Implementation of a "Lisp comprehension" macro

Guy Lapalme
Département d'informatique et de recherche opérationnelle
Université de Montréal
CP 6128, Succ "A"
Montréal Québec Canada
H3C 3J7
e-mail:lapalme@iro.umontreal.ca

January 1991

1 Introduction

This paper describes a set of Lisp macros that enable the use of "list comprehensions" which are a very powerful notation that provides a very compact expression of common list operations see [2] for a full discussion. List comprehensions have been introduced by David Turner in KRC, where they were called ZF-expressions [7]. They have since been introduced in several other pure functional languages like SASL [8], Miranda[1] [5] and Haskell[3].

List comprehension are often associated with "pure" functional languages but their principles are independant of the fact that assignment is allowed in the language. Having used them quite extensively in Miranda, we realized how much they would be useful in Lisp also. After all, comprehensions are merely "syntactic sugar" over standard functional programs but they give an intuitive reading of common operations over lists.

We first describe this notation in Miranda and then give the equivalent Lisp expression with our macros; we finally show the implementation of our macros which follow closely the derivation given by P. Wadler in [4, p127-138].

2 Description of list comprehensions in Miranda

List comprehensions have the form:

[1] *Miranda* is a registered trademark of Research Software Ltd.

[<expression> | <qualifier$_1$> , . . . , <qualifier$_n$>]

where each *qualifier* is either a *generator* (of the form *pat* <- *expr*) or a *filter* (a Boolean expression). The syntax derives from analogy with common set notation. For example:

$\{x|x \in S; \mathcal{P}(x)\}$

defines the subset of S for which P holds. Similarly:

[x | x <- s, p x]

defines the sublist of s for which $p\ x$ is *True*. The scope of variables defined using generators extends from the generator itself to all qualifiers defined to the right of the generator, and to the main expression.

For example, the following expression keeps all odd numbers of the list of numbers `xs`

```
[x | x <- xs ; odd x]
```

Generators can be nested as in

```
[x+y | x <- xs; odd x; y <- ys; even y]
```

which returns all possible sums for all odd numbers in `xs` with all even number of `ys`. The last generators change most rapidly and a later qualifier may refer to a variable defined in an earlier one. Thus this defines a simple but quite powerful list iteration mechanism; it is much simpler than most "loop" or "do" macros found in Common Lisp [6]; it is also more general than the "collect" macro given in [1, p 254-256]. It is in a way similar in spirit with the "Series Macro Package" of Waters [9] but considerably simpler (our macro is less than 20 lines of Lisp...). Of course, our model is much less powerful but is very useful anyhow. The limitations of the comprehension notation imply that the generated code is efficient without resorting to a program analysis like the one necessary in some cases for the series.

The following Miranda functions show the power of expression of list expressions (and also of pattern matching in the function heads):

- function for quicksort:

```
q_sort [] = []
q_sort (a:x) = q_sort [y|y <- x; y<a] ++ [a] ++ q_sort[y|y <- x; y >=a]
```

which means: sorting the empty list gives the empty list; sorting a list beginning with `a` followed by the list `x` is appending three lists: the first one is the result of sorting all elements of `x` smaller than `a`, the second only comprises the "pivot " `a` and the third is the result of sorting all elements of `x` greater or equal to `a`.

- function for finding all permutations of a list of elements

```
perms [] = [[]]
perms x  = [a:p | a <- x ; p <- perms (x -- [a])]
```

This is read as finding the permutations of the empty list is the list of one empty list; finding the permutations of list `x` is the list of all elements `a` from `x` in front of all possible permutations of the remaining elements of `x`. (`:` is the "cons" operator and `--` denotes list difference).

3 Expressing comprehensions in Lisp

Given these definitions, it is quite easy but tedious to translate them into Lisp using `cons`, `append`, `map`, `remove-if-not` ... but we would like to obtain a systematic and efficient translation from an equivalent Lisp expression bearing a much closer ressemblance to the original list comprehension.

So we have defined read macros associated with the `[` and `]` characters that build a call to another macro defining the "real" Lisp expression. In Lisp, each expression is well delimited, so we omit the "`|`" and the "`;`". The first expression in a comprehension is the value of the expression to be evaluated and the following ones are qualifiers; a qualifier is a generator if it is a three element list whose second element is the `<-` symbol; in this case the first element of the list should be a variable. If a qualifier is not a generator then it is a filter that either returns `nil` if it fails or not `nil` if it succeeds. So the previous examples are translated into Lisp in the following:

```
[ x (x <- xs) (oddp x)]

[(+ x y) (x <- xs) (y <- ys) (evenp y)]

(defun qsort (ax)
  (and ax
       (let ((a (car ax))
             (x (cdr ax)))
```

$$\mathbf{TQ}[\![\ [\ \mathrm{E}\ |\ ]\ +\!\!+\mathrm{L}\ ]\!] \equiv \qquad \text{(rule A)}$$
$$\quad (\ \mathbf{TE}[\![\ \mathrm{E}\ ]\!] : \mathbf{TE}[\![\ \mathrm{L}\ ]\!]\ )$$
$$\mathbf{TQ}[\![\ [\ \mathrm{E}\ |\ \mathrm{B}\ ,\ \mathrm{Q}\ ]\ +\!\!+\mathrm{L}\ ]\!] \equiv \qquad \text{(rule B)}$$
$$\quad \mathit{if}\ \mathbf{TE}[\![\ \mathrm{B}\ ]\!]\ \mathit{then}\ \mathbf{TQ}[\![\ [\ \mathrm{E}\ |\ \mathrm{Q}\ ]\ +\!\!+\mathrm{L}\ ]\!]\ \mathit{else}\ \mathbf{TE}[\![\ \mathrm{L}\ ]\!]$$
$$\mathbf{TQ}[\![\ [\ \mathrm{E}\ |\ \mathrm{v} \leftarrow \mathrm{L}_1\ ,\ \mathrm{Q}\ ]\ +\!\!+\mathrm{L}_2\ ]\!] \equiv \qquad \text{(rule C)}$$
$$\quad \mathit{letrec}$$
$$\qquad \mathrm{h} = \ \mathrm{us} \rightarrow \mathit{case}\ \mathrm{us}\ \mathit{of}$$
$$\qquad\quad [] \qquad\quad \rightarrow \quad \mathbf{TE}[\![\ \mathrm{L}_2\ ]\!]$$
$$\qquad\quad (\mathrm{v} : \mathrm{us}') \quad \rightarrow \quad \mathbf{TQ}[\![\ [\ \mathrm{E}\ |\ \mathrm{Q}\ ]\ +\!\!+(\mathrm{h}\ \mathrm{us}')\ ]\!]$$
$$\quad \mathit{in}\ (\ \mathrm{h}\ \mathbf{TE}[\![\ \mathrm{L}_1\ ]\!]\ )$$

Figure 1: Wadler's optimal translation rules for list comprehensions

```
(append (qsort [y (y <- x) (< y a)])
        (list a)
        (qsort [y (y <- x) (>= y a)])))))
```

```
(defun perms (x)
  (if (null x) '(())
    [(cons a p) (a <- x) (p <- (perms (remove a x :count 1)))]))
```

4 Translating into Lisp

Wadler [4, p132-135] describes a series of transformations to translate list comprehensions into an "enriched" λ-calculus (i.e. λ-calculus with pattern matching, `let` and `letrec`) and gives the following translation rules where $\mathbf{TE}[\![\ e\ ]\!]$ is the translation of expression e and $\mathbf{TQ}[\![\ \mathrm{c}\ +\!\!+\mathrm{l}\ ]\!]$ is the translation of a comprehension c with partial result l. In Lisp, $\mathbf{TE}[\![\ e\ ]\!]$ is merely the corresponding Lisp expression (as we use the backquote facility, we prefix e by a comma). A comprehension is translated by a macro with two parameters: the first one comprises the the expression and the qualifiers; the second is the partial result. This scheme is *optimal* in that it performs the minimum number of `cons` (i.e. exactly one for each element in the returned list). The macro is given in Figure 2 which is a straight translation of the rules where $\mathbf{TQ}[\![\ e\ ]\!]$ is converted to a recursive call to the macro. The read macros associated with the brackets are the following:

```
(defun open-bracket (stream ch)
```

```
(defmacro comp ((e &rest qs) l2)
  (if (null qs) `(cons ,e ,l2)          ; rule A
    (let ((q1 (car qs))
          (q (cdr qs)))
      (if (not(eq (cadr q1) '<-))       ; a generator?
          `(if ,q1 (comp (,e ,@q),l2) ,l2) ; rule B
        (let ((v (car q1))              ; rule C
              (l1 (third q1))
              (h (gentemp "H-"))
              (us (gentemp "US-"))
              (us1 (gentemp "US1-")))
          `(labels ((,h (,us)           ; corresponds to a letrec
                      (if (null ,us) ,l2
                        (let ((,v (car ,us))
                              (,us1 (cdr ,us)))
                          (comp (,e ,@q) (,h ,us1))))))
             (,h ,l1)))))))
```

Figure 2: "Lisp macro" adaptation of Wadler's translation rule

```
      (do ((l nil)
           (c (read stream t nil t)(read stream t nil t)))
          ((eq c '|]|) `(comp ,(reverse l) ()))
        (push c l))
    )
(defun closing-bracket (stream ch) '|]|)

(eval-when (compile load eval)
  (set-macro-character #\[ #'open-bracket)
  (set-macro-character #\] #'closing-bracket))
```

For example the translation of

```
[ x (x <- xs) (oddp x)]
```

is the following

```
(LABELS ((H-7 (US-7)
          (IF (NULL US-7) NIL
              (LET ((X (CAR US-7))
                    (US1-7 (CDR US-7)))
```

```
                (IF (ODDP X) (CONS X (H-7 US1-7)) (H-7 US1-7))))))
  (H-7 XS))
```

This is simply the definition and a call to an internal procedure that does a recursive walk on the list keeping only the odd elements. We build the procedure "in place" in order to build the right lexical environment for the main expression. This is not the translation that would come to mind in the the first place but its correctness is guaranteed by the series of transformations proven in [4, p 132-135]. Now the definition of "perms" becomes after being fully expanded:

```
(DEFUN PERMS (X)
  (IF (NULL X) '(NIL)
      (LABELS ((H-8 (US-8)
                    (IF (NULL US-8) NIL
                        (LET ((A (CAR US-8))
                              (US1-8 (CDR US-8)))
                             (LABELS ((H-9 (US-9)
                                           (IF (NULL US-9) (H-8 US1-8)
                                               (LET ((P (CAR US-9))
                                                     (US1-9 (CDR US-9)))
                                                    (CONS (CONS A P) (H-9 US1-9))))))
                                     (H-9 (PERMS (REMOVE A X :COUNT 1))))))))
               (H-8 X))))
```

5 Extensions

We have shown a direct translation of list comprehensions but it would be interesting to extend this work in some areas. In Common Lisp, lists are a special case of sequences. So using `(elt 1 x)` instead of `(car x)` and `(subseq 2 x)` instead of `(cdr x)` would result in a more general macro package capable of being applied to vectors, strings as well as lists. Unfortunately, `subseq` is specified as creating new instances of sequences at each call so in this case, the generated code would be less space efficient than in the case of list.

In Miranda, list comprehensions are more general than what we described here: pattern matching (using constants and even repeated variables) can be used in the left part of a generator, for example

```
[y | (1,y) <- xys]
```

returns the list of `y` in a list of 2-tuples such that the first element is equal to 1. As pattern-matching is not included in Miranda (except in the "new" `destructuring-bind` macro) we decided not to add it for this case of list comprehension. As lazy evaluation is used in Miranda, infinite lists can be specified in a generator built by a recurrence equation; in Lisp, this is not the usual evaluation mode, so we do not translate these cases.

Given these restrictions, we found this tool a very good and simple alternative to the loop/do/iterate/series macros for the simple but frequently occuring case of iterating over a list.

6 Conclusion

We have given a direct translation of list comprehensions in Lisp, we ran some simple tests in Allegro Common Lisp on a Sparc Station and we found that the resulting expressions ran between 30% slower and 20% faster than a "hand coded" Lisp version using the same "functional" style. The same tests were also run by Jon L. White using the Lucid "Development Quality" and "Production Quality" compilers and found that, in this case, most of the times the "comprehension" versions were faster than the "hand coded" ones. These translations are not the optimal ones because usually Lisp compilers generate much better code from looping constructs such as `do` or `dolist`. The Series macros of Waters[9] strive to translate expressions having a functional style into true iterative style and in some cases they achieve a more efficient translation at the cost of a comprehensive program analysis; our macro is only a straight implementation of the translation rules given by Wadler which could possibly be augmented to generate looping constructs when feasible. But as they stand now, the mechanically translated versions are never much slower than comprehensions so there is no real drawback in using them, but the rewards are great in terms of power of expression.

References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and interpretation of Computer Programs*. MIT Press, 1985.

[2] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice-Hall, 1988.

[3] P. Hudak and P. Wadler (editors). Report on the programming language Haskell, a non-strict purely functional language (Version 1.0). Technical

Report YALEU/DCS/RR777, Yale University, Department of Computer Science, April 1990.

[4] S. L. Peyton-Jones. *The Implementation of Functional Languages*. Prentice-Hall, 1987.

[5] Research Software Limited. *Miranda System Manual*. 1987.

[6] Guy L. Steele Jr. *Common Lisp, the language*. Digital Press, 2nd edition, 1990.

[7] D. A. Turner. *Functional Programming and Its Applications*, chapter Recursion Equations as a Programming Language, pages 1–28. Cambridge University Press, 1982.

[8] D. A. Turner. SASL language manual. UKC computing lab. report, The University of Kent at Canterbury, nov 1983.

[9] Richard C. Waters. The series macro package. *Lisp Pointers*, 3(1):7–28, 1990.