Extended Abstract: Iteration in ACL2, WITH .. DO

Matt Kaufmann
J Moore

UT Austin (retired)

May 26, 2022

TALK OVERVIEW

This talk will primarily consist of a **demo**.

TALK OVERVIEW

This talk will primarily consist of a **demo**.

For more about `DO loop$` expressions, see the documentation and supporting materials:

- :DOC LOOP$.
- :DOC DO-LOOP$.
- books/workshops/2022/kaufmann-moore/, especially:
  - The input file, books/workshops/2022/kaufmann-moore/do-loops-input.lsp; and
  - The output (log) file, books/workshops/2022/kaufmann-moore/do-loops-log.txt.

TODAY

Goal for today: Present the basics of how to use `DO loop$` expressions, but also exhibit their power:

TODAY

Goal for today: Present the basics of how to use `DO loop$` expressions, but also exhibit their power:

- Support for imperative programming, stobjs, multiple-value return, :program mode, and :logic mode

TODAY

Goal for today: Present the basics of how to use `DO loop$` expressions, but also exhibit their power:

- Support for imperative programming, stobjs, multiple-value return, :program mode, and :logic mode

Basic form:

```
(loop$ WITH ... DO ...)
```

TODAY

Goal for today: Present the basics of how to use `DO loop$` expressions, but also exhibit their power:

- Support for imperative programming, stobjs, multiple-value return, :program mode, and :logic mode

Basic form:

```
(loop$ WITH ... DO ...)
```

(It is legal, but unusual, to omit the `WITH` part.)

TODAY

Goal for today: Present the basics of how to use `DO loop$` expressions, but also exhibit their power:

- Support for imperative programming, stobjs, multiple-value return, :program mode, and :logic mode

Basic form:

```
(loop$ WITH ... DO ...)
```

(It is legal, but unusual, to omit the `WITH` part.)

Compare with `FOR loop$` (see 2020 ACL2 Workshop):

```
(loop$ FOR ...)
```

TODAY

Goal for today: Present the basics of how to use `DO loop$` expressions, but also exhibit their power:

- Support for imperative programming, stobjs, multiple-value return, :program mode, and :logic mode

Basic form:

```
(loop$ WITH ... DO ...)
```

(It is legal, but unusual, to omit the `WITH` part.)

Compare with `FOR loop$` (see 2020 ACL2 Workshop):

```
(loop$ FOR ...)
```

Evaluation strips out certain keywords and calls Common Lisp `loop`, for guard-verified and :program-mode evaluation.

** DEMO **

REMARKS

The only proof we've shown is a guard proof. Support for proofs involving `DO loop$` expressions is under further development.

REMARKS

The only proof we've shown is a guard proof. Support for proofs involving `DO loop$` expressions is under further development.

Such proof debugging depends on translation to calls of the recursive function, `DO$`. See the paper and :DOC do-loop$.

REMARKS

The only proof we've shown is a guard proof. Support for proofs involving `DO loop$` expressions is under further development.

Such proof debugging depends on translation to calls of the recursive function, `DO$`. See the paper and :DOC do-loop$.

Note that `OF-TYPE` informs the Common Lisp compiler but `:GUARD` does not, much like types vs. guards for a defun.

A Bigger Example

The following example is from a parser I (Matt) wrote for the project reported in the workshop paper by Hunt, Ramanathan, and Moore: "VWSIM: A Circuit Simulator".

A Bigger Example

The following example is from a parser I (Matt) wrote for the project reported in the workshop paper by Hunt, Ramanathan, and Moore: "VWSIM: A Circuit Simulator".

I don't expect it to make sense in the talk; in fact I've removed comments (and done other formatting) to fit it on the next three slides.

A Bigger Example

The following example is from a parser I (Matt) wrote for the project reported in the workshop paper by Hunt, Ramanathan, and Moore: "VWSIM: A Circuit Simulator".

I don't expect it to make sense in the talk; in fact I've removed comments (and done other formatting) to fit it on the next three slides.

The point here is just to illustrate that `DO loop$` expressions can be used in "real" code. **Note that the following function is in :program mode.**

A Bigger Example

The following example is from a parser I (Matt) wrote for the project reported in the workshop paper by Hunt, Ramanathan, and Moore: "VWSIM: A Circuit Simulator".

I don't expect it to make sense in the talk; in fact I've removed comments (and done other formatting) to fit it on the next three slides.

The point here is just to illustrate that `DO loop$` expressions can be used in "real" code. **Note that the following function is in :program mode.**

Imperative "keywords" are capitalized for emphasis.

```
(defun next-line-etc (pos str lineno end)
  (LOOP$
   WITH newline-pos WITH start-pos = pos
   WITH next-pos = pos WITH next-lineno = lineno
   WITH line = "" WITH comment-p = nil
   DO
   :VALUES (nil nil nil)
   :MEASURE (nfix (- end next-pos)) ; not needed here
   (cond
    ((= next-pos end)
     (RETURN (mv line nil nil)))
    ((and comment-p
          (not (member (char str next-pos) '(#\+ #\*))))
     (RETURN (mv line next-pos next-lineno)))
    (t
     (PROGN
       (SETQ comment-p (eql (char str next-pos) #\*))
       (SETQ newline-pos
             (search *newline-string* str :start2 next-pos))
       (SETQ start-pos
             (scan-past-whitespace str next-pos
                                   (or newline-pos end)))
```

```
(cond
 (newline-pos
  (PROGN
    (SETQ next-pos (1+ newline-pos))
    (SETQ next-lineno (1+ next-lineno))
    (cond
     (comment-p (PROGN)) ; just keep going
     (t (PROGN
          (SETQ line
                (let ((s1 (subseq str
                                  start-pos
                                  newline-pos)))
                  (if (equal line "")
                      s1
                    (concatenate 'string line s1))))
          (cond
           ((= next-pos end)
            (RETURN (mv line nil nil)))
           ((eql (char str next-pos) #\+)
            (SETQ next-pos (1+ next-pos)))
           ((eql (char str next-pos) #\*) (PROGN))
           (t (RETURN
               (mv line next-pos next-lineno)))))))))))
```

```
(t (RETURN
    (mv (if comment-p
            line
          (let ((s1 (subseq str start-pos end)))
            (if line
                (concatenate 'string line s1)
              s1)))
        nil
        nil)))))))))
```

Thank you. And we thank ForrestHunt, Inc. for the support.