Fast, Maintainable, and Portable Sequence Functions

Irène Durand
Robert Strandh
University of Bordeaux
351, Cours de la Libération
Talence, France
irene.durand@u-bordeaux.fr
robert.strandh@u-bordeaux.fr

ABSTRACT

The Common Lisp sequence functions are challenging to implement because of the numerous cases that need to be taken into account according to the keyword arguments given and the type of the sequence argument, including the element type when the sequence is a vector.

For the resulting code to be fast, the different cases need to be handled separately, but doing so may make the code hard to understand and maintain. Writing tests that cover all cases may also be difficult.

In this paper, we present a technique that relies on a good compiler to optimize each separate case according to the information available to it with respect to types and values of keyword arguments. Our technique uses a few custom macros that duplicate a general implementation of the body of a sequence function. The compiler then specializes that body in different ways for each copy.

CCS Concepts

•Software and its engineering → Abstraction, modeling and modularity; Software performance; Compilers;

Keywords

Common Lisp, Compiler optimization, Portability, Maintainability

1. INTRODUCTION

The Common Lisp [1] sequence functions are challenging to implement for several reasons:

- They take several keyword parameters that modify the behavior in different ways. Several special cases must therefore be taken into account according to the value of these keyword parameters.
- In order for performance to be acceptable, different variations may have to be implemented according to the type of the sequence to be traversed.
- When the sequence is a vector, it may be necessary to specialize the implementation according to the element type of the vector, and according to whether the vector is a *simple array* or not.

For reasons of maintainability, it is advantageous to create a small number of versions, each one containing a single loop over the relevant elements. In each iteration of the loop, tests would determine the exact action based on current values of keyword arguments. In the case of a vector, the general array accessor `aref` would be used to access the elements.

On the other hand, for reasons of performance, it is preferable to create a large number of different versions of each function, each version being specialized according to the exact values of the keyword arguments given. In the case of a vector, it is also advantageous to have versions specialized to the available element types provided by the implementation. However, in this case, maintenance is problematic, because each version has to be maintained and tested separately.

A compromise used by some implementations is to use the Common Lisp macro system to abstract some of the specialization parameters as macro arguments. With this technique, a special version is created by a call to some general macro, providing different cases for keyword parameters, element types, test functions, etc. We find that this technique results in code that is extremely hard to understand, and therefore to be perceived as correct by maintainers.

In this paper, we present a different technique. We use the Common Lisp macro system, but not in order to create macros that, when called, create special versions of a sequence function. Instead, our technique makes it possible to write very few versions of each sequence function, thus keeping a high degree of maintainability. Most of our macros have no apparent role in our functions, so do not require the maintainer to understand them. Instead, they serve the sole purpose of allowing the compiler to generate efficient code.

Our technique was developed as part of the SICL project[1] which aims to supply high quality implementation-independent code for a large part of the Common Lisp standard.

2. PREVIOUS WORK

[1]See https://github.com/robert-strandh/SICL

ELS '17, April 3 – 6 2017, Brussels, Belgium Copyright is held by the owner/author(s).

Most implementations process list elements in reverse order when `:from-end` is true only when the specification requires it, i.e., only for the functions `count` and `reduce`.

We designed a technique [3] that allows us to always process list elements in reverse order very efficiently when `:from-end` is true. Since that paper contains an in-depth description of our technique, and in order to keep the presentation simple, in this paper, no example traverses the sequence from the end.

2.1 ECL and Clasp

The sequence functions of ECL have a similar superficial structure to ours, in that they take advantage of custom macros for managing common aspects of many functions such as the interaction between the `test` and `test-not` keyword arguments, the existence of keyword arguments `start` and `end`, etc. But these macros just provide convenient syntax for handling shared aspects of the sequence functions. They do not assist the compiler with the optimization of the body of the code.

For functions for which the Common Lisp specification allows the implementation to process elements from the beginning of the sequence even when `from-end` is *true*, ECL takes advantage of this possibility. For the `count` function applied to a list, ECL simply reverses the list before processing the elements.

The Common Lisp code base of Clasp is derived from that of ECL, and the code for the sequence functions of Clasp is the same as that of ECL.

2.2 CLISP

The essence of the code of the sequence functions of CLISP are written in C, which makes them highly dependent on that particular implementation. For that reason, CLISP is outside the scope of this paper.

2.3 SBCL

The sequence functions of SBCL are implemented using a mixed approach.

Macros are used to create special versions for the purpose of better performance. Transformations during compilation can replace a general call to a sequence function by a call to a special version when additional information is available such as when the sequence is a specialized vector, or when some keyword argument has a particular explicit value in the call.

Macros are also used to abstract details of combinations of values of keyword arguments.

However, when little information is available at the call site, a call to the general purpose function is maintained, and no particular attempt has been made to optimize such calls. As a result, in order to obtain high performance with the SBCL sequence functions, the programmer has to supply additional explicit information about the element type (in case of a vector) and explicit keyword arguments to such calls.

2.4 Clozure Common Lisp

The sequence functions of Clozure Common Lisp are implemented according to the approach where each function has a number of special versions according to the type of the sequence and the combination of the values of the keyword arguments.

However, the code in Clozure Common Lisp contains very few attempts at optimizing performance. For example, while there is an explicit test for whether a vector to be used as a sequence is a simple array, there is no attempt to specialize according to the element type of the vector.

3. OUR TECHNIQUE

We illustrate our technique with a simplified version of the function `find`. Recall that this function searches a sequence for the first occurrence of some item passed as an argument, and that the behavior can be altered as usual with parameters for determining the comparison function, a key function to apply to each element, the direction of the search, and the interval to search.

Our version is simplified in the following way:

- The only type of sequence handled is `vector`.
- The `test` function is fixed to be `eql`.
- The interval to search is the entire vector.
- The key function to apply to each element is fixed to be `identity`.
- The search is from the beginning of the vector.

In the general version of the `find` function, all these parameters must of course be taken into account, and then our technique becomes even more applicable and even more important. But the general version does not require any additional difficulty beyond what is needed for the special case, and the general case would only clutter the presentation, hence the special version which we will call `find-vector`.

Clearly, in terms of portability and maintainability, it would be desirable to implement `find-vector` like this:

```
(defun find-vector-1 (item vector)
  (declare (optimize (speed 3) (debug 0) (safety 0)))
  (loop for index from 0 below (length vector)
        for element = (aref vector index)
        when (eql item element)
          return element))
```

Unfortunately, most implementations would have difficulties optimizing this version, simply because the exact action required by the function `aref` depends on the *element type* of the vector, and whether the vector is a *simple-array*. This information is clearly *loop invariant*, but most compilers do not contain adequate optimization passes in order to duplicate and specialize the loop.

To improve code layout, in what follows, we assume the following type definition:

```
(deftype simple-byte-vector ()
  '(simple-array (unsigned-byte 8)))
```

To help the compiler, one can imagine a version like this:

```
(defun find-vector-2 (item vector)
  (declare (optimize (speed 3) (debug 0) (safety 0)))
  (if (typep vector 'simple-byte-vector)
      (loop for index from 0 below (length vector)
            for element = (aref vector index)
            when (eql item element)
              return element)
      (loop for index from 0 below (length vector)
            for element = (aref vector index)
            when (eql item element)
              return element)))
```

Here, we have illustrated the specialization with a non-standard element type, so that either an implementation-specific type predicate has to be used, or (as in our example) a call to `typep` is needed.

Whether a local declaration of the type of the vector in addition to the call to `typep` is required for the compiler to optimize the call to `aref` is of course implementation specific. Similarly, whether a special version (possibly implementation-specific) of `aref` is required also depends on the implementation.

Not only do we now have implementation-specific code, but we also have a maintenance problem. The loop will have to be duplicated for each sequence function, and for every specific type that the implementation can handle. This duplication requires separate tests for each case so as to guarantee as much coverage as possible. Given the number of combinations of types, plus the additional parameters we have omitted, this requirement quickly becomes unmanageable.

To solve this problem, we introduce a macro `with-type` that abstracts the implementation-specific information and that takes care of duplicating the loop:

```
(defmacro with-vector-type (vector-var &body body)
  `(macrolet ((vref (array index)
                `(aref ,array ,index)))
     (if (typep ,vector-var 'simple-byte-vector)
         (locally (declare (type simple-byte-vector
                                 ,vector-var))
           ,@body)
         (progn
           ,@body))))
```

Here, we have introduced a new operator named `vref` in the form of a local macro, and that is globally defined to expand to `aref`. This global definition works for SBCL, but different implementations may need different expansions in different branches. For example, some implementations might need for the macro to expand a call to `sbit` in a branch where the vector is a simple bit vector.

We have also introduced a local declaration for exact type of the vector in the specialized branch. Each implementation must determine whether such a declaration is necessary.

Using this macro, we can now write our function `find-vector` like this:

```
(defun find-vector-4 (item vector)
  (declare (optimize (speed 3) (debug 0) (safety 0)))
  (with-vector-type vector
    (loop for index from 0 below (length vector)
          for element = (vref vector index)
          when (eql item element)
            return element)))
```

We notice a couple of essential properties of this code:

- The exact set of available vector types in the implementation is hidden inside the macro `with-vector-type`, which would have a different version in different Common Lisp implementations, but there will be a single occurrence of this macro for all the sequence functions.
- The maintenance problem resulting from duplicating the loop has disappeared, because the macro `with-vector-type` is in charge of the duplication, making it certain that the copy is exact.

For a second example in the same spirit, consider how the keyword parameter `end` is handled when the sequence is a list. Again, we illustrate our technique with a simplified version of the `find` function.

As for the previous example, in terms of portability and maintainability, it would be desirable to implement `find-list` like this:

```
(defun find-list-1 (item list &key end)
  (declare (optimize (speed 3) (debug 0) (safety 0)))
  (loop for index from 0
        for element in vector
        when (and (not (null end)) (>= index end))
          return nil
        when (eql item element)
          return element))
```

As with the previous example, most Common Lisp implementations would have difficulties optimizing the code, even though the test `(null end)` is loop invariant. We solve this problem by introducing the following macro:

```
(defmacro with-end (end-var &body body)
  `(if (null ,end-var)
       (progn ,@body)
       (progn ,@body)))
```

The code for `find-list` can now be written like this:

```
(defun find-list-2 (item list &key end)
  (declare (optimize (speed 3) (debug 0) (safety 0)))
  (with-end end
    (loop for index from 0
          for element in vector
          when (and (not (null end)) (>= index end))
            return nil
          when (eql item element)
            return element)))
```

We notice that the `loop` body looks the same as in the portable and maintainable version shown before, and the only difference is that the loop has been wrapped in a call to the macro `with-end`. A good compiler will now specialize each of the two copies of the loop introduced by the `with-end` macro according to the value (i.e., `nil` or not) of the variable `end`. In the first copy, the entire first `when` clause of the loop will be removed. In the second copy, the test in the first `when` clause of the loop is reduced to the comparison between `index` and `end`.

4. PROPERTIES OF OUR TECHNIQUE

4.1 Performance

We compared performance of our technique for the `find` function shown in Appendix B to the performance of the `find` function shipped with SBCL.

We tested the performance of our technique only on SBCL because it is one of the few implementations that has a compiler that implements all the optimizations that our technique requires in order to perform well.

The results show a significant performance gain compared to the `find` function of SBCL. In fact, as it turns out, the SBCL sequence functions often require the programmer to declare the element type (when the sequence is a vector) in order for performance to be improved. We have not attempted to compare our technique to this case, for the simple reason one of the advantages of our technique is precisely

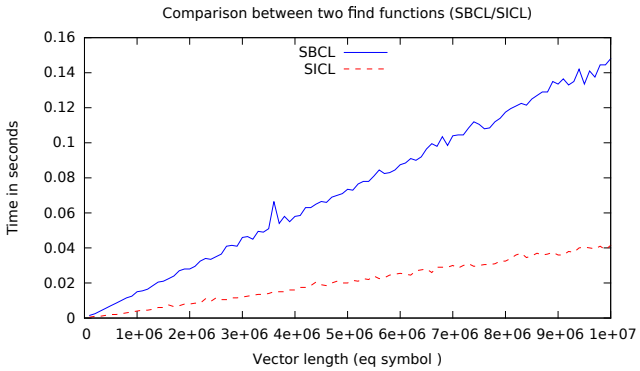
that no additional information is required in order for performance to be acceptable.

Most of our tests use relatively fast `test` functions such as `eq` or `eql`. This is a deliberate choice, as we want to compare the performance of the traversal of the sequence, and a more time-consuming test function would dominate the execution time.

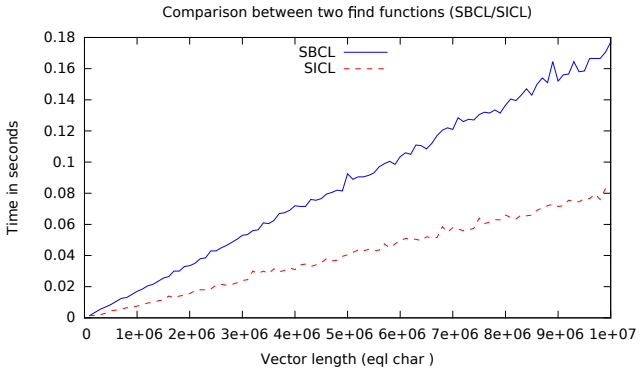
4.1.1 Results on vectors

When the sequence is a vector, the main performance consideration has to do with the element type of the vector. The parameters `start`, `end`, and `from-end` do not significantly alter the way the traversal is implemented. The `key` function may influence performance, but for the cases that we treat specially, only unspecialized vectors are concerned.

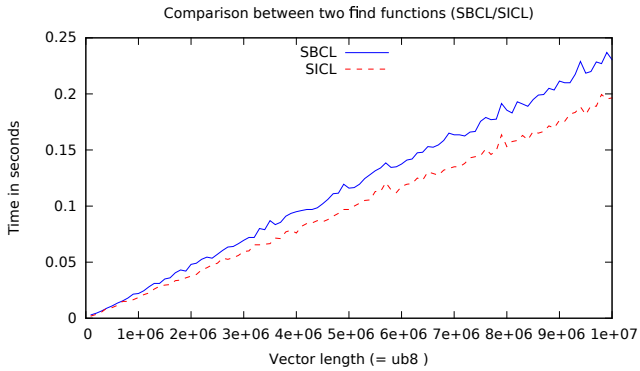
Our first test shows the performance comparison for an unspecialized vector with the `key` function being `identity` and the `test` function being `eq`. As shown in the diagram below, our function is around three times as fast as that of SBCL.



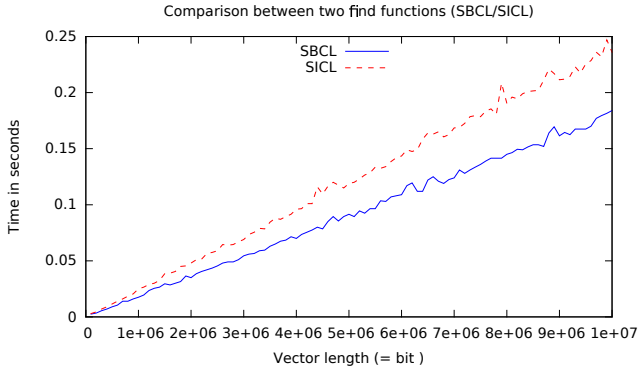
The next test shows a vector with the element type being `character`. The `key` function is `identity`, and the `test` function is `eql`. As shown in the diagram below, our function is around twice as fast as that of SBCL.



The next test shows a vector with the element type being `(unsigned-byte 8)` and the test function being `=`. In this case, our function is only around 20% faster than that of SBCL. This modest improvement can be explained by the fact that this test function is not one of the functions that we treat in a special way. An implementation that wishes better performance for this case can modify the macros to reflect this desire.



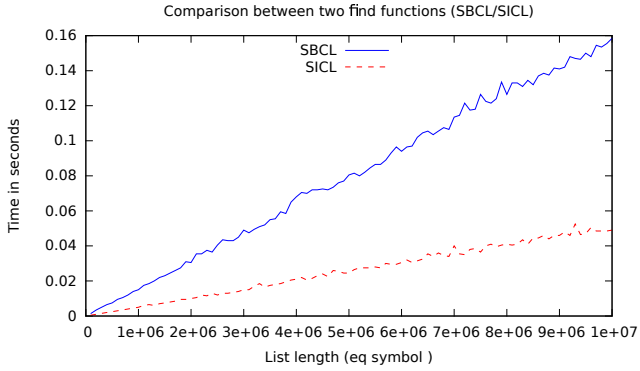
For completeness, we finally show a test for a vector with the element type being `bit`. In this case, our technique is slow, because it accesses the elements one at a time, whereas a good, native implementation of `find` would use available processor instructions to handle an entire word at a time [2].



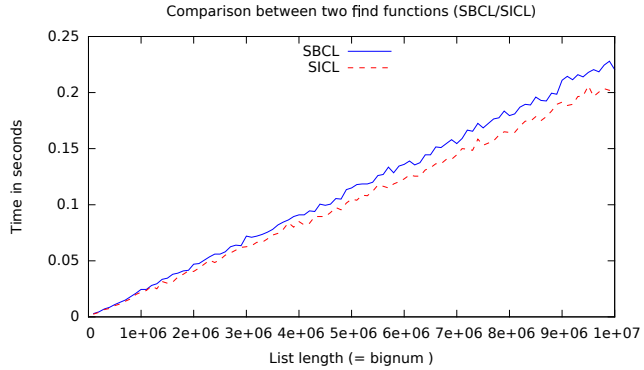
4.1.2 Results on lists

When the sequence is a list, the concept of element type is not applicable. The `key` function is important, because the sequence functions may be used for association lists. For that reason, we include a test with `car` as a key function. Also, for lists, the parameter `end` may influence the performance. In our implementation, we specialize the loops according to whether this parameter is `nil` or a number, allowing for two different specialized versions of the main traversal loop.

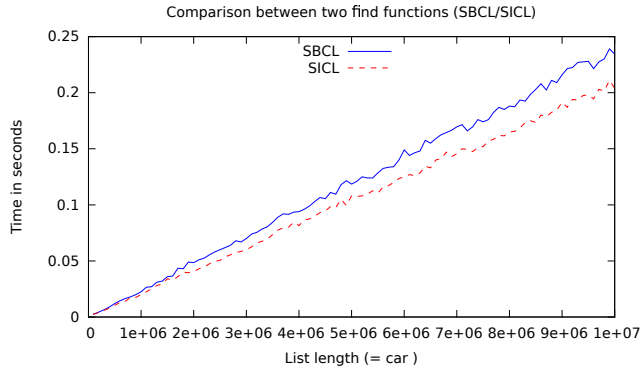
Our first test, like the first one on a vector, traverses a list of symbols. The test function is `eq`, and no `end` parameter has been given, which is equivalent to giving it the value `nil`. Again, our implementation is around three times as fast as that of SBCL.



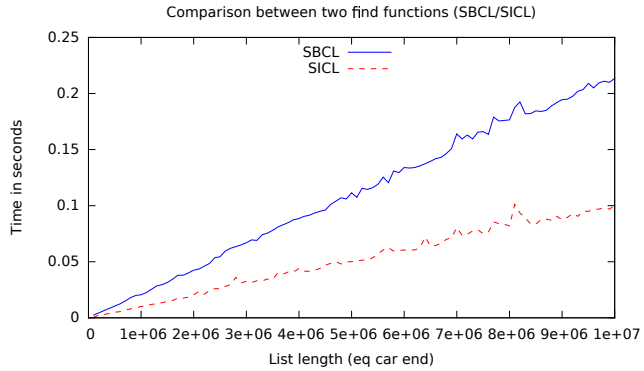
In the next test, the sequence is a list containing only bignums. The `test` function is `=`. As the diagram shows, our technique is only moderately faster than that of SBCL.



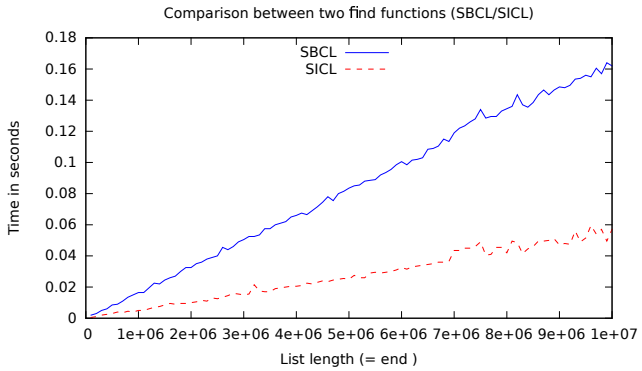
In the next test, the sequence is a list where each element is a list where the first element is a bignum, so that we can use `car` as a `key` function. The `test` function is still `=`. There seems to be no significant difference for this case, compared to the previous one, suggesting that the native implementation of `car` is so fast that calling `=` will dominate the computation.



In the next test, the sequence is a list of pairs of symbols; it uses `car` as key function, `eq` as test function and a non `nil` value for the `end` parameter. Our implementation is at least twice as fast as that of SBCL.



Our final test uses a non `nil` value for the `end` parameter. Despite the fact that we use a slightly more expensive `test` function (namely `=`), the performance of our implementation is very good; it is around three times as fast as that of SBCL.



4.2 Maintainability

From the point of view of maintainability, there are clear advantages to our technique. With only a small amount of macro code, we are able to hide the implementation details of the functions, without sacrificing performance.

The small amount of macro code that is needed to make our technique work is clearly offset by the considerable decrease in the code size that would otherwise have been required in order to obtain similar performance.

4.3 Disadvantages

There are not only advantages to our technique.

For one thing, compilation times are fairly long, for the simple reason that the body of the function is duplicated a large number of times. Ultimately, the compiler eliminates most of the code, but initially the code is fairly large. And the compiler must do a significant amount of work to determine what code can be eliminated. To give an idea of the orders of magnitude, in order to obtain fully-expanded code on SBCL, we had to increase the inline limit from 100 to 10 000, resulting in a compilation time of tens of seconds for a single function.

Another disadvantage of our technique is that it doesn't lend itself to applications using short sequences. For such applications, it would be advantageous to inline the sequence functions, but doing so would make each call site suffer the same long compilation times as we now observe for the ordinary callable functions.

Not all compilers are able to optimize the main body of a function according to some enclosing condition. For a Common Lisp implementation with a more basic compiler, no performance improvement would be observed. In addition, the duplication of the main body of the function would result in a very large increase of the code size, compared to a simpler code with the same performance.

For the special case of bit vectors, our technique will not be able to compete with a good native implementation of the sequence functions. The reason is that, despite the optimizations that the compiler can perform with our technique, the body of a typical sequence function still consists of a loop where each iteration treats a single element. A native implementation would not treat a single element in an iteration. Instead, it would take advantage of instructions that exist in most processors for handling an entire *word* at a time, which on a modern processor translates to 64 bits. An implementation that uses our technique would then typically handle bit vectors as a special case, excluded from the general technique.

5. CONCLUSIONS AND FUTURE WORK

We have presented a technique that allows implementations of most of the Common Lisp sequence functions that are simultaneously fast, maintainable, and portable, provided the compiler supplied by the implementation is sufficiently sophisticated to apply certain standard optimization techniques.

The main exception for which our technique is generally unable to compete with a native implementation is when the sequence is a bit vector. Any implementation that accesses the elements of the bit vector one at a time, rather than using native instructions that can handle an entire word at a time, is unable to match the native performance [2]. On the other hand, our technique allows the Common Lisp implementation to treat bit vectors as an exceptional case, and use our general technique for the other cases.

We have yet to perfect the exact declarations to include in our implementation, and the exact places where these declarations should be added. Different Common Lisp implementations have different requirements in this respect, so this work may have to be repeated for different implementations.

At the moment, we have been working exclusively with SBCL, for the simple reason that the SBCL compiler does provide the optimizations that are required in order for our technique to yield excellent performance. We intend to experiment with other major implementations as well in order to determine which ones are suited for our technique.

The Cleavir compiler framework of the SICL project will ultimately include a technique for *path replication* in intermediate code, that, while not specifically meant for the kind of optimization required for the technique presented in this paper, will have the same effect as more direct techniques currently used in advanced compilers.

Our technique is well adapted to processing sequences with a relatively large number of elements. When the sequence contains few elements, the overhead of the call and of processing the keyword arguments may be significant. Also, we do not take advantage of any declaration of element type, in the case when the sequence is a vector. We plan to investigate the possibility of modifying our macros so that definitions of specialized functions are automatically generated, leaving a fairly small general function that can then be inlined.

6. ACKNOWLEDGMENTS

We would like to thank Bart Botta, Pascal Bourguignon, and Philipp Marek for providing valuable feedback on early versions of this paper.

7. REFERENCES

[1] *INCITS 226-1994[S2008] Information Technology, Programming Language, Common Lisp*. American National Standards Institute, 1994.

[2] H. G. Baker. Efficient implementation of bit-vector operation in common lisp. *SIGPLAN Lisp Pointers*, III(2-4):8–22, Apr. 1990.

[3] I. Durand and R. Strandh. Processing list elements in reverse order. In *Proceedings of the 8th European Lisp Symposium*, ELS '15, 2015.

APPENDIX

A. PROTOCOL

In this appendix, we describe the macros and functions that are part of the protocol of our technique, used for implementing most of the sequence functions.

`apply-key-function` *element key-function* [*Function*]

This function takes an element of the sequence and a function to apply in order to obtain the object to use for comparison. For performance reasons, this function should be inlined.

A typical definition of this function might look like this:

```
(defun apply-key-function (element key-function)
  (declare (optimize (speed 3) (debug 0) (safety 3)))
  (declare (type function key-function))
  (cond ((eq key-function #'identity)
         element)
        ((eq key-function #'car)
         (car element))
        ((eq key-function #'cdr)
         (cdr element))
        (t
         (funcall key-function element))))
```

`canonicalize-key` *key-var* [*Macro*]

This macro takes a single argument which must be a variable that holds the value of the `&key` keyword argument. Its role is to make sure the contents of the variable is a function. A typical implementation might look like this:

```
(defmacro canonicalize-key (key-var)
  `(cond ((null ,key-var)
          (setf ,key-var #'identity))
         ((not (functionp ,key-var))
          (setf ,key-var (fdefinition ,key-var)))
         (t nil)))
```

`with-key-function` *key-function-var* `&body` *body* [*Macro*]

This macro takes a single argument which must be a variable that holds the value of the canonicalized `&key` keyword argument. It is used to duplicate *body* for different typical values for the *key* argument to many sequence functions. A typical implementation of this macro looks like this:

```
(defmacro with-key-function (key-function-var &body body)
  `(cond ((eq ,key-function-var #'identity)
          ,@body)
         ((eq ,key-function-var #'car)
          ,@body)
         ((eq ,key-function-var #'cdr)
          ,@body)
         (t
          ,@body)))
```

In each clause of the `cond` form in this macro, the inlined version of the function `apply-key-function` will be simplified in a different way by the compiler, resulting in a specialized loop.

`for-each-relevant-cons`
(cons-var index-var list start end from-end) `&body` *body* [*Macro*]

This macro executes *body* for each *relevant* `cons` *cell*. It takes into account the values of *start* and *end* to restrict the

execution to a particular sub-sequence, and it takes into account the value of *from-end* to determine the order in which the relevant `cons` cells are supplied to the *body* code. The parameter *cons-var* is the name of a variable that contains a reference to the relevant `cons` cell for each execution of *body*. Similarly, the parameter *index-var* is the name of a variable that contains the index of the particular `cons` cell to be processed.

Because of the size of the definition of this macro, due mainly to the code for processing `cons` cells in reverse order [3], we do not show its definition here.

`with-test-and-test-not`
(test-var test-not-var) `&body` *body* [*Macro*]

The role of this macro is to supply certain special cases for the possible values of the keyword parameters `test` and `test-not` of a typical sequence function. It is assumed that it has already been verified that at most one of the two keyword arguments has a value other than `nil`. A typical implementation might look like this:

```
(defmacro with-test-and-test-not
    ((test-var test-not-var) &body body)
  `(cond ((null ,test-not-var)
          (locally (declare (type function ,test-var))
            (cond ((eq ,test-var #'eq)
                   ,@body)
                  ((eq ,test-var #'eql)
                   ,@body)
                  (t
                   ,@body))))
         ((null ,test-var)
          (locally (declare (type function ,test-not-var))
            (cond ((eq ,test-not-var #'eq)
                   ,@body)
                  ((eq ,test-not-var #'eql)
                   ,@body)
                  (t
                   ,@body))))
         (t nil)))
```

`with-from-end` *from-end-var* `&body` *body* [*Macro*]

This macro duplicates *body* for the two cases where the value of the argument variable *from-end-var* is either *true* or *false*. A typical implementation looks like this:

```
(defmacro with-from-end (from-end-var &body body)
  `(if ,from-end-var
       (progn ,@body)
       (progn ,@body)))
```

`satisfies-two-argument-test-p`
item element test test-not [*Function*]

This function is typically inlined. It provides special cases for common values of the `test` and `test-not` keyword arguments of a typical sequence function. All but one of these cases will be eliminated in each branch of the macro `with-test-and-test-not` in which this function is located. A typical implementation might look like this:

```
(defun satisfies-two-argument-test-p
    (item element test test-not)
  (declare (optimize (speed 3) (debug 0) (safety 3)))
  (cond ((null test-not)
         (locally (declare (type function test))
           (cond ((eq test #'eq)
                  (eq item element))
                 ((eq test #'eql)
                  (eql item element))
                 (t
                  (funcall test item element)))))
        ((null test)
         (locally (declare (type function test-not))
           (cond ((eq test-not #'eq)
                  (not (eq item element)))
                 ((eq test-not #'eql)
                  (not (eql item element)))
                 (t
                  (not (funcall test-not item element))))))
        (t nil)))
```

`for-each-relevant-element`
element-var index-var vector start end
from-end `&body` *body* [*Macro*]

This macro is used to traverse the elements of a vector. The argument *element-var* is a symbol that is bound to each element during the execution of *body*. Similarly, *element-var* is a symbol that is bound to the index of the relevant element. The *vector* argument is an expression that must evaluate to a vector. The arguments *start* and *end* are expressions that evaluate to the two indices of the interval to traverse. Finally, `from-end` is a generalized Boolean that indicates whether the traversal is to be done from the end of the relevant interval. A typical implementation of this macro might look like this:

```
(defmacro for-each-relevant-element
    ((elementv indexv vector start end from-end)
     &body body)
  (let ((vectorv (gensym))
        (startv (gensym))
        (endv (gensym)))
    `(let ((,vectorv ,vector)
           (,startv ,start)
           (,endv ,end))
       (declare (type fixnum ,startv ,endv))
       (if ,from-end
           (loop for ,indexv downfrom (1- ,endv)
                   to ,startv
                 do (let ((,elementv
                           (aref ,vectorv ,indexv)))
                      ,@body))
           (loop for ,indexv from ,startv below ,endv
                 do (let ((,elementv
                           (aref ,vectorv ,indexv)))
                      ,@body))))))
```

`with-simple` *vector* `&body` *body* [*Macro*]

This macro simply checks whether *vector* is a `simple-array`, and duplicates *body* in each branch of the test. A typical implementation might look like this:

```
(defmacro with-simple (vector &body body)
  `(if (typep ,vector 'simple-array)
       (progn ,@body)
       (progn ,@body)))
```

`with-vector-type` *vector-var* `&body` *body* [*Macro*]

This macro duplicates *body* for each possible value of `array-upgraded-element-type` that the implementation provides. It also provides a local definition for the macro `vref` which we use instead of `aref` to access the elements of the vector in *body*. If the compiler of the implementation is unable to

specialize `aref` according to the element type, then the implementation may provide different definitions of the macro `vref` for different element types. Since the supported element types vary from one implementation to another, we do not provide an example of how this macro may be implemented.

B. EXAMPLE IMPLEMENTATION

As an example of how the sequence functions might be implemented using the functions and macros in Appendix A, we show our implementation of `find-list` which is called from `find` when the sequence is known to be a list:

```
(defun find-list
    (item list from-end test test-not start end key)
  (declare (optimize (speed 3) (debug 0) (safety 0)))
  (declare (type list list))
  (with-bounding-indices-list (start end)
    (with-key-function key
      (with-test-and-test-not (test test-not)
        (with-from-end from-end
          (for-each-relevant-cons
              (cons index list start end from-end)
            (let ((element (apply-key-function
                            (car cons) key)))
              (when (satisfies-two-argument-test-p
                     item element test test-not)
                (return-from find-list element)))))))))
```