Iteration in ACL2

Matt Kaufmann
The University of Texas at Austin
Dept. of Computer Science

ACL2 Seminar, April 19, 2019

Joint work with J Moore
OUTLINE

Introduction

Syntax and Semantics

Support for Generic Reasoning with Loop$

Warrant Hypotheses

Evaluation

Limitations, Future Work, and Conclusion
OUTLINE

Introduction
Syntax and Semantics
Support for Generic Reasoning with Loop$
Warrant Hypotheses
Evaluation
Limitations, Future Work, and Conclusion
INTRODUCTION

- **Recursion** is natural for equational logic.
INTRODUCTION

- **Recursion** is natural for equational logic.
- **Iteration** is natural for programming.
**INTRODUCTION**

- **Recursion** is natural for equational logic.
- **Iteration** is natural for programming.
- **Loop$** provides both in ACL2; analogous to Common Lisp `loop`
Introduction

Recursion is natural for equational logic.

Iteration is natural for programming.

Loop$ provides both in ACL2;
analogous to Common Lisp loop

ACL2 !>(loop$ for x in '(1 2 3 4) sum (* x x))
30
ACL2 !>:q

Exiting the ACL2 read-eval-print loop....

? (loop for x in '(1 2 3 4) sum (* x x))
30
?

? (loop for x in '(1 2 3 4) sum (* x x))
INTRODUCTION (2)

Today I will discuss:
INTRODUCTION (2)

Today I will discuss:

- how to use loop$ ...
INTRODUCTION (2)

Today I will discuss:

- how to **use** `loop`... 
  - but see :DOC `loop` for details; and
Today I will discuss:

- how to use `loop`...
  - but see :DOC `loop` for details; and
- a bit about the `implementation` of `loop`....
Today I will discuss:

- how to **use** `loop$` ...
  - but see :DOC `loop$` for details; and
- a bit about the **implementation** of `loop$` ....
  - but see the ACL2 source code if you want details, notably the “Essay on Loop$” and the “Essay on Evaluation of Apply$ and Loop$ Calls During Proofs”.

**INTRODUCTION (2)**
Today I will discuss:

- how to use `loop$` ...
  - but see :DOC `loop$` for details; and
- a bit about the **implementation** of `loop$` ...
  - but see the ACL2 source code if you want details, notably the “Essay on Loop$” and the “Essay on Evaluation of Apply$ and Loop$ Calls During Proofs”.

This talk will draw from a paper on this topic (in preparation). Examples may be found in community book `projects/apply/loop-tests.lisp`. 
INTRODUCTION (3)

Prior work: an Nqthm analogue to loop$ is FOR.
INTRODUCTION (3)

Prior work: an Nqthm analogue to loop$ is FOR. Much as loop$ depends on apply$, FOR depended on an evaluator, V&C$.
INTRODUCTION (3)

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.
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\texttt{local}.

\begin{verbatim}
(encapsulate ()
  (local (defun f (x) x))
  (defthm lemma-1 (equal (some-eval '(f 3)) 3)))
(defun f (x) (1+ x))
(defthm lemma-2 (equal (some-eval '(f 3)) 4))
(thm nil :hints ("Goal"
  :in-theory nil
  :use (lemma-1 lemma-2))))
\end{verbatim}
INTRODUCTION

Syntax and Semantics

Support for Generic Reasoning with Loop$

Warrant Hypotheses

Evaluation

Limitations, Future Work, and Conclusion
OUTLINE

Introduction

Syntax and Semantics

Support for Generic Reasoning with Loop$

Warrant Hypotheses

Evaluation

Limitations, Future Work, and Conclusion
SYNTAX AND SEMANTICS

Semantics are given by translating \texttt{loop} expressions into the ACL2 logic.
SYNTAX AND SEMANTICS

Semantics are given by translating \texttt{loop$} expressions into the ACL2 logic. For example,

\begin{verbatim}
(loop$ for x in '(1 2 3 4) sum (* x x))
\end{verbatim}

\textit{essentially} translates to the term

\begin{verbatim}
(sum$ '(LAMBDA (X) (BINARY-* X X))
 ' '(1 2 3 4))
\end{verbatim}
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* — notice apply$:

```
(defun sum$ (fn lst)
  (if (endp lst)
    0
    (+ (apply$ fn (list (car lst)))
       (sum$ fn (cdr lst)))))
```
SYNTAX AND SEMANTICS (2)

Here is a more complex example showing introduction of $\text{loop}$, $\text{scions}$, $\text{collect}$, $\text{when}$, and $\text{until}$.

ACL2 !>(loop for i from 0 to 100 by 5
    until (> i 30)
    when (evenp i) collect (* i i))
(0 100 400 900)
ACL2 !>
SYNTAX AND SEMANTICS (2)

Here is a more complex example showing introduction of \texttt{loop}\$ \textit{scions} \texttt{collect}\$, \texttt{when}\$, \textbf{and} \texttt{until}\$.

ACL2 $\rightarrow$ (\texttt{loop$\ for\ i\ from\ 0\ to\ 100\ by\ 5$
\texttt{until} \ (\texttt{> i 30}$
\texttt{when} \ (\texttt{evenp i}) \texttt{collect} \ (\texttt{* i i}))
(0 100 400 900)
ACL2 $\rightarrow$

The translation of this \texttt{loop}\$ expression is \textit{essentially}:

(C\texttt{OLLECT$\ '(LAMBDA (I) (BINARY$-* I I))$
\texttt{(WHEN$\ '(LAMBDA (I) (EVENP I))$
\texttt{(UNTIL$\ '(LAMBDA (I) (< '30 I))$
\texttt{(FROM$-TO-BY '0 '100 '5))$)})
SYNTAX AND SEMANTICS (3)

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

(RETURN-LAST
 'PROGN
 '(LOOP$ FOR I FROM 0 TO 100 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 '100 '5))))
Introduction Syntax and Semantics Support for Generic Reasoning with \texttt{Loop}$ \textit{Warrant Hypotheses Evaluation Limitations, Future Work, and Conclusion}$
OUTLINE

Introduction

Syntax and Semantics

Support for Generic Reasoning with Loop$

Warrant Hypotheses

Evaluation

Limitations, Future Work, and Conclusion
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)))
```

; Lemmas? Step 2 [joke]

```
(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.

```lisp
(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)))
```

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

```lisp
(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)))
```

```lisp
(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

Introduction

Syntax and Semantics

Support for Generic Reasoning with $\text{Loop}$

Warrant Hypotheses

Evaluation

Limitations, Future Work, and Conclusion
OUTLINE

Introduction

Syntax and Semantics

Support for Generic Reasoning with Loop$

Warrant Hypotheses

Evaluation

Limitations, Future Work, and Conclusion
**Warrant Hypotheses**

Loop$ scions invoke apply$, which is a function with weak constraints.
WARRANT HYPOTHESES

Loop$ scions invoke apply$, which is a function with weak constraints.

Key property needed for applying a user-defined function, $F$: a warrant hypothesis, (apply$-warrant-$F), which implies:

\[
\begin{align*}
\text{(equal} & \text{ (apply$'F (list t_1 \ldots t_n) ) } \\
& (F \ t_1 \ldots t_n) ) .
\end{align*}
\]
WARRANT HYPOTHESES

Loop$ scions invoke apply$, which is a function with weak constraints.

Key property needed for applying a user-defined function, \( F \): a warrant hypothesis, \((\text{apply$-\text{warrant$-} F})\), which implies:

\[
\text{equal} \ (\text{apply$ ' F \ (\text{list} \ t_1 \ \ldots \ t_n))} \ \\
(F \ t_1 \ \ldots \ t_n) .
\]

More background on apply$ is in our JAR paper [1].
WARRANT HYPOTHESES

Loop$ scions invoke apply$, which is a function with weak constraints.

Key property needed for applying a user-defined function, $F$: a warrant hypothesis, (apply$-warrant-F), which implies:

(equal (apply$ 'F (list t₁ ... tₙ))
      (F t₁ ... tₙ)).

More background on apply$ is in our JAR paper [1]. Aside: (apply$-warrant-F) is sometimes written (warrant $F$).
Loop$ scions invoke apply$, which is a function with weak constraints.

Key property needed for applying a user-defined function, $F$: a warrant hypothesis, $(apply$-warrant-$F)$, which implies:

$$(\text{equal } (apply$ 'F (list $t_1 \ldots t_n))$$
$$(F t_1 \ldots t_n)).$$

More background on apply$ is in our JAR paper [1]. Aside: (apply$-warrant-$F)$ is sometimes written (warrant $F$).

We illustrate reasoning about loop$ with an example....
WARRANT HYPOTHESES (2)

**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))
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))

ACL2 !>
WARRANT HYPOTHESES (3)

Here is the key property of the warrant hypothesis for square, (apply$-warrant-square).
WARRANT HYPOTHESES (3)

Here is the key property of the warrant hypothesis for \texttt{square}, \(\text{apply}\$\text{-warrant}\text{-square}\).

\begin{verbatim}
(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 ...)
\end{verbatim}
**Warrant Hypotheses (3)**

Here is the key property of the warrant hypothesis for `square`, `(apply$-warrant-square).

```
(DEFTHM APPLY$-SQUARE
  (IMPLIES (FORCE (APPLY$-WARRANT-SQUARE))
    (AND (EQUAL (BADGE 'SQUARE) '(APPLY$-BADGE 1 1 . T))
         (EQUAL (APPLY$ 'SQUARE ARG5) (SQUARE (CAR ARG5))))
     :HINTS ...)
```

It is **forced** so that a proof can proceed (to a forcing round) even when the warrant hypothesis is missing from the conjecture.
WARRANT HYPOTHESES (3)

Here is the key property of the warrant hypothesis for square, 
(apply$-warrant-square).

(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 ...)

It is forced so that a proof can proceed (to a forcing round) even when the warrant hypothesis is missing from the conjecture.

Continuing with our example....
WARRANT HYPOTHESES (4)

(defun f2 (lower upper)
    (declare (xargs :guard (and (integerp lower)
                               (integerp upper))))
    (loop$ for i of-type integer from lower to upper
           collect (square i)))

(assert-event (equal (f2 3 5) '(9 16 25)))

(thm (implies
      (and (warrant square) ; required
           (natp k1) (natp k2) (natp k3)
           (<= k1 k2) (<= k2 k3))
      (member (* k2 k2) (f2 k1 k3))))
WARRANT HYPOTHESES (5)

Let’s look at a simplified the base case in the induction proof. Note: 
\((\text{lambda}\ldots)\) is essentially just ’ \((\text{lambda} \ldots)\), but \text{lambda}\$ allows untranslated terms.
WARRANT HYPOTHESES (5)

Let's look at a simplified base case in the induction proof. 

Note: (lambda$... ) is essentially just ' (lambda ...), but lambda$ allows untranslated terms.

(IMPLIES
  (AND (APPLY$-WARRANT-SQUARE) ; warrant hypothesis
       (INTEGERP K3) (INTEGERP K1)
       (<= 0 K1) (<= K1 K3))
  (MEMBER-EQUAL (* K1 K1)
    (COLLECT$ (LAMBDA$ (I)
                   (DECLARE ...)
                   (SQUARE I))
               (FROM-TO-BY K1 K3 1)))))
WARRANT HYPOTHESES (5)

Let's look at a simplified the base case in the induction proof.

Note: $(\text{lambda}$ $\ldots)$ is essentially just $'(\text{lambda}$ $\ldots)$, but $\text{lambda}$ allows untranslated terms.

\[
\text{(IMPLIES}
\begin{align*}
&\text{(AND (APPLY$-$WARRANT$-$SQUARE)) ; warrant hypothesis} \\
&\text{(INTEGERP K3) (INTEGERP K1)} \\
&\text{(<= 0 K1) (<= K1 K3))} \\
&\text{(MEMBER$-$EQUAL \((\ast K1 K1)\))} \\
&\text{(COLLECT$ (LAMBDA$ (I) \\
&\text{\textbf{(DECLARE}} \ldots) \\
&\text{\textbf{(SQUARE I))}}} \\
&\text{\textbf{(FROM$-$TO$-$BY K1 K3 1))))}
\end{align*}
\]

Follows from this simplification, by the warrant hypothesis:
WARRANT HYPOTHESES (5)

Let’s look at a simplified the base case in the induction proof.

Note: \((\text{lambda}\ldots)\) is essentially just ‘(lambda \ldots), but \text{lambda}\$ allows untranslated terms.

\[
\begin{align*}
&\text{(IMPLIES} \\
&\quad \text{(AND} \\
&\quad \quad (\text{APPLY}\$-\text{WARRANT}\$-\text{SQUARE}) ; \text{ warrant hypothesis} \\
&\quad \quad \text{(INTEGERP K3)} \ (\text{INTEGERP K1)} \\
&\quad \quad (\leq 0 \ K1) \ (\leq \ K1 \ K3)) \\
&\quad \ (\text{MEMBER-EQUAL} \ (* \ K1 \ K1) \\
&\quad \quad \ (\text{COLLECT}\$ \ (\text{LAMBDA}\$ \ (I) \\
&\quad \quad \quad \ (\text{DECLARE} \ \ldots) \\
&\quad \quad \quad \ (\text{SQUARE} \ I)) \\
&\quad \quad \quad \ (\text{FROM-TO-BY} \ K1 \ K3 \ 1)))) \\
&\text{Follows from this simplification, by the warrant hypothesis:}
\end{align*}
\]

\[(\text{APPLY}\$ \ '\text{SQUARE} \ (\text{LIST} \ K1)) = (* \ K1 \ K1).\]
OUTLINE

Introduction

Syntax and Semantics

Support for Generic Reasoning with Loop$

Warrant Hypotheses

Evaluation

Limitations, Future Work, and Conclusion
EVALUATION

Common Lisp loop is run when evaluating loop$ expressions under guard-verified function calls.
EVALUATION

Common Lisp `loop` is run when evaluating `loop$` expressions under `guard`-verified function calls. The paper has an example illustrating an order of magnitude speed-up in this case, compared to evaluation of `loop$` using `loop$` scions. Consider the following example.

```lisp
(include-book "projects/apply/top" :dir :system)
(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* '(a (b c) "hello")
(trace$ sum$)
```

25/34
Evaluation

Common Lisp loop is run when evaluating loop\$ expressions under guard-verified function calls. The paper has an example illustrating an order of magnitude speed-up in this case, compared to evaluation of loop\$ using loop\$ scions. Consider the following example.

(include-book "projects/apply/top" :dir :system)

(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* '(a (b c) "hello"))

(trace$ sum$)
EVALUATION (2)

; Not in a function body: calls sum$
(loop$ for x in *lst* sum (acl2-count x))
**Evaluation (2)**

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

; In non-guard-verified function body: calls sum$
(sum-acl2-counts *lst*)
EVALUATION (2)

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

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

(verify-guards sum-acl2-counts)
**EVALUATION (2)**

; Not in a function body: **calls sum$**
(\text{loop$ for } x \text{ in } \ast \text{lst}\ast \text{ sum (acl2-count } x))

; In non-guard-verified function body: **calls sum$**
(sum-acl2-counts \ast \text{lst}\ast)

(verify-guards sum-acl2-counts)

; In guard-verified function body: **DOES NOT call sum$**
(sum-acl2-counts \ast \text{lst}\ast)
**EVALUATION (2)**

; Not in a function body: *calls sum*
(\text{loop} for \text{x in } *\text{lst}* \text{ sum (acl2-count x)})

; In non-guard-verified function body: *calls sum*
(sum-acl2-counts *\text{lst}*)

(\text{verify-guards sum-acl2-counts})

; In guard-verified function body:
; DOES NOT call *sum*
(sum-acl2-counts *\text{lst}*)

; In a proof: *calls sum*
; (even though the function is guard-verified)
(thm (equal (sum-acl2-counts *\text{lst}*) 7))
**EVALUATION (3)**

There is a subtlety for evaluation during proofs:
There is a subtlety for evaluation during proofs:

Warrant hypotheses may be required!
(Attachments aren’t allowed during proofs.)
EVALUATION (3)

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.

```lisp
#-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

Introduction

Syntax and Semantics

Support for Generic Reasoning with Loop$

Warrant Hypotheses

Evaluation

Limitations, Future Work, and Conclusion
LIMITATIONS AND FUTURE WORK

- Apply restrictions
  - Logic mode, tame functions
    
    ```lisp
    (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 (2)

- Top-level evaluation does not use Common Lisp `loop`; maybe insist on the use of `top-level`?
LIMITATIONS AND FUTURE WORK (2)

- 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).
ACL2 !>(time$
  (top-level (loop$ for i from 1 to 10000000 sum i)))
  50000005000000
ACL2 !>
LIMITATIONS AND FUTURE WORK (2)

- 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).

ACL2 !>(time$ (top-level (loop$ for i from 1 to 10000000 sum i)))
50000005000000
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 \texttt{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 \texttt{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 \texttt{:DOC loop} and the ACL2 sources.
THANK YOU.

Reference for apply$: