Generalized comprehensions for Common Lisp

Sven-Olof Nyström
Department of Information Technology,
Uppsala University, Sweden
svenolof@csd.uu.se

ABSTRACT

Several functional programming languages have a feature called *list comprehensions*. This paper presents an implementation of list comprehensions for Common Lisp. It has been extended to match the expressiveness of Common Lisp's loop facility. For example, it allows traversal and construction of hash tables, arrays and vectors, general iteration, parallel iteration over two or more sequences, and the ability to terminate a collection when a condition are met. It is also possible to extend it to handle the traversal and collection in other data structures.

1. INTRODUCTION

Several programming languages (for example Haskell, Erlang and Python) have a feature called *list comprehensions* [5, 6] to facilitate the manipulation of lists. List comprehensions are inspired by a notation sometimes used in mathematics which allows expressions such as

$$\{x * x \mid x \in S, x \text{ odd}, x < 5\}$$

For example, in Erlang, the following expression

```
[X*X || X <- L, X rem 2 == 1, X < 5].
```

traverses a list `L` and returns the square of those elements that are odd and less than 5.

Common Lisp already has a powerful construct for expressing iteration, the loop facility [4, Chapter 26]. It resembles list comprehensions in that it supports list traversals and collecting results in a list. It is in some sense more powerful as it also allows (for example) iterating over ranges of integers and collecting the result as a sum.

This paper presents an implementation of list comprehensions in Common Lisp. It has been extended to match the expressiveness of Common Lisp's loop facility. For example, it allows traversal and construction of hash tables, arrays and vectors, general iteration, parallel iteration over two or more sequences, and the ability to terminate a collection when a condition are met. It is also possible to extend it to handle the traversal and collection in other data structures. The implementation can be downloaded from the author's homepage.

Section 2 presents the generalized comprehension system. Section 3 shows how the comprehension system can be extended to handle new data types. In the Section 4, the formal semantics of the system is discussed briefly. Section 5 compares the generalized comprehension system with the loop facility, and Section 6 discusses related work.

2. GENERALIZED COMPREHENSIONS

Let's first look at some simple uses of list comprehensions.

The expression

```
(collect (list) ((* x x))
  (in x '(1 2 3 4 5 6 7 8)))
```

evaluates to the list

```
(1 4 9 16 25 36 49 64)
```

In general, a collect expression has three parts;

1. A description of the type being built (*the collection type*),
2. a list of expressions giving the values to be inserted. and
3. one or more clauses.

The list comprehension given early in the introduction is written:

```
(collect list ((* x x))
  (in x l)
  (when (= (mod x 2) 1))
  (when (< x 5)))
```

To build a vector instead of a list, change the collection type from list to vector:

```
(collect vector ((* x x))
   ...)
```

2.1 Generator clauses

A clause of the form

(`if` *var exp*)

or

(`in` (*var*`*`) *exp*)

iterates over the data structure returned by the expression *exp*. The first form can be used when the expression evaluates to a sequence (for example, a list or a vector). Then, the variable will be bound to each element of the sequence, in turn.

In some cases, one position in the data structure may correspond to two or more values. For example, if `h` is a hash table, the clause

```
(in (k v) h)
```

will bind the variables `k` and `v` to each key and corresponding value in the table. If `a` is a vector (i.e., a one-dimensional array),

```
(in (i v) a)
```

will bind `i` and `v` to each index and corresponding value in the array.

As a more interesting example, consider the following function which builds a list of all permutations of a list (first given by Turner [5]).

```
(defun perms(l)
  (cond
   ((null l) (list nil))
   (t (collect list ((cons a b))
       (in a l)
       (in b (perms (remove a l)))))))
```

For example, the function call

```
  (perms '(a b c))
```

returns

```
  ((A B C) (A C B) (B A C)
   (B C A) (C A B) (C B A)).
```

2.2 Filters

A clause

(`when` *exp*)

will cause the iteration to skip any value for which the expression evaluates to `nil`. In the example early in this section, we used two when-clauses to skip those values for `x` which were even or greater or equal to 5.

2.3 Computed sequences

A clause

(`step` *var init-exp next-exp*)

resembles a `for`-statement in C. It will bind the variable *var* to the elements of an unbounded sequence, where the first element is computed by evaluating the *init-exp* and subsequent values are computed by evaluating *next-exp*, which may contain references to the previous value of *var*. For example,

```
(step x 0 (< x 100) (+ 1 x))
```

will bind `x` to each number from 0 to 99.

2.4 Terminating iteration

A clause

(`while` *exp*)

will terminate the iteration as soon as the expression *exp* evaluates to `nil`.

2.5 Parallel iteration

A clause

(`for` *clause*`*`)

combines several in- and step- clauses. All iterations are performed in parallel. For example, the clause

```
(for (in x '(a b c))
     (in y '(2 3 5 7 11)))
```

gives us three iterations; in the first `x` is bound to the atom `a` and `y` to the integer 1, in the second `x` is bound to `b` and `y` to 2, and in the last iteration `x` and `y` are bound to the atom `c` and the integer 3, respectively.

2.6 Pure side effects

A clause

(`do` *exp*)

will evaluate the expression for side effects. For example,

```
(collect (list) (x)
   (in x '(1 2 3 4 5))
```

```
(when (= (mod x 2) 1))
(do (print x)))
```

will print the integers 1 3 and 5 (and return a list of those integers).

2.7 Simple collection types

We have already seen uses of one basic collection type, `list`. This will simply collect the results in a list. In the same way, an expression

```
(collect vector ...)
```

will collect the results in a vector.

In some situations, we are only interested in the last value of a variable. The collection type "`t`" gives us this. For example,

```
(collect t (x)
   (in x '(1 2 3)))
```

returns 3. (There will be more interesting examples later.)

The collection type `nil` always returns `nil`. It is intended for situations where the comprehension is evaluated for side effects. Example:

```
(collect nil ()
   (in x '(1 2 3))
   (do (print x)))
```

The collection type `max` returns the maximum value, and sum the sum of all values. So

```
(collect max (x)
   (in x '(4 7 1 1)))
```

returns 7, and

```
(collect sum (x)
   (in x '(4 7 1 1)))
```

returns 13.

It is also possible to specify directly how values are to be combined. A collection type of the form `(reduce f)` takes a binary function f and uses it to combine the values generated by the iteration, so the collection type `(reduce #'+)` has the same effect as the collection type sum. Similarly,

```
(collect (reduce #'*) (x)
   (in x '(4 7 1 1)))
```

returns the product of the values in the list, 28.

2.8 Hash tables

Sofar, we have only considered comprehensions which either produce a sequence of results, or a single result. A hash table is of course an association of keys to values, which would correspond to a sequence of pairs of values rather than a sequence of values.

Let us first consider the simple case:

```
(collect hash-table (x)
   (in x '(a b c )))
```

This expression will build a hash table with the keys `a`, `b` and `c`. The corresponding value for each key is the atom `t`.

To associate a value with each key, we use collection types of the form `(hash-table` *type*`)` where *type* is also a collection type. For example

```
(collect (hash-table t) (k v)
   (in pair '((a 2) (b 3) (c 5)))
   (let k (car pair))
   (let v (cadr pair)))
```

builds a hash table with three entries, mapping the atom a to the integer 2, b to 3 and c to 5. The collection type `t` given as an argument indicates how values are to be combined. Thus, if the same key occurs several times in the sequence, the last entry is kept.

One may of course give other collection types as arguments. For example,

```
(collect (hash-table sum) (x 1)
   (in x l))
```

computes a hash table which maps each element of the list to its frequency.

But sometimes many keys have the same value. Suppose for example that `h` is a table mapping cities to their countries; say

City	Country
berlin	germany
hamburg	germany
liverpool	england
london	england
lyon	france
paris	france
oslo	norway

If we are interested in listing for each country the cities in that country, one might want to build a hash table which maps each country to a list of cities. This can be done with the following expression:

```
(collect (hash-table list) (v k)
   (in (k v) h))
```

Of course, if we preferred to store the cities in a vector, we would write

```
(collect (hash-table vector) (v k)
   (in (k v) h))
```

instead.

2.9 Collecting Arrays

Using the collection type `(array ...)` resembles collecting in a hash table. In both cases, the position of the entry must be given, and it also necessary to indicate how the values in each entry are to be combined.

The general syntax for the array collection type is

(`array` *type* (*exp*∗) `&rest args`),

where *type* indicates how entries are to be combined, the list of expressions gives the dimensions of the array, and the following arguments are passed directly to the `make-array` constructor. For example, the expression

```
(collect (array t (10)) (i (* i i))
     (in i '(2 3 5 7)))
```

will build the array

```
#(NIL NIL 4 9 NIL 25 NIL 49 NIL NIL).
```

As a more complex example, suppose we want to add a list of vectors, which may be of different lengths. (The shorter vectors should be treated as if they were padded with zeros.)

```
(defun add-vectors (&rest vectors)
  (let ((l (collect (max) ((length v))
                    (in (v) vectors))))
    (collect (array (sum) (l)
                    :initial-element 0) (i x)
      (in (v) vectors)
      (in (i x) v))))
```

The first collect expression simply collects the maximum length of the vectors in the list. The second collect expression creates an array of the maximum length, iterates over each array and adds its contents to the result.

2.10 More about complex collections

Most collections are simple; collecting the results in a list, a vector or a hash table, or simply summing the results. Even when we limit ourselves to simple data structures the collect macro is surprisingly versatile.

However, the collect macro also allows the result to be collected in more complex data structures. Let's look at some examples.

Suppose we have a list

```
vehicles = ((ford sedan john-smith id12334) ...),
```

where each vehicle is represented as a four-element list consisting of manufacturer, type of vehicle, name of owner, and identity number (to simplify the example we assume that all fields contain atoms).

A collect expression of the form

```
(collect ...
  (in entry vehicles)
  (let make (car entry))
  (let type (cadr entry))
  (let owner (caddr entry))
  (let id (nth  entry)
  ...)
```

can be used to extract various types of information from the list of vehicles. Writing

```
(collect sum (1)
  ...
  (when (eq make 'ford)))
```

counts the number of vehicles manufactured by Ford. If we want to create a table of all manufacturers and the number of vehicles that are listed for each make, we can write

```
(collect (hash-table sum) (make 1)
  ...)
```

One can of course use the same pattern to build a table of all owners and how many vehicles the have:

```
(collect (hash-table sum) (owner 1)
  ...)
```

Suppose now that we want to build a table of all makes. For each make we want to build a table of all people who own at least one vehicle of this make, and list the vehicles they own of this make. All this can be accomplished with this collect expression:

```
(collect
  (hash-table (hash-table list)) (make owner entry)
  ...)
```

Suppose we want to find groups of words that are each other's permutations, i.e., containing the same letters in another order. If the collection type hash-table is passed more than one argument, the rest of the argument list is passed to the constructor `make-hash-table`. Thus, the collection type `(hash-table ... :test #'equal)` creates a hash table which uses a hash function appropriate for strings.

```
(defun anagram (l n)
```

```
(let ((table
       (collect (hash-table list
                            :test #'equal)
         (key s)
         (in s l)
         (let key (sort (copy-seq s) #'char<)))))
  (collect list (l)
    (in (k l) table)
    (when (>= (length l) n))))
```

The program takes a list of strings and an integer, and returns those groups of words that are larger than n.

For example, the call

```
(anagram '("foo" "oof" "fo" "ofo" "of"
           "ba" "bar" "bart" "rab") 2)
```

returns

```
(("bar" "rab")
 ("fo" "of")
 ("foo" "oof" "ofo"))
```

2.11 Syntax and semantics

The syntax of a collect expression is given by the syntax

collect ::= (`collect` *type-exp exp∗ clause∗*)
type-exp ::= *type-name*
| (*type-name type-exp∗*`&rest` *args*)
clause ::= *generator*
| (`when` *exp*)
| (`let` *var exp*)
| (`do` *exp∗*)
| (`while` *exp*)
| (`for` *generator∗*)
generator ::= (`in` (*var∗*) *exp*)
| (`in` *var exp*)
| (`step` *var exp exp*)

where *exp* is any Lisp expression and *var* is a variable.

Suppose `l` is a list, `a` an arbitrary sequence (for example a vector) and `h` a hash table. The following iterations are implemented:

`(in v l)`	Bind `v` to each element of `l`
`(in v a)`	Bind `v` to each element of `a`
`(in (k v) a)`	As above, but let `k` be the index of the element
`(in (k v) h)`	Bind `k` and `v` to each key-value pair of `h`
`(in k h)`	Bind `k` to each key of `h`

3. EXTENDING COMPREHENSIONS

As the implementation of list comprehensions presented in this paper is written using the Common Lisp Object System (CLOS), it is easy to extend it to handle new data types. Suppose, for example, that we are interested in computing the average of a set of results. To introduce a new collection type, one gives a new method definition for `make-collector`:

```
(defmethod make-collector ((kind (eql 'ave)) args)
  (assert (null args) (kind args)
          "Collector ave expects 2 arguments")
  (values (list (gensym))
          '#'(lambda () (cons 0 0))
          '#'(lambda (s x) (cons (+ x (car s))
                                 (1+ (cdr s))))
          '#'(lambda (s) (/ (car s) (cdr s)))))
```

`make-collector` should take two arguments, the first should be the name of the collection type (usually an atom) and the second should be a list of arguments. In this case, the collection type `ave` should not receive any arguments so we include an assertion that checks that `args` is indeed the empty list. `make-collector` should return three quoted functions;

1. A function which returns an initial value giving the state of the collection before any values have been inserted.

   (In our example, the initial value is the pair `(0 . 0)`, i.e., total sum is 0, and 0 elements have been inserted.)

2. A function which takes the state of the collection and a new value, and returns a new state.

   In the example, we keep track of the sum and the number of elements inserted.

3. A function which takes a collection state and returns a final result.

   In the example, the result is calculated by dividing the sum with the number of elements collected.

With the definition above, calculating the average of a list, say,

```
(collect ave (x)
  (in x '(2 3 5 7)))
```

works as expected (the result is 17/4). Note that the collection type `ave` can also be combined with other collection types. For example, suppose `l` is a list associating atoms with values, say,

```
((john 2) (bob 23) (john 39)
 (john 38) (bert 32) (bob 102))
```

To compute the average of the values associated with each atom, simply write

```
(collect (hash-table ave) (name result)
   (in entry l)
   (let name (car entry))
   (let result (cadr entry)))
```

which gives us a table which maps `bert` to 32, `bob` to 125/2, and `john` to 79/3.

In the same way, it is possible to define iterators for new data types. Suppose, for example, that we want to iterate over the bits of integers. The method definition

```
(defmethod iterator ((arity (eql 1)) (a integer) f)
  (funcall f
    #'(lambda ()
        (let ((a0 a))
          (setq a (floor (/ a 2)))
          (values (not (= a0 0)) (mod a0 2))))))
```

of `iterator` describes how to iterate over an integer. An `iterator` method should take three arguments; its arity, i.e., the number of values obtained at each iteration; the object being iterated over, and a function. The function should be called with another function as argument. This function, in turn, is what one would normally think of as the iterator. Each time it is called, it returns two values; a truth value indicating whether there are more elements in the iteration, and the next value (if any).

Given the definition above, we can immediately perform iterations over integers. For example,

```
(collect list (x) (in x 42))
```

returns the list

```
(0 1 0 1 0 1).
```

4. SEMANTICS

In this section we will briefly discuss the semantics of the extensions of list comprehensions presented in this paper. No formal specification will be attempted, but hopefully the discussion will give the reader some idea on what a formal semantics would look like. We will only consider programs with no side-effects.

Wadler [6] gives a semantics for the basic form of list comprehensions as a set of reduction rules.

Suppose we evaluate a list comprehension

`(collect list (`*exp*`) ...)`

in a state σ. The clauses of the comprehension give us a sequence of states $\sigma_0, \sigma_1, \ldots, \sigma_{n-1}$. The expression *exp* is evaluated for each state, giving a sequence of values $x_0, \ldots, x_{n-1}$, which will be the the elements of the resulting list.

In-, when- and step-clauses can be seen as taking a state and return a sequence of zero or more states.

An in-clause

`(in` *var exp*`)`

evaluated in a state σ where *exp* evaluates to the list

$$(x_0 \ x_1 \ \ldots \ x_{n-1})$$

will give us the sequence of states

$$\sigma_0, \sigma_1, \ldots \sigma_{n-1},$$

where $\sigma_i = \sigma[var \mapsto x_i]$.

A when-clause

`(when` *exp*`)`

evaluated in a state σ will either give us the empty sequence of states, if *exp* evaluates to `nil`, or to the sequence consisting of the single state σ, if *exp* evaluates to anything else.

A step-clause

`(step` *var init-exp test-exp next-exp*`)`

evaluated in a state σ produces a sequence $\sigma_0, \sigma_1, \ldots, \sigma_{n-1}$ where

$$\sigma_0 = \sigma[var \mapsto x_0]$$
$$\sigma_{i+1} = \sigma_i[var \mapsto x_{i+1}], \text{for } i \leq 0$$

where x_0 is the result of evaluating expression *init-exp* in state σ, x_{i+1} is the result of evaluating *next-exp* in state σ_i, and *test-exp* evaluates to a non-nil value in all states $\sigma_0, \ldots \sigma_{n-1}$ and to nil in σ_n.

Recall that a for-clause gives a parallel combination of several step- and in- clause. To simplify the discussion, we only consider the case with two clauses, and assume that the sets of variables defined by the two clauses are disjoint.

Consider a clause

`(for` c_1 c_2`)`.

Assume that given the initial state σ, clause c_1 gives the sequence $\sigma_0, \sigma_1, \ldots, \sigma_{n-1}$ and clause c_2 the sequence of states $\rho_0, \rho_1, \ldots, \rho_{m-1}$.

Now, we would like the result of the for-clause to be a combination of the two sequences, i.e., a sequence of states of the same length as the shorter of the two sequences, where each state is a combination of the corresponding states in the two sequences.

Given states σ_i and ρ_i, we know that for any variable v, it is either defined in only one of the two states (or in none), or it has the same value in both. We define the resulting state δ_i as follows:

$$\delta_i(v) = \begin{cases} \sigma_i(v), & \text{if } v \text{ is defined in } \sigma_i \\ \rho_i(v), & \text{if } v \text{ is defined in } \rho_i \text{ but not in } \sigma_i \\ \text{undefined} & \text{otherwise} \end{cases}$$

The combination of the two iterations gives the sequence

$$\delta_0, \delta_1, \ldots, \delta_{k-1},$$

where k is the minimum of n and m.

5. A COMPARISON WITH THE LOOP FACILITY

The loop facility of Common Lisp [4] is a very powerful and flexible programming language construct for expressing various types of iteration. We will briefly present the main features of the loop facility and compare it to the generalized comprehension system.

For example, it allows iteration over a range of integers, a list, the entries of a hash table or the external symbols of a package. It supports various forms of value accumulation; for example, the result may be collected in a list, or (if the values are numerical) the result may be computed as the sum of values. For example, the following loop expression returns the list of the squares of integers from 1 to 9.

```
(loop
   for x from 1 to 9
   collect (* x x))
```

The loop facility allows several collect clauses, causing the values produced from different clauses to be interleaved. More interesting, a collect clause can name the result being collected, and the intermediate result can be used in the loop. For example,

```
(loop
  for x from 1 to 3
  collect (* x x) into y
  do (print y))
```

prints

```
(1)
(1 4)
(1 4 9)
```

Being able to name a result is convenient if we want to partition the elements of a list. For example, the following expression

```
(loop
  for x in l
  when (symbolp x) collect x into y
  when (integerp x) collect x into z
  finally return (list y z))
```

returns a list of two lists, where the first contains all elements that are symbols and the second all elements that are integers. If `l` is the list

```
(one 2 "three" 4 and five))
```

the result is `((ONE AND FIVE) (2 4))`.

Expressing this partition using comprehensions is not completely straight-forward. One solution is

```
(collect (array list (2)) (i x)
  (in x l)
  (when (or (symbolp x) (numberp x)))
  (let i (if (symbolp x) 0 1))))
```

which returns the two list in a two-element array. One might also consider using let-bound variables to store the accumulated result, i.e.,

```
(let
    ((symbols nil)
     (numbers nil))
  (collect nil ()
    (in x l)
    (do (when (symbolp x)
          (push x symbols)))
    (do (when (numberp x)
          (push x numbers))))
  (list symbols numbers))
```

A single list comprehension can contain several nested loops, as for example the permutation function given in an earlier section. A straight-forward rewrite of this function using the loop facility gives a slightly longer program. In the author's opinion, the version written using list comprehensions is much easier to understand.

```
(defun perms (l)
  (cond
   ((null l) (list nil))
   (t (loop
       for a in l
       append (loop
               for b in (perms (remove a l))
               collect (cons a b))))))
```

Compared to the comprehension system, the loop facility does not offer much support in the construction of complex data structures. Consider, for example, the last example in Section 2.8. Solving this problem in Common Lisp using the loop facility would not be much easier than solving it in some conventional programming language (assuming that the necessary data structures are available in some standard library).

The loop facility has sometimes been criticized for having a non-lispy syntax with keywords etc. Paul Graham [1] points out that some combinations of clauses in a loop expression do not have a well-defined meaning and recommends against the us of the loop facility.

6. RELATED WORK

Some of the extensions to list comprehensions proposed in this paper have counterparts in other programming languages.

Step clauses can easily be simulated in Haskell [2] using other constructs.

A recent proposal [7] describes an extension to Python called *dict comprehensions*. This proposal would allow the convenient creation of dictionaries from a sequence of key-value pairs, offering a functionality similar to the use of hash tables in expressions of the form:

(collect (hash-table t) (k v) ...))

Reade [3] proposed an extension to Haskell called *list terminators*. These correspond exactly to the while clauses described in Section 2.4.

7. CONCLUSIONS

We have presented an implementation of list comprehensions for Common Lisp. The system has been extended to handle the rich set of data types offered by the language. The system presented here can traverse and construct lists, vectors, arrays, and hash-tables (even nested structures), and has control structures (parallel iteration and termination) not normally offered by list comprehensions. It is also relatively straight-forward to extend the system to handle more data types.

8. REFERENCES

[1] Paul Graham. *ANSI Common Lisp*. Prentice Hall, 1996.

[2] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, April 2003.

[3] Chris Reade. Terminating comprehensions. *Journal of functional programming*, 3(2):247–250, April 1993.

[4] G. L. Steele Jr. *Common Lisp: The Language, Second Edition*. Digital Press, Bedford (MA), USA, 1990.

[5] D. A. Turner. Recursion equations as a programming language. In J. Darlington et al, editor, *Functional programming and its applications*. Cambridge University Press, 1982.

[6] Philip Wadler. List comprehensions. In Simon Peyton Jones, editor, *The implementation of functional programming languages*. Prentice-Hall, 1987.

[7] Barry A. Warsaw. Dict comprehensions. Python Enhancement Proposal 274, 2001.