The Series Macro Package

by

Richard C. Waters

MIT Artificial Intelligence Laboratory
545 Technology Sq.; Cambridge MA 02139
INTERNET: dick@ai.mit.edu

Abstract

The benefits of programming in a functional style are well known. Algorithms that are expressed as compositions of functions operating on series/vectors/streams of data elements are much easier to understand and modify than equivalent algorithms expressed as loops. Unfortunately, many programmers hesitate to use series expressions. In part, this is due to the fact that series expressions are typically implemented very inefficiently.

A portable Common Lisp macro package (called Series) has been implemented that can evaluate a wide class of series expressions very efficiently by transforming them into iterative loops. When using this class of series expressions, programmers can obtain the advantages of expressing computations as series expressions without incurring any runtime overhead.

Overview

The main body of this paper briefly summarizes everything you need to know to start using the Series macro package. It includes a detailed example of how series are intended to be used and information about how to obtain the macro package over the INTERNET.

A concise reference manual for the Series macro package is included as an appendix. This manual is excerpted from [5], which describes the macro package in full detail. A companion paper [6] gives an overview of the theory underlying the macro package and compares the macro package with related systems.

Series combine aspects of sequences, streams, and loops. Like sequences, series represent totally ordered multi-sets. In addition, the series functions have the same flavor as the sequence functions—namely, they operate on whole series, rather than extracting elements to be processed by other functions. For instance, the series expression below computes the sum of the positive elements in a list.

```
(collect-sum (choose-if #'plusp (scan '(1 -2 3 -4))))
  ⇒ 4
```

Like streams, series can represent unbounded sets of elements and are supported by lazy evaluation: The *i*th element of a series is not computed until it is needed. For instance, the series expression below returns a list of the first five even natural numbers and their sum. The call on `scan-range` returns a series of all the even natural numbers. However, since no elements beyond the first five are ever used, no elements beyond the first five are ever computed.

```
(let ((x (subseries (scan-range :from 0 :by 2) 0 5)))
  (values (collect x) (collect-sum x)))
  ⇒ (0 2 4 6 8) and 20
```

Like sequences and unlike streams, the act of accessing the elements of a series does not alter the series. For instance, both users of `x` above receive the same elements.

In a loop, a totally ordered multi-set of elements can be represented by the successive values of a variable. This is extremely efficient, because it avoids the need to store the elements as a group in any kind of data structure. In most situations, series expressions achieve this same high level of efficiency, because they are automatically transformed into loops before being evaluated or compiled. For instance, the first expression above is transformed into a loop like the following.

```
(let ((sum 0))
  (dolist (i '(1 -2  3 -4) sum)
    (if (plusp i) (setq sum (+ sum i))))) ⇒ 4
```

A wide variety of algorithms can be expressed clearly and succinctly using series expressions. In particular, most of the loops programmers typically write can be replaced by series expressions that are much easier to understand and modify, and just as efficient. From this perspective, the key feature of series is that they are supported by a rich set of functions. These functions more or less correspond to the union of the operations provided by the sequence functions, the `loop` clauses, and the vector operations of APL.

Unfortunately, some series expressions cannot be transformed into loops. This matters because, while transformable series expressions are much more efficient than equivalent expressions involving sequences or streams, nontransformable series expressions are much less efficient. Whenever a problem comes up that blocks the transformation of a series expression, a warning message is issued. Based on the information in the message, it is usually easy to provide an efficient fix for the problem.

Fortunately, most series expressions can be transformed into loops. In particular, pure expressions (ones that do not store series in variables) can always be transformed. As a result, the best approach for programmers to take is to simply write series expressions without worrying about transformability. When problems come up, they can be ignored (since they cannot lead to the computation of incorrect results) or dealt with on an individual basis.

The series data type. The Series macro package supports the series data type and a suite of functions operating on this data type. Series are self-evaluating objects. In analogy with `#(`*items*`)`, the `#` macro character syntax `#Z(`*items*`)` is provided for writing literal series. This same syntax is used when series are printed. If `*print-length*` is not `nil`, then long (or unbounded) series are abbreviated using "...", as in the second example below.

```
#Z(a (b c) d) ⇒ #Z(a (b c) d)
#Z(a b . #1=(c d . #1#)) ⇒ #Z(a b c d c d ...)
```

Predefined series functions. The heart of the Series macro package is a set of several dozen functions that operate on series. These functions divide naturally into three classes. *Scanners* produce series without consuming any. *Transducers* compute series from series. *Collectors* consume series without producing any.

Predefined scanners include: `series` which creates a series indefinitely repeating a given value, `scan` which enumerates the elements in a sequence, `scan-range` which enumerates the integers in a range, and `scan-plist` which creates a series of the indicators in a property list along with a second series containing the corresponding values. The first argument of `scan` specifies the type of sequence to be scanned. If omitted, the type defaults to `list`.

```
(series 'a) ⇒ #Z(a a a ...)
(scan '(a b c)) ⇒ #Z(a b c)
(scan 'vector '#(a b c)) ⇒ #Z(a b c)
(scan-range :from 1 :upto 3) ⇒ #Z(1 2 3)
(scan-plist '(a 1 b 2)) ⇒ #Z(a b) and #Z(1 2)
```

Predefined transducers include: `positions` which returns the positions of the non-null elements in a series and `choose` which selects the elements of its second argument that correspond to non-null elements of its first argument.

```
(positions #Z(a nil b c nil nil)) ⇒ #Z(0 2 3)
(choose #Z(nil T T nil) #Z(1 2 3 4)) ⇒ #Z(2 3)
```

Predefined collectors include: `collect` which combines the elements of a series into a sequence, `collect-sum` which adds up the elements of a series, `collect-length` which computes the length of a series, and `collect-first` which returns the first element of a series. The first argument of `collect` specifies the type of the sequence to be produced. If omitted, the type defaults to `list`.

```
(collect #Z(a b c)) ⇒ (a b c)
(collect 'simple-vector #Z(1 2 3)) ⇒ #(1 2 3)
(collect-sum #Z(1 2 3)) ⇒ 6
(collect-length #Z(a b c)) ⇒ 3
(collect-first #Z(a b c)) ⇒ a
```

Higher-Order series functions. The Series macro package provides a number of higher-order functions, which support general classes of series operations. For example, the function `(map-fn` *type function items*`)` supports the generic transduction operation of mapping a function over a series. The *type* argument specifies the type of the elements in the series being created. Each element of the output is computed by applying *function* to the corresponding element of *items*.

```
(map-fn T #'sqrt #Z(4 9 16)) ⇒ #Z(2 3 4)
```

Scanning is supported by `(scan-fn` *type init step test*`)`. The *type* argument specifies the type of the elements in the series being created. The function *init* is called to obtain the first element of the output. Subsequent elements are obtained by applying the function *step* to the previous element. The series consists of the elements up to, but not including, the first element for which the function *test* returns non-null.

```
(scan-fn 'integer #'(lambda () 3) #'1- #'minusp)
  ⇒ #Z(3 2 1 0)
```

Collecting is supported by `(collect-fn` *type init function items*`)`. The elements of the series *items* are combined together using *function*. The quantity returned by *init* is used as an initial seed value for the accumulation. The *type* argument specifies the type of the summary value returned.

```
(collect-fn 'integer #'(lambda () 3) #'+ #Z(1 2 3))
  ⇒ 9
```

Convenient support for mapping. Mapping is by far the most commonly used series operation. In cognizance of this fact, the Series macro package provides three mechanisms that make it easy to express particular kinds of mapping. The `#` macro character syntax `#M`*f* converts a function *f* into a transducer that maps *f*.

```
(#Msqrt #Z(4 16)) ≡ (map-fn T #'sqrt #Z(4 16))
  ⇒ #Z(2 4)
```

The form `mapping` can be used to specify the mapping of a complex expression over one or more series without having to write a literal `lambda` expression. For example,

```
(mapping ((x (scan '(2 -2 3))))
  (expt (abs x) 3)) ⇒ #Z(8 8 27)
```

is the same as

```
(map-fn T #'(lambda (x) (expt (abs x) 3))
       (scan '(2 -2 3))) ⇒ #Z(8 8 27)
```

The form `iterate` is the same as `mapping` except that the value `nil` is always returned.

```
(iterate ((x (scan '(2 -2 3))))
  (if (plusp x) (prin1 x))) ⇒ nil <after printing "23">
```

To a first approximation, `iterate` and `mapping` differ in the same way as `mapc` and `mapcar`. In particular, like `mapc`, `iterate` is intended to be used in situations where the body is being evaluated for side effect rather than for its result. However, due to the lazy evaluation semantics of series, the difference between `iterate` and `mapping` is more than just a question of efficiency. If `mapping` is used in a situation where the output is not used, no computation is performed, because series elements are not computed until they are used.

User-defined series functions. As shown by the definitions of simplified versions of `collect-sum` and `mapping` shown below, the standard Lisp forms `defun` and `defmacro` can be used to define new series functions. However, when a series function is defined with `defun`, the Series macro package is not capable of optimizing series expressions containing this new function unless the declaration `optimizable-series-function` is specified in the `defun`. This declaration is not required when using `defmacro`.

```
(defun simple-collect-sum (numbers)
   (declare (optimizable-series-function))
  (collect-fn 'number #'(lambda () 0) #'+ numbers))

(defmacro simple-mapping (var-value-pair-list &body b)
  (let* ((pairs (scan var-value-pair-list))
         (arg-list (collect (#Mcar pairs)))
         (value-list (collect (#Mcadr pairs))))
    `(map-fn T #'(lambda ,arg-list ,@ b)
             ,@ value-list)))
```

Benefits. The advantage of series expressions is that they retain most of the virtues of loop-free, functional programming, while eliminating most of the costs. However, given the fact that optimization is not always possible, the question naturally arises as to whether optimization is possible in a wide enough range of situations to be of real pragmatic benefit.

An informal study [3] was undertaken of the kinds of loops programmers actually write. This study suggests that approximately 80% of the loops programmers write are constructed by combining a few common kinds of looping algorithms in a few simple ways. The Series macro package is designed so that all of these loops can be trivially expressed as optimizable series expressions. Many more loops can be expressed as optimizable series expressions with only minor modification.

Moreover, the benefits of using series expressions go beyond replacing individual loops. A major shift toward using series expressions would be a significant change in the way programming is done. At the current time, most programs contain one or more loops and most of the interesting computation in these programs occurs in these loops. This is quite unfortunate, since loops are generally acknowledged to be one of the hardest things to understand in any program. If series expressions were used whenever possible, most programs would not contain any loops. This would be a major step forward in conciseness, readability, verifiability, and maintainability.

It should also be noted that, while this paper concentrates on the Lisp implementation of series, the ideas behind the Series macro package have nothing to do with the Lisp language *per se*. A Pascal implementation of series is described in [2, 4].

Example

The following example shows what it is like to use series expressions in a realistic programming context. The example consists of two parts: a pair of functions that convert between sets represented as lists and sets represented as bits packed into an integer and a graph algorithm that uses the integer representation of sets.

Bit sets. Small sets can be represented very efficiently as binary integers where each 1 bit in the integer represents an element in the set. Here, sets represented in as binary integers are referred to as *bit sets*.

Common Lisp provides a number of bitwise operations on integers, which can be used to manipulate bit sets. In particular, `logior` computes the union of two bit sets while `logand` computes their intersection.

```
(defun bset->list (bset universe)
  (collect (choose (#Mlogbitp (scan-range :from 0)
                              (series bset))
                   (scan universe))))

(defun list->bset (items universe)
  (collect-fn 'integer #'(lambda () 0) #'logior
    (mapping ((item (scan items)))
      (ash 1 (bit-position item universe)))))

(defun bit-position (item universe)
  (or (collect-first
        (positions
          (#Meq (series item) (scan universe))))
      (1- (length (nconc universe (list item))))))
```

Figure 1: Converting between lists and bit sets.

The functions in Figure 1 convert between sets represented as lists and bit sets. To perform this conversion, a mapping has to be established between bit positions and potential set elements. This mapping is specified by a *universe.* A universe is a list of elements. If a bit set integer b is associated with a universe u, then the ith element in u is in the set represented by b if and only if the ith bit in b is 1. For example, given the universe `(a b c d e)`, the integer `#b01011` represents the set `{a,b,d}`. (By Common Lisp convention, the 0th bit in an integer is the least significant bit.)

Given a bit set and its associated universe, the function `bset->list` converts the bit set into a set represented as a list of its elements. It does this by scanning the elements in the universe along with their positions and constructing a list of the elements which correspond to 1s in the integer representing the bit set. (When no `:upto` argument is supplied, `scan-range` counts up forever.)

The function `list->bset` converts a set represented as a list of its elements into a bit set. Its second argument is the universe that is to be associated with the bit set created. For each element of the list, the function `bit-position` is called to determine which bit position should be set to 1. The function `ash` is used to create an integer with the correct bit set to 1. The function `collect-fn` is used to combine the integers corresponding to the individual elements together into a bit set corresponding to the list.

The function `bit-position` takes an item and a universe and returns the bit position corresponding to the item. The function operates in one of two ways depending on whether or not the item is in the universe. The first line of the function contains a series expression that determines the position of the item in the universe. If the item is not in the universe, the expression returns `nil`. (The function `collect-first` returns `nil` if it is passed a series of length zero.)

If the item is not in the universe, the second line of the function adds the item onto the end of the universe and returns its position. The extension of the universe is done by side effect so that it will be permanently recorded in the universe.

Figure 2 shows the definition of two collectors that operate on series of bit sets. The first function computes the

```
(defun collect-logior (bsets)
    (declare (optimizable-series-function))
  (collect-fn 'integer #'(lambda () 0)
              #'logior bsets))

(defun collect-logand (bsets)
    (declare (optimizable-series-function))
  (collect-fn 'integer #'(lambda () -1)
              #'logand bsets))
```

Figure 2: Operations on series of bit sets.

union of a series of bit sets, while the second computes their intersection.

Live variable analysis. As an illustration of the way bit sets might be used, consider the following. Suppose that in a compiler, program code is being represented as blocks of straight-line code connected by possibly cyclic control flow. The top part of Figure 3 shows the data structure that represents a block of code. Each block has several pieces of information associated with it. Two of these pieces of information are the blocks that can branch to the block in question and the blocks it can branch to. A program is represented as a list of blocks that point to each other through these fields.

In addition to control flow information, each structure contains information about the way variables are accessed. In particular, it records the variables that are written by the block and the variables that are used by the block (i.e., either read without being written or read before they are written). An additional field (computed by the function `determine-live` discussed below) records the variables that are *live* at the end of the block. (A variable is live if it has to be saved, because it can potentially be used by a following block.) Finally, there is a temporary data field, which is used by functions (such as `determine-live`) that perform computations involved with the blocks.

The rest of Figure 3 shows the function `determine-live` which, given a program represented as a list of blocks, determines the variables that are live in each block. To perform this computation efficiently, the function uses bit sets. The function operates in three steps. The first step (`convert-to-bsets`) looks at each block and sets up an auxiliary data structure containing bit set representations for the written variables, the used variables, and an initial guess that there are no live variables. This auxiliary structure is defined by the third form in Figure 3 and is stored in the `temp` field of the block. The integer 0 represents an empty bit set.

The second step (`perform-relaxation`) determines which variables are live. This is done by relaxation. The initial guess that there are no live variables in any block is successively improved until the correct answer is obtained.

The third step (`convert-from-bsets`) operates in the reverse of the first step. Each block is inspected and the bit set representation of the live variables is converted into a list, which is stored in the live field of the block.

On each cycle of the loop in `perform-relaxation`, a block is examined to determine whether its live set has to be changed. To do this (see the function `live-estimate`), the successors of the block are inspected. Each successor needs to have available to it the variables it uses, plus the variables that are supposed to be live after it, minus the variables it writes. (The function `logandc2` takes the difference of two bit sets.) A new estimate of the total set of variables needed by the successors as a group is computed by using `collect-logior`.

```
(defstruct (block (:conc-name nil))
  predecessors ;Blocks that can branch to this one.
  successors   ;Blocks this one can branch to.
  written      ;Variables written in the block.
  used         ;Variables read before written.
  live         ;Variables needed at exit.
  temp)        ;Temporary storage location.

(defun determine-live (program-graph)
  (let ((universe (list nil)))
    (convert-to-bsets program-graph universe)
    (perform-relaxation program-graph)
    (convert-from-bsets program-graph universe))
  program-graph)

(defstruct (temp-bsets (:conc-name bset-))
  used written live)

(defun convert-to-bsets (program-graph univ)
  (iterate ((block (scan program-graph)))
    (setf (temp block)
          (make-temp-bsets
            :used (list->bset (used block) univ)
            :written (list->bset (written block)
                                 univ)
            :live 0))))

(defun perform-relaxation (program-graph)
  (let ((to-do program-graph))
    (loop
      (when (null to-do) (return (values)))
      (let* ((block (pop to-do))
             (estimate (live-estimate block)))
        (when (not (= estimate
                      (bset-live (temp block))))
          (setf (bset-live (temp block)) estimate)
          (iterate ((prev (scan (predecessors block))))
            (pushnew prev to-do)))))))

(defun live-estimate (block)
  (collect-logior
    (mapping ((next (scan (successors block))))
      (logior (bset-used (temp next))
              (logandc2 (bset-live (temp next))
                 (bset-written (temp next)))))))

(defun convert-from-bsets (program-graph univ)
  (iterate ((block (scan program-graph)))
    (setf (live block)
          (bset->list (bset-live (temp block)) univ))
    (setf (temp block) nil)))
```

Figure 3: Live variable analysis.

If this new estimate is different from the current estimate of what variables are live, then the estimate is changed. In addition, if the estimate is changed, `perform-relaxation` has to make sure that all of the predecessors of the current block will be examined to see if the new estimate for the current block requires that their live estimates be changed. This is done by adding each predecessor onto the list `to-do` unless it is already there. As soon as the estimates of liveness stop changing, the computation stops.

Summary. The function `determine-live` is a particularly good example of the way series expressions are intended to be used in two ways. First, series expressions are used in a number of places to express computations which would otherwise be expressed less clearly as loops or less efficiently as sequence function expressions. Second, the main relaxation algorithm is expressed as a loop. This is done, because neither optimizable series expressions (nor Common Lisp sequence function expressions) lend themselves to expressing the relaxation algorithm. This highlights the fact that series expressions are not intended to render iterative programs entirely obsolete, but rather to provide a greatly improved method for expressing the vast majority of loops.

Setting Up the Series Macro Package

The Series macro package is written in standard Common Lisp and has been tested in several different versions of Common Lisp. To use the Series macro package, the file containing it has to be loaded.

The source for the Series macro package can be obtained over the INTERNET by using FTP. Connection should be made to the `TRIX.AI.MIT.EDU` machine (INTERNET number 128.52.32.6). Login as 'anonymous' and copy the files shown below. It is advisable to run the tests in `stest.lisp` after compiling the Series macro package for the first time on a new system. A comment at the beginning of the file describes how to run the tests.

files on `TRIX.AI.MIT.EDU`,

`/com/ftp/pub/series/s.lisp`	source code
`/com/ftp/pub/series/stest.lisp`	tests
`/com/ftp/pub/series/sdoc.txt`	brief documentation

As the series macro package is being made available free of charge, it is being distributed *as is*, with no warrantee of any kind either expressed or implied. Neither the author nor MIT accepts liability of any kind. In addition, if you wish to use the Series macro package for anything other than your own experimental use, you will have to get a license from MIT. Information about obtaining a non-exclusive, royalty-free license can be obtained by sending a message to "`dick@ai.mit.edu`".

The functions and forms discussed in this manual are defined in the package `"series"`. To make these names easily accessible, you must *use* the package `"series"`. The most convenient way to do this is to call the function `series::install`, which also sets up some additional features of the series macro package. The examples in this manual assume that the form `(series::install)` has been evaluated.

- `series::install &key (:pkg *package*) (:macro T)`
  `(:shadow T) (:remove nil)` ⇒ `T`

Calling this function sets up Series for use in the package `:pkg`. The argument `:pkg` can either be a package, a package name, or a symbol whose name is the name of a package. It defaults to the current package.

The package `"series"` is used in `:pkg`. If the `:macro` argument is not `nil`, the `#` macro character syntax `#Z` and `#M` is set up. If the `:shadow` argument is not `nil`, the symbols `series::let`, `series::let*`, `series::multiple-value-bind`, `series::funcall`, and `series::defun` are shadowing imported into `:pkg`. These forms are identical to their standard counterparts, except that they support various features of the Series macro package. When shadowing is not done, you have to explicitly use `series::let`, `series::let*`, and `series::multiple-value-bind` when binding series in an expression you want optimized; `series::funcall` when funcalling a series function you want optimized; and `series::defun` when defining a series function with the declaration `optimizable-series-function`.

If `:remove` is not `nil`, the effects of having previously installed the Series macro package are undone. In particular, the package is unused and any shadowing is undone. However, any changes to the readtable are left in place.

Acknowledgments.

The Series macro package has benefited from the suggestions of a number of people. In particular, A. Meyer, C. Perdue, C. Rich, D. Wile, Y. Feldman, D. Chapman, Y. Tan, and P. Anagnostopoulos made suggestions that led to significant improvements. Financial Support for the development of the Series macro package was provide by the National Science Foundation, the Advanced Research Projects Agency of the Department of Defense, and the IBM, NYNEX, Siemens, and Microelectronics and Computer Technology corporations.

Bibliography

[1] A. Aho, J. Hopcraft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading MA, 1974.

[2] J. Orwant, *Support of Obviously Synchronizable Series Expressions in Pascal*, MIT/AI/WP-312, September 1988.

[3] R. Waters, "A Method for Analyzing Loop Programs", *IEEE Trans. on Software Engineering*, 5(3):237–247, May 1979.

[4] R. Waters, "Using Obviously Synchronizable Series Expressions Instead of Loops" *Proc. 1988 International Conference on Computer Languages*, 338–346, Miami FL, IEEE Computer Society press, October 1988.

[5] R. Waters, *Optimization of Series Expressions: Part I: User's Manual for the Series Macro Package*, MIT/AIM-1083, January 1989.

[6] R. Waters, *Optimization of Series Expressions: Part II: Overview of the Theory and Implementation*, MIT/AIM-1083, January 1989.