the use of `top-level`?

```
ACL2 !>(time$ (loop$ for i from 1 to 10000000 sum i))
; (EV-REC *RETURN-LAST-ARG3* ...) took
; 1.33 seconds realtime, 1.34 seconds runtime
; (320,039,824 bytes allocated).
50000005000000
ACL2 !>(time$
        (top-level (loop$ for i from 1 to 10000000 sum i)))
50000005000000
; (EV-REC *RETURN-LAST-ARG3* ...) took
; 0.05 seconds realtime, 0.05 seconds runtime
; (235,648 bytes allocated).
ACL2 !>
```

Note: All bytes allocated in the second evaluation are from the use of `top-level`; none is from the use of `loop$`.

Conclusion

Despite these limitations, we have seen that `loop$` provides efficient execution and can make reasoning more succinct.

Conclusion

Despite these limitations, we have seen that `loop$` provides efficient execution and can make reasoning more succinct.

We expect to evolve its implementation as users tell us what most needs improvement.

Conclusion

Despite these limitations, we have seen that `loop$` provides efficient execution and can make reasoning more succinct.

We expect to evolve its implementation as users tell us what most needs improvement.

More details are (of course) in the paper — and in :DOC loop$ and the ACL2 sources.

This material is based upon work supported in part by DARPA under Contract No. FA8650-17-1-7704. We are also grateful for support by ForrestHunt, Inc., including contributions from Centaur and Oracle.

THANK YOU.

Reference for `apply$`:


M. Kaufmann and J S. Moore.
Limited second-order functionality in a first-order setting.
Journal of Automated Reasoning, 12 2018.