Iteration in ACL2

Matt Kaufmann and J Strother Moore
Department of Computer Science
The University of Texas at Austin, Austin, TX, USA
{kaufmann,moore}@cs.utexas.edu

Iterative algorithms are traditionally expressed in ACL2 using recursion. On the other hand, Common Lisp provides a construct, `loop`, which — like most programming languages — provides direct support for iteration. We describe an ACL2 analogue `loop$` of `loop` that supports efficient ACL2 programming and reasoning with iteration.

1 Introduction

Recursion is a natural way to define notions in an equational logic. But for programming, iteration is often more natural and concise than recursion.

This paper introduces an ACL2 iteration construct, `loop$`. We give an overview of its usage in programming and reasoning, and we touch on some key aspects of its implementation. Those who want more complete user-level documentation are welcome to see :DOC loop$ [1], and those who want more implementation details may visit Lisp comments in the ACL2 sources, in particular the "Essay on Loop$" and the "Essay on Evaluation of Apply$ and Loop$ Calls During Proofs".

The rest of this introduction illustrates the basics of `loop$` and then mentions some related prior work. Next, Section 2 outlines the syntax of `loop$` expressions. Section 3 provides necessary background on `apply$` and especially *warrants*. Then Section 4 gives semantics to `loop$` expressions by translating them into the logic, which provides background for the discussion of reasoning about `loop$` in Section 5. Section 6 discusses evaluation of calls of `loop$`, including considerations involving guards. We conclude by considering possible future enhancements to `loop$`.

The community book `projects/apply/loop-tests.lisp` constitutes supporting materials for this paper: it contains most ACL2 events and expressions discussed below.

1.1 `Loop$` basics

Common Lisp supports iteration with the construct `loop`, and ACL2 provides the analogous (though less general) construct `loop$`.[2] These two forms evaluate to 30, in Common Lisp and in ACL2, respectively.

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

A call of `loop$` is given semantics by translating it to a logical *translated* *term* that is a call of a *`loop$` scion*, in the spirit of a higher-order function call. For example the translation of the `loop$` form above is the following call of the `loop$` scion, `sum$`, whose first argument is a quoted *lambda* *object*.

[1] In the online version of this paper, we follow the usual convention of adding links to documentation on the web [4]. Sometimes we say "see :DOC *topic*", but other times we simply underline a documentation topic.

[2] The dollar sign is commonly used as a suffix in ACL2 to distinguish from a similar Common Lisp utility. Note that the symbol `loop` in the `"COMMON-LISP"` package is one of the 978 external symbols in that package that are imported into the `"COMMON-LISP"` package, so we cannot distinguish `acl2::loop` from `common-lisp::loop`; these are the same symbol.

G. Passmore, R. Gamboa (Eds.): ACL2 Workshop 2020
EPTCS ??, 2020, pp. 1–16, doi:10.4204/EPTCS.??.??

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

Sum$ has the following definition, ignoring declarations and calls of mbe and fix.

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

Notice the use of apply$, to apply a given function symbol or lambda object to a list of arguments. Support for loop$ thus depends on the sort of "higher-order" capability provided by apply$. (We return to apply$ in Section 3.)

In the following example we use a different loop$ scion, collect$, to collect into a list instead of summing results. Also, we filter by considering only even numbers that are multiples of 5 (thus: multiples of 10), stopping when we hit a number greater than 30.

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

This time we get essentially the following translation of the loop$ form. Reading it bottom-up (inside-out), we see that from-to-by builds a list of integers (0 5 10 15 ... 1000000), which until$ reduces to (0 5 10 15 20 25 30) by stopping when 30 is exceeded; then when$ restricts that list to the even integers to produce the list (0 10 20 30), and finally collect$ assembles the squares of the members of that list to produce the result, (0 100 400 900). We believe that this compositional, though computationally inefficient, formal semantics is useful for reasoning about applications of loop$ since lemmas can be proved to simplify the respective steps.

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

Loop$ scions (including collect$, when$, and until$) repeatedly call apply$ on a given lambda object (or function symbol). In Section 3 we explore consequences of these uses of apply$.

If we define a guard-verified function using a corresponding call of loop$, then evaluation is more efficient. Consider the following variation of the example just above, which adds an of-type clause to aid in guard verification (more on that is in Section 6). The include-book event shown is important for almost all work that involves apply$ or loop$.

```
ACL2 !>(include-book "projects/apply/top" :dir :system)
[[.. output omitted ..]]
ACL2 !>(defun f1 ()
         (declare (xargs :guard t))
         (loop$ for i of-type integer from 0 to 1000000 by 5
```

```
               until (> i 30)
               when (evenp i) collect (* i i)))
[[.. output omitted ..]]
ACL2 !>(f1)
(0 100 400 900)
ACL2 !>
```

After `f1` is admitted with guards verified, its Common Lisp definition is used for evaluation (as usual). The Common Lisp macroexpansion of the `loop$` form above is essentially as follows, which presumably allows the Common Lisp compiler to generate efficient code, in particular avoiding the indirection of `apply$`, the repeated passes implied by our compositional semantics, and (presumably) use of the heap to form the intermediate lists noted above.

```
(LOOP FOR I OF-TYPE INTEGER FROM 0 TO 1000000 BY 5
      UNTIL (> I 30)
      WHEN (EVENP I) COLLECT (* I I))
```

1.2 Before ACL2

The addition of an iteration construct within the first-order logic of the Boyer-Moore provers has been a longstanding goal. It was first explored by Moore in 1975 [6]. The Nqthm prover [2] of Boyer and Moore provided an iterative construct, called `FOR`, which provided simple loops such as

```
(FOR 'X (APPEND U V)
     '(MEMBER X B)
     'COLLECT '(TIMES X C)
     (LIST (CONS 'B B) (CONS 'C C))).
```

The expression above can be written in Common Lisp as

```
(loop for x in (append u v) when (member x b) collect (* x c)).
```

Nqthm's `FOR` was defined using a universal interpreter for Nqthm called `V&C$` and many familiar theorems about `FOR` were proved by Nqthm [1]. However, `V&C$` was a non-constructive function [5] and, in addition, was not compatible with local events; Nqthm, unlike ACL2, did not support `local`. Thus `V&C$` was abandoned for ACL2.

2 Syntax

The syntax of `loop$` is a restriction of the syntax of Common Lisp's `loop` macro:

```
(LOOP$ FOR v1 OF-TYPE s1 tgt1       ; where OF-TYPE s1 is optional
; The following AS clauses are optional.
       AS  v2 OF-TYPE s2 tgt2       ; where OF-TYPE s2 is optional
...
       AS  vn OF-TYPE sn tgtn       ; where OF-TYPE sn is optional
; The following UNTIL and WHEN clauses are optional.
       UNTIL :GUARD gu u            ; where :GUARD gu is optional
       WHEN  :GUARD gw w            ; where :GUARD gw is optional
       op    :GUARD gb b            ; where :GUARD gb is optional
       )
```

where the v_i are distinct variables, the *iteration variables*; each `OF-TYPE` s_i is optional where s_i is a type-spec; each tgt_i is a *target clause* (see below); each of g_u, u, g_w, w, g_b, and b is a term; and `op` is one of the *loop operators* `ALWAYS`, `THEREIS`, `APPEND`, `COLLECT`, or `SUM`. We call b the *loop body*. See :DOC loop$ for restrictions, e.g., on terms being in `:logic` mode and *tame* — where user-defined function symbols all have warrants (see Section 3) — and on not allowing both a `WHEN` clause and an `ALWAYS` or `THEREIS` operator.

A *target clause* has one of the following forms, where *lst* is a term denoting a list, *lo* and *hi* are terms denoting integers, and the *step expression s* (if supplied) is a term denoting a positive integer.

- `IN` *lst*
- `ON` *lst*
- `FROM` *lo* `TO` *hi*
- `FROM` *lo* `TO` *hi* `BY` *s*

The corresponding *target list* is *lst* in the first case, the list of non-empty tails of *lst* in the second, the list of integers $(lo\ \ lo+1\ \ \ldots\ \ hi)$ in the third case, and the list of integers $(lo\ \ lo+s\ \ \ldots)$ terminating just before exceeding *hi* in the fourth case.

3 `Apply$` and Warrants

All of the `loop$` scions are defined using calls of `apply$`. Thus, the value of a `loop$` expression depends on values of `apply$` calls. If F is a function symbol of arity n, then it is clearly desirable for both rewriting and evaluation to replace a call `(apply$ 'F (list` $t_1 \ldots t_n$`))` by `(`$F\ t_1 \ldots t_n$`)`. But for user-defined F, the equality of these two terms depends on the *warrant hypothesis* for F: the assertion `(apply$-warrant-`F`)`, where `apply$-warrant-`F is the warrant of F. The interested reader may find further background on `apply$`, including logical issues about the need for warrants, in our paper on `apply$` [3] and in discussions in :DOC `apply$`, :DOC warrant, and (regarding the so-called "local problem") :DOC introduction-to-apply$.

The example below illustrates warrants and warrant hypotheses. We begin as follows where as noted above, the `include-book` event is important for almost all work that involves `apply$` or `loop$`.

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

(defun$ square (n)
  (declare (xargs :guard (integerp n)))
  (* n n))
```

A call of `defun$` generates a corresponding `defun` event and introduces the warrant for `square`, which is the function symbol `apply$-warrant-square`, with the event `(defwarrant square)`. The following rewrite rule, named `apply$-square`, is the key property of the warrant hypothesis, which is forced here so that a proof may progress to the forcing round as described below.

```
(implies (force (apply$-warrant-square))
         (equal (apply$ 'SQUARE args)
                (square (car args))))
```

We may abbreviate (apply$-warrant-square) with the macro call, (warrant square). More generally, for a function symbol F, (warrant F) abbreviates (apply$-warrant-$F$). We see that the rewrite rule above replaces a call of apply$ on 'SQUARE by the corresponding call of square.

In summary: the warrant hypothesis for F justifies replacing (apply$ 'F (list $t_1 \ldots t_n$)) by (F $t_1 \ldots t_n$).

The following sequence of events, extending the include-book and defun$ forms displayed above, illustrates the role of warrant hypotheses.

```
; Return the list of squares of integers between the given bounds.
(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))) ; example evaluation

(thm (implies (warrant square)
              (equal (f2 3 5) '(9 16 25))))

(thm (implies (and (natp k1) (natp k2) (natp k3) (<= k1 k2) (<= k2 k3)
                   (warrant square))
              (member (* k2 k2) (f2 k1 k3))))
```

If the warrant hypothesis is omitted in the two calls of thm above, the proofs will progress to forcing rounds before failing. The forced goals make clear the need for a warrant hypothesis. For example, if the warrant hypothesis is removed before submitting the first thm form above, then the forcing round has the goal (APPLY$-WARRANT-SQUARE).[3]

Note that proofs succeed automatically for both thm forms above. They succeed without the initial include-book event and without any warrant hypotheses if we replace (square x) by (* x x) in the definition of f2; only user-defined functions have warrants (or require warrant hypotheses), not primitives like * (more precisely, binary-*; * is a macro).

4 Semantics: Translation to Logic

The introduction presented the example

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

and stated that it "essentially" translates to the following term.

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

In this section we explain precisely how the simplest, *plain* loop$ expressions such as this one are translated into loop$ scion calls. We then discuss the translation of more complex *fancy* loop$ expressions. We conclude with a note about relaxation of some restrictions for loop$ expressions in theorems.

[3] Missing warrants from otherwise provable theorems do not *necessarily* lead to forcing because the proof may fail before then.

4.1 Plain loops and `lambda$`

A *plain loop* is a `loop$` expression that has a single iteration variable — that is, a `FOR` clause but no `AS` clause — which is the only variable that occurs free in the `UNTIL` test, `WHEN` test, or loop body. ACL2 gives semantics to a plain loop by generating a call of the `loop$` scion corresponding to the loop operator, where the first argument is a *`lambda$` expression* whose single formal is the iteration variable and whose body is the loop body. A `lambda$` expression, in turn, represents a corresponding quoted lambda object whose body is a translated term. Finally, additional calls of `return-last` are inserted (see below).

Let's see how this works on the `loop$` expression in the following definition.

```
(defun sum-squares (lst)
  (loop$ for x in lst sum (* x x)))
```

This is a simple loop: it binds only the variable `x`, which is the only variable occurring free in the (translation of) the loop body. ACL2 first transforms this `loop$` expression to

```
(SUM$ (LAMBDA$ (X) (* X X))
      LST).
```

We see that the loop operator `SUM` generates a call of the corresponding `loop$` scion, `sum$`. The first argument of that call is the `lambda$` expression whose single formal is the iteration variable, `x`, and whose body is exactly the loop body. This `lambda$` expression, in turn, is transformed to a quoted lambda object by replacing the loop body `(* x x)` with its translation `(binary-* x x)`, and by declaring that the formals — in this case, the single formal, `x` — may be ignored. (Such a declaration permits loop bodies that do not mention all of the iteration variables.)

```
(SUM$ '(LAMBDA (X)
               (DECLARE (IGNORABLE X))
               (BINARY-* X X))
      LST)
```

But we're not done yet! The body of the `lambda$` expression is actually replaced by the translation of `(prog2$ '(LAMBDA$ (X) (* X X)) (* x x))` — the system uses that quoted `lambda$` expression for efficient execution of compiled lambda objects[4] — and the `loop$` expression similarly is wrapped with a call of `prog2$` that preserves the original `loop$` expression in the first argument, as discussed below in Subsection 6.2. Here is the translation of the body of `sum-squares`.

```
(RETURN-LAST 'PROGN
             '(LOOP$ FOR X IN LST SUM (* X X))
             (SUM$ '(LAMBDA (X)
                            (DECLARE (IGNORABLE X))
                            (RETURN-LAST 'PROGN
                                         '(LAMBDA$ (X) (* X X))
                                         (BINARY-* X X)))
                   LST))
```

Fortunately, one does not generally see such complexity during a proof. One reason is that the `return-last` calls are removed; see :DOC guard-holders. Another reason is that during a proof one

[4] See ACL2 source file `apply-raw.lisp` for details, in particular, the "Essay on the CL-Cache Implementation Details".

sees *untranslated terms* (see :DOC term), and the untranslation process typically restores the original `lambda$` expression, so that in this case one sees `(sum$ (lambda$ (x) (* x x)) lst)`.

We remark that a `lambda$` expression is only legal when, roughly speaking, it is ultimately used only as the first argument of `apply$` calls.[5]

4.2 Fancy loops

We explain translation of fancy loops using the following example. This definition contains a fancy loop both because the variables `m` and `n` are not bound by the `loop$` expression and because of the `AS` clause.

```
(defun g (m n lst1 lst2)
  (loop$ for x1 in lst1 as x2 in lst2 sum (* m n x1 x2)))
```

For our discussion of fancy loops we will focus on basic semantics, ignoring `return-last` calls and `ignorable` declarations discussed for plain loops above.

Before we show the (essential) translation of the fancy loop above, we introduce auxiliary functions, `loop$-as` and `sum$+`. `Loop$-as` is given a list `L` of n lists and returns the sequence of n-tuples containing corresponding elements from each list until the shortest list is exhausted. For example:

```
ACL2 !>(loop$-as '((1 2 3 4 A B C) (5 6 7 8)))
((1 5) (2 6) (3 7) (4 8))
ACL2 !>
```

We next consider the *fancy `loop$` scion* `sum$+`, which is called in the translation of the fancy loop above. Its first parameter, `fn`, is expected to be a function symbol or lambda object with two arguments. The first argument of `fn` is expected to be a list of *globals*: values of the variables in the loop body that are not iteration variables. The second argument of `fn` is expected to be a list of *locals*: the result of a `loop$-as` call, which (again) is a list whose n^{th} member lists the n^{th} iteration values of the iteration variables. Thus `sum$+` is defined much like `sum$` (here we ignore declarations as well as `mbe` and `fix` wrappers), but where the globals are passed when applying `fn` and the locals are wrapped into a list.

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

The fancy loop in the definition of `g`, above, has the following semantics (translation). We see that the first argument is indeed a function of globals and locals, where the globals is the list containing the values of formals `m` and `n` of the function `g` defined above and the locals come from the `loop$-as` call.

```
(SUM$+ (LAMBDA$ (LOOP$-GVARS LOOP$-IVARS)
                (DECLARE (XARGS :GUARD ...))
                (LET ((M (CAR LOOP$-GVARS))
                      (N (CAR (CDR LOOP$-GVARS)))
                      (X1 (CAR LOOP$-IVARS))
                      (X2 (CAR (CDR LOOP$-IVARS))))
                     (* M N X1 X2)))
       (LIST M N)
       (LOOP$-AS (LIST LST1 LST2)))
```

[5] To be precise, it must be in a position of *ilk* `:FN` [3].

We have seen that the `loop$` operator, `SUM`, produces a call of a `loop$` scion: `sum$` for a plain loop and `sum$+` for a fancy loop. More generally, a `loop$` scion arises from any use of `loop$`, and the scions introduced are all plain or all fancy depending on the particulars of the `loop$`. The plain ones are `sum$`, `collect$`, `always$`, `thereis$`, `append$`, `until$`, and `when$`, respectively, and their fancy counterparts are `sum$+`, `collect$+`, `always$+`, `thereis$+`, `append$+`, `until$+`, and `when$+`. The plain `loop$` scions may be summarized as follows, where the elements of the true-list `lst` are $e_1, \ldots, e_n$.

- `(sum$ fn lst)`: sums all `(apply$ fn (list` e_i`))`
- `(always$ fn lst)`: tests that all `(apply$ fn (list` e_i`))` are non-`nil`
- `(thereis$ fn lst)`: returns the first non-`nil` value of `(apply$ fn (list` e_i`))`, $i = 1, 2, \ldots, n$
- `(collect$ fn lst)`: conses together all `(apply$ fn (list` e_i`))`
- `(append$ fn lst)`: appends together all `(apply$ fn (list` e_i`))`
- `(until$ fn lst)`: lists all e_i, in order, until the first i such that `(apply$ fn (list` e_i`))` is non-`nil`
- `(when$ fn lst)`: lists those e_i, in order, such that `(apply$ fn (list` e_i`))` is non-`nil`

The fancy `loop$` scions are analogous. We have already seen the definition of `sum$`, and we have seen a similar definition of `sum$+` that accommodates the globals. Here is an analogous pair of definitions for `when$` and `when$+`.

```
(defun when$ (fn lst)
  (if (endp lst)
      nil
    (if (apply$ fn (list (car lst)))
        (cons (car lst) (when$ fn (cdr lst)))
      (when$ fn (cdr lst)))))

(defun when$+ (fn globals lst)
  (if (endp lst)
      nil
    (if (apply$ fn (list globals (car lst)))
        (cons (car lst) (when$+ fn globals (cdr lst)))
      (when$+ fn globals (cdr lst)))))
```

4.3 `Loop$` restrictions are eased in theorems

Conventional syntactic requirements are enforced for the loop body of a `loop$` expression that occurs either at the top level or in the body of a definition. In particular, the loop body should return a single, non-stobj value. As usual, such requirements are relaxed when the `loop$` expression occurs in a theorem. For example, the following form is legal, and in fact is admitted by ACL2, even though the `loop$` expression is illegal both at the top level and in any function body.

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

5 Reasoning with `Loop$`

We recall the definition of `sum-squares` above and state a simple theorem about that function.

```
(defun sum-squares (lst)
  (loop$ for x in lst sum (* x x)))

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

The proof fails, but the experienced ACL2 user quickly completes the proof by noticing terms of the form `(sum$ FN (revappend X Y))` in the checkpoints and then proving the following lemma.

```
(defthm sum$-revappend
  (equal (sum$ fn (revappend x y))
         (+ (sum$ fn x) (sum$ fn y))))
```

Of course, one could instead define `sum-squares` recursively and go through a similar process to find and prove a suitable lemma about `(sum-squares (revappend x y))`, and then prove the `thm`. But by using `loop$`, one needn't prove such a lemma more than once. For example, suppose we define:

```
(defun sum-cubes (lst)
  (loop$ for x in lst sum (* x x x)))
```

Then the following theorem is proved automatically by applying the lemma above, `sum$-revappend`.

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

If instead `sum-cubes` were defined by recursion, then one would need first to prove a new lemma about `(sum-cubes (revappend x y))`.

We have seen that reasoning about `loop$` expressions generally reduces to reasoning about `loop$` scions. The community book `projects/apply/loop`, included in the book `projects/apply/top` that is typically included when reasoning about `apply$` or `loop$`, has helpful lemmas to support automation of reasoning about `loop$`. Perhaps over time, the lemma `sum$-revappend` (above) and other useful lemmas will be included in lemma libraries for reasoning about `loop$`. Also see community book `projects/apply/report.lisp` for examples of reasoning with (close analogues of) `loop$` scions.

Let's look at one more proof example.

```
(thm (equal (sum-squares '(1 2 3 4)) 30))
```

This proves automatically using these rules.

```
Rules: ((:EXECUTABLE-COUNTERPART EQUAL)
        (:EXECUTABLE-COUNTERPART SUM-SQUARES))
```

Thus, evaluation involving `loop$` expressions (in this case, in the body of the definition of `sum-squares`) can take place during proofs. This simple example does not require warrants, but those are handled with evaluation as well; see Section 6.3.

The community book `projects/apply/mempos.lisp` may be helpful for reasoning about `loop$` expressions that have more than one iteration variable or an `UNTIL` clause.

6 Evaluation

We have seen that `loop$` calls are translated into logic by generating calls of `loop$` scions. Thus, evaluation of `loop$` calls can take place by evaluating those translations. Indeed, that is what happens when `loop$` expressions occur at the top level, as we discuss in the first subsection below. However, when `loop$` expressions are in guard-verified code, their evaluation generally reduces to efficient evaluation of corresponding Common Lisp `loop` expressions, as we discuss in the second subsection below. We next discuss a key exception: during proofs, `loop$` scions are always used for evaluation, with a twist: there is accounting for warrants. Finally we discuss performance.

6.1 Evaluation in the top-level loop

We first consider evaluation of `loop$` calls taking place directly in the top-level loop (rather than within a function body). Every top-level form is evaluated by first translating into logic. In particular, evaluation of a `loop$` call in the top-level loop takes place by translating into logic, that is, into calls of `loop$` scions as described in Section 4. The following example shows this behavior in action.

```
ACL2 !>(trace$ collect$)
 ((COLLECT$))
ACL2 !>(loop$ for i from 1 to 5 collect (* i i))
1> (ACL2_*1*_ACL2::COLLECT$
        (LAMBDA (I)
                (DECLARE (IGNORABLE I))
                (RETURN-LAST 'PROGN
                             '(LAMBDA$ (I) (* I I))
                             (BINARY-* I I)))
        (1 2 3 4 5))
  2> (COLLECT$ (LAMBDA (I)
                       (DECLARE (IGNORABLE I))
                       (RETURN-LAST 'PROGN
                                    '(LAMBDA$ (I) (* I I))
                                    (BINARY-* I I)))
               (1 2 3 4 5))
  <2 (COLLECT$ (1 4 9 16 25))
<1 (ACL2_*1*_ACL2::COLLECT$ (1 4 9 16 25))
(1 4 9 16 25)
ACL2 !>
```

Top-level evaluation treats all warrants as being true at the top level. In short, the reason is that at the top level, attachments may be used (see :DOC defattach), and all warrants have true attachments. Thus for example, if we introduce the function symbol `square` as defined in Section 3 and we replace the body of the `loop$` expression above by `(square i)`, then we get the same evaluation result.

6.2 Guards and fast Common Lisp evaluation for function calls

Recall that guard verification permits ACL2 to execute with Common Lisp definitions. When this is done, `loop$` expressions are computed by evaluating corresponding Common lisp `loop` expressions. For example, recall this definition from Section 3.

```
(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)))
```

If we evaluate `(f2 1 5)`, we get the same result as in the preceding example: `(1 4 9 16 25)`. However, if we first trace `collect$`, as before, then this time we see no calls of `collect$`. The reason is clear from the single-step Common Lisp macroexpansion of the `loop$` expression displayed below, which shows that when attachments are allowed (i.e., `*aokp*` is true), as is the case when evaluating directly in the top-level loop, then the corresponding `loop` expression is evaluated. (We'll discuss the other case when we discuss evaluation during proofs.)

```
(COND (*AOKP* (LOOP FOR I OF-TYPE INTEGER FROM LOWER TO UPPER
                    COLLECT (SQUARE I)))
      (T ...))
```

Guard verification is necessary before a `loop$` call will be evaluated using `loop` in Common Lisp. Indeed, if we instead define `f2` by adding the `xargs` declaration `:verify-guards nil`, then evaluation of `(f2 1 5)` will show `collect$` calls in a trace as displayed in the preceding subsection. The natural question is then: What are the guard proof obligations generated for a `loop$` expression?

Let us begin by considering our `f2` example. In that case, ACL2 reports the following "non-trivial part of the guard conjecture".

```
(IMPLIES (AND (INTEGERP LOWER)
              (INTEGERP UPPER)
              (APPLY$-WARRANT-SQUARE)
              (MEMBER-EQUAL NEWV (FROM-TO-BY LOWER UPPER 1)))
         (INTEGERP NEWV))
```

This formula states that if `newv` is a member of the indicated `from-to-by` expression, then `newv` is an integer. It is generated by the expression "`i of-type integer`" in the `loop$` expression, which is necessary for guard verification of the loop body, since the guard of `(square n)` is `(integerp n)`.

The formula displayed above illustrates the first of the following three classes of guard proof obligations arising from `loop$` expressions. The second one below is a rather obvious requirement. The third arises because of Common Lisp quirks: all tails may be checked in the `ON` case, including `nil`, and the type-checks for indices from `i` to `j` by `k` may include one step past the point where iteration stops.

- *Special Conjecture (a)*: Every member of the target list satisfies the guard of the function object.
- *Special Conjecture (b)*: On every member of the target list, the function object produces a result of the right type: a number if the loop operator is `SUM` and a true list if the operator is `APPEND`.
- *Special Conjectures (c)*: For `(loop$ ... ON ...)`, `nil` satisfies the type expression (if any). For a `(FROM-TO-BY i j k)` target list: `i`, `j`, `k`, and `(+ i (* k (floor (- j i) k)) k)` each satisfy the type expression (if any).

Warrant hypotheses can be relevant to guard verification, although the user is generally spared from thinking about that. Consider the following variant of `f2` above, where `SUM` replaces `COLLECT`.

```
(defun sum-squares-2 (lower upper)
  (declare (xargs :guard (and (integerp lower) (integerp upper))))
  (loop$ for i of-type integer from lower to upper
         sum (square i)))
```

The hypothesis `(warrant square)` does not appear in the `:guard`, but it is added automatically in the generated guard conjecture. This addition is necessary; the proof fails without it. This addition is also justified: the Special Conjectures are generated only to avoid guard violations when evaluating the Common Lisp `loop` expression that corresponds to the `loop$` expression, and that only happens when attachments are permitted (see the use of `*aokp*` in the Common Lisp definition of `loop$` above). But when attachments are permitted then all warrant hypotheses are true, which justifies them as hypotheses.

We conclude this subsection by noting that instead of using the `of-type` construct, one can supply a `:guard` as indicated in Section 2. The `loop$` expression in the body of `f2` could instead be written as follows, giving an equivalent definition.

```
(loop$ for i from lower to upper
       sum :guard (integerp i) (square i))
```

The `of-type` construct may provide for more efficient Common Lisp code, since the `:guard` construct is ignored by the compiler. On the other hand, the `:guard` construct is more flexible, since it can reference more than one variable.

6.3 Evaluation during proofs

An example in Section 5 shows that evaluation of `loop$` expressions can take place during proofs. That example did not require any warrant hypotheses, but in general they might be required.

As noted above, when attachments are allowed then all warrant hypotheses hold, so ACL2 evaluates `(apply$ 'F ...)` by simply calling F on the supplied arguments. Unfortunately, attachments are not allowed during proofs.

However, a major design goal for ACL2 is fast evaluation during proofs. That includes calls of `apply$`, calls of `loop$`, and calls of functions that lead to calls of `apply$` or `loop$`. Thus, a challenge is for ACL2 to support evaluation of `(apply$ 'F ...)` by simply calling F, as discussed above for the case that attachments are allowed.

ACL2 meets this challenge, as clearly illustrated by its ability to prove the following theorem using evaluation of a `loop$` expression (where `sum-squares-2` is defined as in the preceding subsection).

```
(thm (implies (warrant square)
              (equal (sum-squares-2 1 4) 30)))
```

Let us inspect the summary.

```
Rules: ((:DEFINITION SUM-SQUARES-2)
        (:EXECUTABLE-COUNTERPART EQUAL)
        (:EXECUTABLE-COUNTERPART FROM-TO-BY)
        (:EXECUTABLE-COUNTERPART SUM$)
        (:REWRITE APPLY$-SQUARE))
```

The use of the rule `(:rewrite apply$-square)` is expected, as discussed in Section 5. The use of the definition rule for `sum-squares-2` (as opposed to evaluation of that function's call) is perhaps surprising: it takes place early, in the *preprocess* step of the waterfall, where evaluation of `loop$` expressions that require warrants is not supported. The simplifier then evaluates the body of `sum-squares-2` with respect to the binding alist `((upper . '4) (lower . '1))`. The hints shown below avoid the definition rule but the proof still succeeds by running the executable counterpart of `sum-squares-2`, which actually checks that the necessary warrant is available as a hypothesis.

```
(thm (implies (warrant square)
              (equal (sum-squares-2 1 4) 30))
     :hints (("Goal" :in-theory (disable sum-squares-2))))
```

Deleting the hypothesis causes the proof to fail after the missing warrant is forced by proof-time evaluation.

For more examples involving evaluation of `loop$` expressions during proofs, see the community books `system/tests/apply-in-proofs.lisp` and `system/tests/loop-tests.lisp`.

We conclude our discussion of evaluation during proofs by remarking on the implementation and its impact on performance. In the preceding subsection we discussed the macroexpansion of `loop$` expressions in Common Lisp when attachments are allowed. We now consider the other case, which invokes the usual translation to a call of a `loop$` scion. The expansion is as follows for the `loop$` expression in the definition of `f2`.

```
(COND (*AOKP* ; Attachments are allowed.
       (LOOP FOR I OF-TYPE INTEGER FROM LOWER TO UPPER
             COLLECT (SQUARE I)))
      (T (COLLECT$ '(LAMBDA (I)
                            (DECLARE (TYPE INTEGER I)
                                     (XARGS :GUARD (INTEGERP I) :SPLIT-TYPES T)
                                     (IGNORABLE I))
                            (RETURN-LAST 'PROGN
                                         '(LAMBDA$ (I)
                                                   (DECLARE (TYPE INTEGER I))
                                                   (SQUARE I))
                                         (SQUARE I)))
                    (FROM-TO-BY LOWER UPPER '1))))
```

The `loop$` scion call is generated by finding a term associated with the `loop$` expression in the world.

```
ACL2 !>(cdr (assoc-equal '(LOOP$ FOR I OF-TYPE INTEGER
                                 FROM LOWER TO UPPER COLLECT (SQUARE I))
                         (global-val 'LOOP$-ALIST (w state))))
(LOOP$-ALIST-ENTRY
     (COLLECT$ '(LAMBDA (I)
                        (DECLARE (TYPE INTEGER I)
                                 (XARGS :GUARD (INTEGERP I)
                                        :SPLIT-TYPES T)
                                 (IGNORABLE I))
                        (RETURN-LAST 'PROGN
                                     '(LAMBDA$ (I)
                                               (DECLARE (TYPE INTEGER I))
                                               (SQUARE I))
                                     (SQUARE I)))
               (FROM-TO-BY LOWER UPPER '1)))
ACL2 !>
```

ACL2 updates the `loop$-alist` structure at the conclusion of a `defun` event (here, for `f2`), by collecting all *tagged loops* `(RETURN-LAST 'PROGN '(LOOP$ ...) e)` and associating each such `(LOOP$ ...)`

with its corresponding translation, *e*. Well, almost: the problem is that *e* might not be executable, in particular because mv-let calls are eliminated by translation using mv-nth calls. So the implementation (with function convert-tagged-loop$s-to-pairs) converts the loop$ translation to executable code (with function logic-code-to-runnable-code), notably by inserting calls of mv-list.

A more complete answer also takes into account early loading of compiled files and removes ACL2-specific guard information. Here is the definition of loop$ (from the ACL2 sources) in Common Lisp.

```
(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 "Unable to translate loop$ (defun given directly ~
                            to raw Lisp?)."))))))
```

The following two issues are raised by this method of evaluating loop$ expressions during proofs.

- Section 3 noted the necessity of warrant hypotheses because attachments are disallowed during proofs. Specifically, as we have seen, calls of loop$ scions may ultimately lead to calls of apply$ on user-defined functions, which is where warrant hypotheses are required. The implementation permits evaluation of such apply$ calls during rewriting but forces the necessary warrant hypotheses that are not known in the current rewriting context (the so-called "type-alist").
- Evaluation of loop$ expressions can be considerably slower during proofs than it is at the top level, because of the use of loop$ scions and apply$ rather than evaluation of a Common Lisp loop expression. We considered allowing this faster evaluation during proofs, but that would require checking (or forcing) the warrant hypothesis of *every* user-defined function in the loop body (and also the WHEN and UNTIL clauses, if present), even for functions that are rarely called when evaluating the loop body, such as in the error branch of an IF expression.

6.4 Performance

The results reported in the comments below were produced using an ACL2 version (git hash 5eb79e7697) built on CCL on April 4, 2018, running on a 3.5 GHz 4-core Intel(R) Xeon(R) with Hyper-Threading. Times in seconds are realtime; also shown are bytes allocated. The key take-away is the comparison between (c) and (d): Evaluation in the ACL2 loop, including the integer-listp call in the guard, is very close in time to the corresponding evaluation directly in Common Lisp.

```
(include-book "projects/apply/top" :dir :system)
(defun$ double (n)
  (declare (xargs :guard (integerp n)))
  (+ n n))
(defun sum-doubles (lst)
  (declare (xargs :guard (and (integer-listp lst)
                              (warrant double))
                  :verify-guards nil))
```

```
  (loop$ for x of-type integer in lst sum (double x)))
(make-event `(defconst *m* ',(loop$ for i from 1 to 10000000 collect i)))
; (a) ACL2 top-level loop$ call [0.98 seconds, 160,038,272 bytes]:
(time$ (loop$ for i of-type integer in *m* sum (double i)))
; (b) ACL2 top-level non-guard-verified function call [0.89 seconds, 160M bytes]
(time$ (sum-doubles *m*))
(verify-guards sum-doubles)
; (c) ACL2 top-level guard-verified function call [0.14 seconds, 16 bytes]
(time$ (sum-doubles *m*))
(value :q)
; (d) Common Lisp guard and function call [0.13 seconds, 0 bytes]:
(time$ (and (integer-listp *m*) (sum-doubles *m*)))
; (e) Common Lisp function call [0.09 seconds, 0 bytes]:
(time$ (sum-doubles *m*))
; (f) Common Lisp loop call [0.08 seconds, 0 bytes]:
(time$ (loop for i of-type integer in *m* sum (double i)))
```

7 Limitations, Future Work, and Conclusion

We know of several limitations that may be addressed in future work. For more information search for "Possible Future Work on Loop$" in ACL2 source file `translate.lisp`.

- Reliance on `apply$` and scions means that quoted `lambda` objects may appear in formulas that, when entered by the user, had no such objects in them. This can cause trouble because two obviously equivalent quoted `lambda` objects may be distinct objects. This arises especially when a theorem statement may involve a `loop$` iteration variable with a different name than used in some function definition. E.g., `'(lambda (x) (sq x))` is a different object than `'(lambda (y) (sq y))` even though they are obviously functionally equivalent. `Declare` forms in such objects can also render them unnecessarily distinct. While the functional equivalence of `lambda` objects is, of course, undecidable, trivial cases like these ought to be caught by the prover but are not. Furthermore, because they are so obvious the user may overlook the differences! Until these issues are smoothed out in the prover, checkpoints involving `lambda` objects must be read with unusual care!
- `Apply$` only works on logic mode functions so it is an error if a program mode function appears in a `loop$` body or a `WHEN` or `UNTIL` clause.
- `Apply$` does not work on state- or stobj-using functions, hence neither does `loop$`. This may be very difficult to change.
- At the time this paper was submitted, recursion within a `loop$` body was not supported and so it is not described here. But since then we have added support for it; see :DOC loop$-recursion.
- As noted at the end of Subsection 6.3, evaluation of a `loop$` expression uses `loop$` scions instead of the more efficient Common Lisp `loop` when either during a proof or directly in the top-level loop (rather than in a function body). (Of course, such evaluation can be expected to be much faster than term rewriting.) For the top-level case, perhaps ACL2 should report an error and insist on the use of the utility `top-level`, as illustrated by the following example.

  ```
  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 !>
```

- Common Lisp `loop` supports more general forms than `loop$`, some of which are feasible to support in `loop$`. Here are a couple of examples.

```
? (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)
?
```

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.

Acknowledgments

This material is based upon work supported in part by DARPA under Contract No. FA8650-17-1-7704. We are also grateful for support of this work by ForrestHunt, Inc., and for helpful reviewer feedback.

References

[1] R. Boyer & J S. Moore (1988): *The Addition of Bounded Quantification and Partial Functions to a Computational Logic and Its Theorem Prover*. Journal of Automated Reasoning 4(2), pp. 117–172, doi:10.1007/BF00244392. See `http://www.cs.utexas.edu/users/moore/publications/quant.pdf`.

[2] R. S. Boyer & J S. Moore (1997): *A Computational Logic Handbook, Second Edition*. Academic Press, New York.

[3] M. Kaufmann & J S. Moore (2018): *Limited Second-Order Functionality in a First-Order Setting*. Journal of Automated Reasoning, doi:10.1007/s10817-018-09505-9. Available at `http://www.cs.utexas.edu/users/kaufmann/papers/apply/`.

[4] M. Kaufmann, J S. Moore & The ACL2 Community (2020): *The Combined ACL2+Books User's Manual*. `http://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index.html`.

[5] K. Kunen (1998): *Nonconstructive Computational Mathematics*. J. Autom. Reason. 21(1), pp. 69–97, doi:10.1023/A:1005888712422.

[6] J S. Moore (1975): *Introducing Iteration into the Pure LISP Theorem-Prover*. IEEE Transactions on Software Engineering 1(3), pp. 328–338, doi:10.1109/TSE.1975.6312857. See `http://www.cs.utexas.edu/users/moore/publications/parc-csl-tr-74-3.pdf`.