

Doplus

the high-level, Lispy, extensible iteration construct

Alessio Stalla
ManyDesigns s.r.l.
alessiostalla@gmail.com

ABSTRACT

In this paper, we briefly present a novel iteration macro for Common Lisp.

Categories and Subject Descriptors

D.3.3 [Programming Languages]: Language Constructs and Features—*control structures*.

General Terms

Languages, Experimentation.

Keywords

Lisp, macro, iteration.

1. INTRODUCTION

There are three general-purpose, widely used iteration constructs in Common Lisp. Of these, two are part of the Common Lisp standard itself (`do/do*` and `loop`) and one is a third-party library (`Iterate`).

The author was unsatisfied with all the three of them. `do`, a glorified C `for`, is too low level. `loop`, with its PASCAL-like embedded DSL, does not mix well with the rest of the language (`:collect` can't be used in the body of a `let` form, for example), it's not SLIME-friendly, and it's not extensible in a portable, easy way. `Iterate` is mostly fine, except for a not-so-tiny detail: it is implemented using a code walker that has freedom to move pieces of user code around. This breaks `macrolet` and creates some potentially unexpected effects when a form lexically following another is moved back before the preceding one.

So, in pure Lisp hacker spirit, the author rolled his own and began marketing it all around the world as the best thing since sliced bread. This very paper is a prime example of the aforementioned phenomenon.

2. DOPLUS

`do+` (`doplus`) is an iteration macro for Common Lisp. It is conceptually a high-level `do` in that it retains from `do` the separation of iteration control forms from the body of the loop itself. In other words, a `doplus` form has the following shape:

```
(do+ (clause*) form*)
```

We will not go into unnecessary details here because the basic features of `doplus` are the same as `loop` and `Iterate`. Clauses comprise the usual paraphernalia: numeric iteration, walking over sequences, accumulators, generators, etc., as well as user-defined ones.

Clauses are regular macros: no special processing is done on their syntax by `do+`, and users can define their own using natural syntax. User-defined clauses are indistinguishable from built-in ones. Most clauses take positional and/or keyword arguments that control their behaviour; a few take other clauses as arguments: for example, `for`'s syntax is:

```
(for variable-or-lambda-list iteration-specification)
```

where `iteration-specification` is actually a clause that can interact with the environment established by `for`. Certain clauses are exposed through more than one name, to help read a `doplus` form like an English sentence; for example `for - finding` or `until - stop-when`.

More technical details about clauses are discussed towards the end of this paper.

Inside the body, a few macros can be used to control iteration: `update`, `terminate` and `skip`, which respectively manually update a generator, terminate the loop, or skip to the next iteration.

In addition to all that, `doplus` has some unique features. Some of them are detailed below.

2.1 Atomic updates

In `doplus`, contrarily to other loop macros, variables are updated atomically, i.e. each clause that is executed at the beginning or at the end of an iteration sees consistent values for the variables, independently of the order of the clauses in the head. An example can make it clearer. Consider the following two iterate forms:

```
(iter
  (for k :in '(a b c d e))
  (for x :in-vector #(1 2 3 4))
  (finally (return (list x k))))
=> (4 E)

(iter
  (for x :in-vector #(1 2 3 4))
  (for k :in '(a b c d e))
  (finally (return (list x k))))
=> (4 D)
```

They only change in the order of the two `for` clauses, yet their return values differ. `loop` behaves the same. This doesn't happen with `doplus`; the following two forms return the same value:

```
(do+ ((for x (across #(1 2 3 4) :index index))
      (for k (in '(a b c d e)))
      (returning (list x k :index index))))
=> (4 D :INDEX 3)

(do+ ((for k (in '(a b c d e)))
      (for x (across #(1 2 3 4) :index index))
      (returning (list x k :index index))))
=> (4 D :INDEX 3)
```

That happens because as soon as the index goes out of the vector's bounds, the loop is terminated and any updates remain confined in the atomic section, without being visible to the other forms, such as those computing the return values.

Atomic updates can be disabled for a given invocation of the `do+` macro (for example, for performance reasons) by using a special clause.

2.2 Iteration over arbitrary sequences

The built-in clause `in` can iterate over arbitrary sequences - lists, vectors, as well as user-defined ones in implementations that support extensible sequences [1]. On those implementations¹, the native sequence iterator facility is used; on the others, a port of SBCL's iterator implementation is used instead.

There are also specialized clauses for lists and vectors that, besides being slightly more efficient, provide additional features that do not make sense for arbitrary sequence types, such as the ability to bind a loop variable to the successive CDRs of a list, or to the index of a vector, which is being iterated over.

¹At the time of writing, the only CL implementations known to the author and supporting the extensible sequences protocol are ABCL and SBCL.

2.3 Flexible collectors

In `doplus`, collection or accumulation is not fixed to a few built-in cases. Accumulators are defined by ad-hoc clauses with sensible defaults. Internally, an accumulator consists of a variable, bound during the lifetime of the loop, and a function of two arguments: the current value of the variable and the accumulator. That function is used to calculate the next value of the accumulator at the end of every iteration. However, for convenience, built-in clauses that define an accumulator also allow to specify an init form, whose value will be the initial value of the accumulation variable, and a result processor, that is a function to be applied to the accumulator at the end of the loop.

The canonical example, corresponding to `:collect ... :to x` in `loop`, is represented by an accumulation clause with `x` as the variable, `cons` as the function, a `nil` init form, and `nreverse` as the result processor. As a clause in a `doplus` form, it would appear as `(collecting-into x)`, since `cons` and `nreverse` are defaults for `collecting-into`.

Inside the body of a loop, the `collect` macro can be used to collect a value into an accumulation: `(collect value :into acc)`. `:into` is optional; if not provided, a default accumulator is used, using the aforementioned defaults. This is a more or less random example:

```
(do+ ((for x (in '(4 5 6 7)))
      (for y (from 3 :to 10))
      (for z (being (+ x y)))
      (stop-when (> z 11)))
      (if (oddp x)
          (collect (list x z))
          (collect (list y z)))))
```

2.4 Nested loops

Nested `doplus` forms can interact in certain ways with outer `doplus` forms. In Iterate, it is possible for an inner loop to execute a piece of code as if it were part of an outer loop[2]. `Doplus` does not employ a code walker and thus limits these kinds of interactions to some selected cases. In particular, it is possible to refer by name to an outer accumulator or generator, and, if an outer loop is given a name, the `skip` and `terminate` macros can refer to that name and behave accordingly. For example:

```
(do+ ((for x (in (list 1 2 3)))
      (accumulating-to result)
      (stop-when (> (length result) 10))
      (returning result))
      (do+ ((for y (in (list 'a 'b)))
            (collect (list x y) :into result))))

=> ((1 A) (1 B) (2 A) (2 B) (3 A) (3 B)
    (4 A) (4 B) (5 A) (5 B) (6 A) (6 B))

(do+ (outer-loop ;;<-- name of the loop
      (for x (in (list 1 2 3))))
      (print x)
      (do+ ((for k (to 3))
            (when (> (+ x k) 5)
              (terminate outer-loop)))))
```

3. EXTENSIONS AND IMPLEMENTATION NOTES

Extending `doplus` amounts to writing regular Lisp macros, that in most cases expand to a list of clauses as the user would have written them in the head of a `doplus` form. For example, the code below is adapted from the definition a clause built in `doplus`:

```
(defclause in-vector
  (vector &key (index (gensym "INDEX")))
  "Loops across a vector."
  (let ((tmp-var (gensym "VECTOR")))
    `((with (,tmp-var ,vector) (,index 0))
      (declaring
        (type (integer
              0
              ,(1- array-total-size-limit))
              ,index))
      (for ,index
          (from 0 :to (1- (length ,tmp-var))
                :by +1))
      (for ,*iteration-variable*
          (being (aref ,tmp-var ,index))))))
```

Here we can observe the only two peculiarities one can encounter when writing `doplus` extensions. First, the variable `*iteration-variable*`, which is bound by the `for` macro to the variable (or lambda list) provided by the user as the first argument to `for`. Second, the `defclause` macro, which is equivalent to `defmacro`, except that it also records the macro name as a known clause for documentation purposes.

You can also observe how clauses (built-in or user-defined) can instruct `do+` to insert declarations in the loop body.

Regarding the implementation, there is a notable aspect, in our opinion, in the use of macros to implement iteration clauses. These macros have the peculiarity that they do not return Lisp code - rather, they return `doplus` "code", that is, instances of structures that instruct the `doplus` macro on how to generate code². Effectively, the head of a `doplus` clause is written in a Lisp-based DSL that adopts the CL macro system for its own purposes. `macroexpand` is thus used as a kind of compile-time `funcall`.

3.1 Further development

The author considers `doplus` to be fairly complete; however, there is room for improvement. Certain kinds of iterations are not built in, for example sequence traversal always terminates the loop as soon as the sequence ends, while sometimes it is desirable to continue until other termination conditions are met, simply stopping to update the variable. Also, a few forms support destructuring using destructuring-bind; certain users have expressed preference for third-party destructuring macros, and a pluggable mechanism for adding them could be designed. Finally, performance has not been investigated much; on some simple loops, with atomic updates enabled and no declarations, `doplus` was found to be roughly twice as slow as `Iterate`, but further, more scientific analysis is needed.

3.2 Obtaining `doplus`

The `doplus` project is hosted at <http://code.google.com/p/tapulli/>. `Doplus` is also available through `Quicklisp`.

4. REFERENCES

- [1] C. Rhodes. 2007. *User-extensible sequences in Common Lisp*. ILC 2007 Proceedings, <http://doc.gold.ac.uk/~mas01cr/papers/ilc2007/sequences-20070301.pdf>
- [2] J. Amsterdam, 1989 and L. Oliveira, 2006. *The Iterate Manual*. <http://common-lisp.net/project/iterate/doc/Named-Blocks.html>

²Although technically structures are valid Lisp forms, since they evaluate to themselves, we emphasize the fact that `doplus` clause structures are used as the building blocks of a mini-language understood by the `doplus` macro.