Simple and Efficient Compilation of List Comprehension in Common Lisp

Mario Latendresse
Bioinformatics Research Group
SRI International
Menlo Park, CA
latendre@ai.sri.com

ABSTRACT

List Comprehension is a succinct syntactic form to describe lists in functional languages. It uses nested generators (i.e., iterators) and filters (i.e., Boolean expressions). The former generates lists, whereas the latter restricts the contents of these lists. List Comprehension translation is commonly based on Wadler's rules that emit code based on recursive functions. This code often results either in stack overflow or in inefficient execution for many Common Lisp implementations.

We present a very simple technique to compile List Comprehension in the Loop Facility of Common Lisp, resulting in efficient code that avoids stack overflow. Our translation code is also very short, with an emitted code very close to the user-specified list comprehension.

We also present a translation into more fundamental constructs of Common Lisp that often results in even more efficient code, although this translation is more complex than using the Loop Facility.

The author has used the first translation technique for compiling a new language called *BioVelo*, which is part of bioinformatics software called Pathway Tools [2].

1. INTRODUCTION

In this paper, list comprehensions will be described as S-exprs of the form `(`$h\ q_1 \ldots q_n$`)` where the Common Lisp expression h is the *head* and the q_i are qualifiers. A qualifier is either a *generator* or a *filter*. A generator has the form `(`x `<-` X`)` where x is some identifier (this defines a new variable) and X is a Lisp expression of type list. A filter is a Boolean expression typically based on the variables defined by generators. (This syntax will be extended in Section 4.)

For example, `((list x y) (x <- '(1 2 3)) (y <- '(a b)))` gives the list `((1 a)(1 b)(2 a)(2 b)(3 a)(3 b))`. We are interested in translating these list comprehensions into efficient code in Common Lisp.

In March 2006, when searching on Google with the keywords '`translating List Comprehension Common Lisp`' the first entry was Guy Lapalme's paper, published in 1991 in Lisp Pointers, titled *Implementation of a "Lisp comprehension" macro* [3]. This is the main publication on translating list comprehensions in Common Lisp. Figure 1 shows Lapalme's macro. It is certainly short with about twenty lines of Common Lisp. It is based on Wadler's three translation rules that use recursive definitions [4, p132–135]. This macro handles the core of what a list comprehension is: a sequence of generators and filters with a head expression collecting the desired elements.

The simplicity of Lapalme's macro is appealing — it was considered simple enough to be the starting point of a more elaborate translation involving a few more features for a bioinformatics querying language, being implemented in Common Lisp, called *BioVelo*, which is based on List Comprehension.

But it was soon realized that many Common Lisp implementations were not handling tail-recursive calls well — and these are prevalent in Wadler's translation. For example, a list comprehension with several thousands iterations, but with a short list result, would overflow the stack. This was surprising since the generated code had tail-recursive calls, except for the recursive call handling the few elements that were part of the result. This was unavoidable when the code was interpreted on all tested implementations. One way to avoid stack overflow was to compile the code with optimization options for speed and not for debugging. This only partly solved the problem as some implementations would still create a stack overflow. Also, when analyzing the disassembled compiled code, we could still see inefficiency as some compilers maintain a constant stack space by literally copying stack frames. It was time to find another translation technique to avoid these shortcomings.

The Common Lisp Loop Facility [1, Chapter 26, by Jon L White] became a candidate as a translation target to avoid stack overflow and increase efficiency. Indeed, most Common Lisp implementations should implement it in such a way that the stack space used is not proportional to the number of iterations — even if the number of items collected is proportional to the number of iterations. But it was first thought that the translation of list comprehensions into the Loop Facility would be complex given its quirky syntax. It turned out just the opposite; the syntax of the Loop Facility is such that the translation is very simple and easy to understand. This was quite a surprise. It also turned out that the additional features of our list comprehensions in *BioVelo*, namely, the use of tuple in generators and binding assignments, were very easy to translate. Various optimizations

```
(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)         ; a letrec
                (if (null ,us) ,l2
                 (let ((,v   (car ,us))
                       (,us1 (cdr ,us)))
                   (comp (,e ,@q) (,h ,us1))))))
            (,h ,l1)))))))
```

Figure 1: A macro based on Wadler's rules to translate a Comprehension List in Common Lisp, using recursive functions. This is taken verbatim from Lapalme's paper.

can also be done without much effort.

Although the code emitted by our translation into the Loop Facility should be easy to compile into efficient fundamental code, we also investigated the translation of list comprehensions into fundamental Common Lisp constructs, namely, the `prog` with `go` statements. We wanted to know if the Loop Facility for the Common Lisp implementations tested was compiled adequately for the loops we were emitting. This lead to a translation that emits even more efficient code for these implementations. It showed that some implementations do compile the loop macro into efficient code while some others could do better.

The next section presents earlier work on translating list comprehensions in Common Lisp, in particular, Lapalme's technique based on Wadler's three rules and the problem of efficiently executing its emitted code for many Common Lisp implementations. In Section 3 we present our translation scheme into the Loop Facility with translation examples. Section 4 presents some extensions of List Comprehension that are easily handled by our translation technique. Section 5 shows some useful and simple optimizations that can be applied to our emitted code. Section 6 presents another translation mechanism based on the `prog` construct. Using four Common Lisp implementations, Section 7 compares the running times of the code emitted by our two translation techniques to the translation using Wadler's rules. We conclude in Section 8.

2. PREVIOUS WORK

Sven-Olof Nyström [5] has created a library under the name *collect* that translates list comprehensions into Common Lisp. The list comprehension syntax used by Nyström is a bit unusual. The implementation is much more complex than what is presented in this paper and it is not clear that it is efficient as the translation uses a liberal amount of `funcall`s, `apply`s, and other expensive operations. The translation is also much more complex, relying on CLOS

```
(LABELS ((H-1000 (US-1001)
           (IF (NULL US-1001)
               NIL
             (LET ((X (CAR US-1001))
                   (US1-1002 (CDR US-1001)))
               (LABELS ((H-1003 (US-1004)
                          (IF (NULL US-1004)
                              (H-1000 US1-1002)
                            (LET ((Y (CAR US-1004))
                                  (US1-1005 (CDR US-1004)))
                              (IF (> 0 X)
                                  (CONS X (H-1003 US1-1005))
                                (H-1003 US1-1005))))))
                        (H-1003 '(A B C)))))))
  (H-1000 I50000))
```

Figure 2: Translation of `(x (x <- i50000) (y <- '(a b c)) (> 0 x))` based on Lapalme's macro of Figure 1.

and more than 300 lines of code.

One of the main publication of a simple translation technique of list comprehensions in Common Lisp is Lapalme's paper [3]. Figure 1 shows Lapalme's macro to translate a comprehension list into recursive functions using Wadler's three rules [4, p132–135]. An example of a translation, shown in Figure 2, demonstrates a stack overflow problem frequent for many Common Lisp implementations. Indeed, assume that the global variable `i50000` is bound to a list of 50000 positive integers. This makes the Boolean expression `(> 0 x)` always false. That is, the result of executing this code is the empty list. All the recursive calls are in a tail location except the call `(H1003 US1-1005)` in `(CONS x (H1003 US1-1005))`, but it is never executed since no integer satisfies the expression `(> 0 x)`. Therefore, we expect a compiler to generate this code in such a way that the stack space used — due to recursive calls — does not depend on the length of `i50000`. Unfortunately, many Common Lisp compilers fail to do so as soon as the compiler options are not set toward strong optimization — for example, forcing the removal of debugging information.

Even if the list is not empty, it would be possible to create a translation such that the stack space does not grow with the length of the result. Using an accumulator for every defined recursive function would make this possible, but this would bring more complexity to the translation technique.

This problem of stack overflow alone has prompted the author to seek another translation technique of list comprehensions for Common Lisp. We discuss this translation in the next section.

3. TRANSLATION TO THE LOOP FACILITY

Our simple translation mechanism of list comprehensions into the Loop Facility is shown in Figure 3 which presents the entire translation via the macro `lc` and the function `lcr`. The macro `lc` always emits an external loop that repeats only once — this is a syntax trick to handle the case where the list of qualifiers does not contain a generator. See the end of this section for a proof sketch where an example of this case is presented.

```
(defmacro lc ((h &rest qs))
  `(loop repeat 1 ,@(lcr h qs)))

(defun lcr (h qs)
  (if (null qs) `(collect ,h)          ;; head
    (let ((q1 (first qs))
          (qr (rest  qs)))
      (if (eq (second q1) '<-)
        (let ((v  (first q1))          ;; generator
              (r  (third q1)))
          `(nconc (loop for ,v in ,r ,@(lcr h qr))))
       `(if ,q1 ,@(lcr h qr))    ;; filter
      ))))
```

Figure 3: Our translation mechanism of list comprehensions into the Loop Facility. It translates the same set of list comprehensions as Lapalme's macro of Figure 1. The macro `lc` is the main entry point for translating a list comprehension.

```
(loop repeat 1
  nconc (loop for x in i50000 nconc
                (loop for y in '(a b c)
                    if (> 0 x) collect x)))
```

Figure 4: Translation of `(x (x <- i50000) (y <- '(a b c)) (> 0 x))` by macro `lc` of Figure 3. This can be compared with the result of Figure 2 resulting from Wadler's rules.

The function `lcr` does the main translation. It has three cases: no more qualifier, the next qualifier `q1` is a filter or a generator. In the first case the translation is `` `(collect ,h) `` since there is no generator or filter to apply. In the second case, the generation of an `if` guard on the rest of the qualifiers captures the correct semantics. Notice that we use `,@` to insert the result of translating the rest of the qualifiers. This is required by the Loop Facility where one of `if`, `nconc` and `collect` will be taken as a Loop Facility keyword — not an expression to evaluate. The third case is for the generators, which is translated directly into a loop where `v` stands for the variable provided by the user and `r` the expression on the right side of `<-`.

As can be seen, the translation does not generate any recursive definitions. This is the main difference with Wadler's technique. Also, no new variable name is generated as all the loop variables are provided by the list comprehension itself. The translation code is also quite short, emitting code that is short and efficient (see Section 7 for some benchmark results).

Figure 4 shows the translation of the list comprehension `(x (x <- i50000) (y <- '(a b c)) (> 0 x))`. This code is shorter than the code resulting from Wadler's rules presented in Figure 2. Notice how the keyword `nconc` is inserted right after `repeat 1` and `i50000`. Similarly, the `collect` and `if` keywords were inserted by the same piece of code of `lcr`. This syntactic construct of the Loop Facility makes the translated code efficient as the result of the inner-most `collect` is passed up the nested loops without list copying.

The `lc` macro can handle tuple assignments in a generator, which is a form of restricted pattern matching. For

```
(loop repeat 1
   nconc (loop for (x y) in '((1 a) (2 b) (3 c))
               collect x))
```

Figure 5: The translation of `(x ((x y) <- '((1 a) (2 b) (3 c))))` by macro `lc` of Figure 3. Its execution gives the list `(1 2 3)`. This shows that the translation done by `lc` handles tuple assignments.

example, the list comprehension `(x ((x y) <- '((1 a) (2 b) (3 c))))` contains the tuple `(x y)` which is a pair of variables. Figure 5 shows the translation of `(x ((x y) <- '((1 a) (2 b) (3 c))))`. This list comprehension successively binds `x` to `1`, `2`, and `3` at the same time (i.e., in parallel) variable `y` is bound to `a`, `b`, and `c`. This cannot be handled by the `comp` macro of Figure 2.

In the rest of this section we present a proof sketch of the correctness of the translation. This is inductively done on the length of the list of qualifiers.

For the base cases: the translation is correct when the list of qualifiers is empty, that is, for a list comprehension of the form `(`h`)`. Indeed, it generates the code `(loop repeat 1 collect` h`)` which simply gives a list containing the singleton h. For a list of qualifiers of length 1, the qualifier is either a filter or a generator. This would be either `(loop repeat 1 if` q `collect` h`)` or `(loop repeat 1 nconc (loop for` x `in` X`collect` h`))` for `(`h q`)` and `(`h `(`x `<-` X`))`, respectively. In both cases, these are correct translations.

Inductively, assume the translation is correct for a list of qualifiers of length $n \geq 0$; then it is correct for a list of length $n+1$ by considering four cases as the last and penultimate qualifiers are either filters or generators. These four cases are

1. ...`nconc (loop for` x `in` X `nconc`
   `(loop for` y `in` Y `collect` h`))`...
2. ...`nconc (loop for` x `in` X `if` q `collect` h`)`...
3. ...`if` q `nconc (loop for` x `in` X `collect` h`)`...
4. ...`if` q `if` r `collect` h...

that correspond to the cases generator/generator, generator/filter, filter/generator, and filter/filter, respectively. They can be directly verified to be correct according to the Loop Facility semantics.

4. EXTENDING THE TRANSLATION

The previous section presented the basic translation into the Loop Facility. Here, we show that it can easily be extended in several directions still based on the Loop Facility.

The basic translation mechanism assumes that the generators are based on lists. That is, when writing `(x <- L)` the expression `L` is assumed to return a list. It would be useful to generalize `L` to vectors and strings. This can easily be done since the Loop Facility allows vectors and strings as arguments for the loop variable by replacing the keyword `in` to `across`.

The tuple matching still works with the data type vector although it does not make sense with the string type as each element, a character, cannot be a tuple. The translation is

```
(defmacro lce ((h &rest qs))
   `(loop repeat 1 ,@(lcer h qs)))

(defun lcer (h qs)
  (if (null qs) `(collect ,h)   ;; head
    (let* ((q1 (first qs))
           (op (second q1))
           (qr (rest  qs)))
      (if (member op '(<- <-- := <..))
        (let ((v (first q1)) ;;generator or binder
              (r (third q1)))
          `(nconc
            (loop for ,v
             ,(if (eq op '<-- ) 'across
              (if (eq op '<..) 'from 'in))
             ,(if (eq op ':=) `(list ,r) r)
             ,@(if (eq op '<..) `(to ,(fourth q1)))
            ,@(lcer h qr))))
       `(if ,q1 ,@(lcer h qr))    ;; filter
      ))))
```

Figure 6: Translation of extended List comprehensions into the Loop Facility. The extension includes generators that can be list, vectors, strings, and a range of integers. It also has the binding operation `:=`.

simple if the type of the sequence is known. This can be implemented by using, say, `<--` for string and vector, instead of `<-` for list.

In *BioVelo*, there is a binding operator denoted `:=`, where a tuple of variables or a single variable is on its left side and a general expression is on its right side. It binds the single variable or list of variables to the value or tuple of values of the expression. The single or multiple variables are then in the scope of all following qualifiers. This can be translated by still using a loop construct where the list is made of only one element, the value of the expression.

Our last extension provides the primary ingredient to generate lists of integers ranging from one integer to another integer — that is translated to the `from/to` Loop Facility. It is specified by using `<..` instead of `<-`, for a generator, as in `(x (x <.. 1 153))` which generates the list of integers from 1 to 153.

These extensions are handled by the translation macro `lce` of Figure 6.

5. OPTIMIZATIONS

Translating into the Loop Facility opens up several desirable optimizations. Here, we sketch the translation needed to accomplish some of them.

Sometimes only the size of a list comprehension is needed, as in `(length (lc (x (x <- i5000) (primep x))))` where `primep` is true only when `x` is a prime number. There is no need to collect the elements of the list to only return its length. Since the Loop Facility has the `count` keyword, we expect it to do exactly that — not constructing any list. So, in our translation, instead of using `collect` we would use the Loop Facility keyword `count`, that is emit `count x` instead of `collect x`; for generators, we replace `nconc` between the emitted loops by `sum`. Also, `lcr` would have to know the context, that is the fact that only the length is needed, not

```
;;; h is the head and qs the list of qualifiers
(defmacro lc2 ((h &rest qs))
   (let ((tail (gensym "tail")))
    `(prog ((,tail (list nil)) rhead)
       (setq rhead ,tail)
       ,(lc2r h qs tail)
       (return (cdr rhead))
     )))

;;; tail is the identifier to access the end
;;; of the resulting list.
(defun lc2r (h qs tail)
  (if (null qs)
      `(block nil (rplacd ,tail (list ,h))
                  (setq ,tail (cdr ,tail)))
    (let ((q1  (first qs))
          (qr  (rest  qs)))
      (if (eq (second q1) '<-)      ;; generator
          (let ((v     (first q1))
                (r     (third q1))
                (lv    (gensym "lv")))
            `(prog (,v (,lv ,r))
              loop
               (if (null ,lv) (go endls))
               (setq ,v (car ,lv))
               ,(lc2r h qr tail)
               (setq ,lv (cdr ,lv))
               (go loop)
              endls
               ))
        `(if ,q1 ,(lc2r h qr tail))  ;; filter
        ))))
```

Figure 7: A macro emitting fundamental code to translate list comprehensions. The resulting code should be as efficient as the Loop Facility, if not more efficient. The benchmark results of Section 7 demonstrate this for four Common Lisp implementations. Note, this code does not handle tuple assignments as the `lc` macro can.

the resulting list. This is easily done by adding a parameter to `lcr`.

In some other cases, only knowing if the length is zero or not is all that is needed. For example, for the following expression `(< 0 (length (lc (x (x <- X) (primep x)))))` there is no need to know the exact length of the lists. For this to be efficient, the iteration should stop as soon as one element is about to be added to the list of results. To do so, it suffices to use the `return` keyword instead of `collect`. That is, to emit `return nil` if length equal to zero is specified and otherwise to emit `return true` if length greater than zero is specified. In this case, the keyword `nconc` must be replaced with `do`.

6. TRANSLATION INTO FUNDAMENTAL CONSTRUCTS

We analyze the following question: how can we generate Common Lisp code that does not depend on the Loop Facility and is as efficient (if not more efficient) as the Loop Facility? More precisely, what are the more fundamental

```
(LET ((#:|tail21| '(NIL)) RHEAD)
  (BLOCK NIL
    (TAGBODY
      (SETQ RHEAD #:|tail21|)
      (PROG (X (#:|lv22| I5000))
       LOOP (IF (NULL #:|lv22|) (GO ENDLS))
        (SETQ X (CAR #:|lv22|))
        (PROG (Y (#:|lv23| I5000))
         LOOP (IF (NULL #:|lv23|) (GO ENDLS))
           (SETQ Y (CAR #:|lv23|))
           (IF (<= Y 10)
            (BLOCK NIL
              (RPLACD #:|tail21| (LIST X))
              (SETQ #:|tail21| (CDR #:|tail21|))))
           (SETQ #:|lv23| (CDR #:|lv23|))
           (GO LOOP)
         ENDLS)
        (SETQ #:|lv22| (CDR #:|lv22|))
        (GO LOOP)
       ENDLS)
      (RETURN (CDR RHEAD)))))
```

Figure 8: The translation of `(x (x <- i5000) (y <- i5000) (<= y 10))` by `lc2`.

constructs of Common Lisp that should be used when emitting code to ensure that it is not dependent on the translation of the Loop Facility?

Doing so eliminates any dependency on the Loop Facility and ensures that list comprehensions are translated into efficient code even when weak optimization settings of the compiler are used. It is also an analysis of what the Loop Facility ought to be doing to compile efficiently the kind of loop constructs that the macro `lc` emits.

All the list comprehensions need to do the following: keep a list of all the results being accumulated, and for each generator, iterate through the list, binding each element to a variable, optionally applying any filters found on its right. The `prog` construct — using the `go` statement — is a fundamental element of Common Lisp that can do this. Figure 7 presents a translation of list comprehensions that uses `prog`. The macro `lc2` emits the base code to define two variables, one of which is used by the code emitted by `lc2r` to add new elements to the final resulting list (see `(rplacd ,tail (list ,h))`). The variable `rhead` simply keeps the head of the resulting list so that the result can be returned from the most external `prog`; whereas the variable `tail` gives access to the tail of the resulting list to insert new elements. Notice that we use the constant list `(nil)` as the first cell for the result, although the `nil` element is dropped once the resulting list is created. This is required by the emitted `rplacd` in `lc2r` to operate properly when the first element is inserted. For each generator, a new variable is emitted for the list. Instructions `go` and `if` are used for flow control. The same tags `endls` and `loop` are reused for all generators as the inner generators shadow the outside tags which is the desired semantics. Figure 8 presents the translation of `(x (x <- i5000) (y <- i5000) (<= y 10))` by `lc2`.

Section 7 presents, among other things, the speed of execution of the code emitted by this macro compared to the one using the Loop Facility. Essentially, this code is often the most efficient regardless of the optimization settings of the compiler used.

7. BENCHMARK RESULTS

We compare the execution speed of the translation schemes `comp`, `lc`, and `lc2` (i.e., the macros presented in Figures 1, 3, and 7 respectively) using four Common Lisp implementations: Allegro Common Lisp, SBCL, LispWorks, and Clisp. We use the comprehensions `(x (x <- i5000) (y <- i5000) (< x 0))` which generates the empty list, and `(x (x <- i5000) (y <- i5000) (<= y 10))` which generates a list of 50000 integers. (The variable `i5000` is bound to the list of integers from 1 to 5000.)

We do not report speed of interpretation as we are interested only in the speed of compiled code. We only report that the interpretation of the code emitted by `comp` overflows the stack for three tested implementations, not the code emitted by `lc` and `lc2`. The exception is SBCL which compiles in its default evaluation mode — the `comp` code did not overflow the stack for this implementation.

We compiled the code using four different optimization settings. For each of these optimizations and each implementation we executed the code three times. We present the rounded arithmetic average of the times in Table 1. The times in bold-face indicate the fastest execution for either the A or B list comprehensions and for each implementation.

There is only one case where the code emitted by `comp` is slightly faster than the code emitted by `lc`, for SBCL with the optimization O0, but in this case, `lc2` emits slightly faster code. For two implementations, `comp` emits code that generates a stack overflow for all optimization settings, and for one implementation it happens if the optimization quality `debug` is not below 2. This is comprehensible since in this case the compiler keeps the evaluation stack to help debugging — even though tail-recursive calls are involved.

Overall, the translation mechanisms `lc` and `lc2` always avoid any stack overflow for the four implementations. It is indeed hard to see how any Common Lisp implementation would create a stack overflow given the code emitted by `lc` and `lc2`. For the four implementations, the fastest executions were obtained using either the `lc` or `lc2` translation.

8. CONCLUSION

We have shown that translating list comprehensions into the Common Lisp Loop Facility is very simple and creates more efficient code when compared to Wadler's translation technique based on recursive functions. The translation can also be easily extended to tuple assignment in generators and on vectors and strings. Compared to general recursive definitions, the Loop Facility avoids any stack overflow since it is often used as an iteration mechanism for which the stack space does not grow with the number of iterations.

The Loop Facility is efficient, but we have also shown that the fundamental construct `prog` can provide even more efficient code when translating some list comprehensions. This comes with an increase in the complexity of the translation mechanism, yet is not more complex than using recursive functions.

The reader may also have concluded that a subset of the Loop Facility can be used as if it were a List Comprehension Facility — although this is not as pleasant as the usual syntax of List Comprehension.

Implementation	Opt.	Time in milliseconds		
		comp	lc	lc2
Code A				
Allegro 8.0	O0	818	351	**317**
	O1	1105	421	344
	O2	so	871	861
	O3	so	984	878
LispWorks 4.4	O0	so	363	360
	O1	so	363	**357**
	O2	so	543	363
	O3	so	537	714
SBCL 0.9.18	O0	673	728	**653**
	O1	680	633	683
	O2	694	670	700
	O3	804	774	731
CLisp 2.41	O0	so	7340	**4786**
	O1	so	7227	5234
	O2	so	7407	5504
	O3	so	8229	7350
Code B				
Allegro 8.0	O0	861	424	**357**
	O1	1181	453	391
	O2	so	961	891
	O3	so	1005	898
LispWorks 4.4	O0	so	627	761
	O1	so	**620**	834
	O2	so	978	994
	O3	so	998	1535
SBCL 0.9.18	O0	704	**583**	593
	O1	741	587	613
	O2	751	587	654
	O3	758	593	758
CLisp 2.41	O0	so	11122	**9440**
	O1	so	11386	10805
	O2	so	10174	10174
	O3	so	12250	10741

Table 1: Execution times of the compiled list comprehensions A) `(x (x <- i5000) (y <- i5000) (< x 0))` **and B)** `(x (x <- i5000) (y <- i5000) (<= y 10))` **for four Common Lisp implementations using various optimization settings. The variable** `i5000` **is bound to the list of integers from 1 to 5000. Each time is an arithmetic average from three runs; 'so' denotes stack overflow. Optimization Ox is ((speed 3) (space x) (debug x) (safety x)). The shortest times for code A or B for each implementation are in bold. All implementations ran on Windows XP, 600 MHz CPU, 128 MB physical RAM.**

9. ACKNOWLEDGMENTS

Thanks to JonL White for discussing optimization results using a Lucid compiler, and an anonymous referee who pointed out a programming error.

10. REFERENCES

[1] Guy L. Steele Jr. *Common Lisp, The Language, Second Edition*. Digital Press, 1990.

[2] Peter D. Karp, Suzanne Paley, and Pedro Romero. The Pathway Tools software. *Bioinformatics*, 18(S):225–32, 2002.

[3] Guy Lapalme. Implementation of a "Lisp comprehension" macro. *SIGPLAN Lisp Pointers*, IV(2):16–23, 1991.

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

[5] Sven-Olof Nyström. Generalized comprehensions for Common Lisp. http://user.it.uu.se/~svenolof/Collect, 2007.