Automatic Transformation of Series Expressions into Loops

RICHARD C. WATERS
MIT Artificial Intelligence Laboratory

The benefits of programming in a functional style are well known. In particular, algorithms that are expressed as compositions of functions operating on sequences/vectors/streams of data elements are easier to understand and modify than equivalent algorithms expressed as loops. Unfortunately, this kind of expression is not used anywhere near as often as it could be, for at least three reasons: (1) most programmers are less familiar with this kind of expression than with loops, (2) most programming languages provide poor support for this kind of expression; and (3) when support is provided, it is seldom efficient

In any programming language, the second and third problems can be largely solved by introducing a data type called *series*, a comprehensive set of procedures operating on series, and a preprocessor (or compiler extension) that automatically converts most series expressions into efficient loops. A set of restrictions specifies which series expressions can be optimized If programmers stay within the limits imposed, they are guaranteed of high efficiency at all times.

A Common Lisp macro package supporting series has been in use for some time. A prototype demonstrates that series can be straightforwardly supported in Pascal.

Categories and Subject Descriptors· D.1.1 [**Programming Techniques**]· Applicative (Functional) Programming; D.3.2 [**Programming Languages**]: Language Classifications—*Pascal, Lisp*; D.3.3 [**Programming Languages**]: Language Constructs—*control structures*; D 3 4 [**Programming Languages**]: Processors—*preprocessors, optimization*; E.1 [**Data Structures**]· —*lists*; I 2.2 [**Artificial Intelligence**]: Automatic Programming—*program transformation*

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Sequences, Vectors, Streams, Series

1. SEQUENCE EXPRESSIONS

The mathematical term *sequence* refers to a mapping from the non-negative integers (or some initial subset of them) to values. Whether called sequences [5, 37], vectors [24, 32, 33, 37], lists [37], streams [6, 20, 26, 31], sets [36], generators [19, 49], or flows [35], data structures providing complete (or partial) support for mathematical sequences are ubiquitous in programming.

The most common use for sequence data structures is as mutable aggregate storage. Almost every programming language provides operations for accessing and altering the elements of at least one such structure.

Author's address: MIT Artificial Intelligence Laboratory, 545 Technology Square, Cambridge MA 02139.

Sequences have another use that is potentially just as important and yet is supported by only a few languages: Most algorithms that can be expressed as loops can also be expressed as functional expressions manipulating sequences. For example, consider the problem of computing the sum of the squares of the odd numbers in a file `Data`. This can be done using a loop as shown in the following Pascal [25] program.

```
type FileOfInteger = file of Integer;

function FileSumLoop (var Data: FileOfInteger): Integer;
  var Sum: Integer;
begin
  Reset(Data);
  Sum := 0;
  while not Eof(Data) do
    begin
      if Odd(Data^) then Sum := Sum+Sqr(Data^);
      Get(Data)
    end;
  FileSumLoop := Sum
end
```

Alternatively, the sum of the squares of the odd numbers in the file can be computed using the sequence expression shown below. This expression assumes that four subroutines have been previously defined: `CollectSum` computes the sum of the elements of a sequence; `MapFn` computes a sequence from a sequence by applying the indicated function to each element of the input; `ChooseIf` selects the elements of a sequence that satisfy a predicate; and `ScanFile` creates a sequence of the values in a file.

```
function FileSum (var Data: FileOfInteger): Integer;
begin
  FileSum := CollectSum(MapFn(Sqr, ChooseIf(Odd, ScanFile(Data))))
end
```

For those who are not accustomed to functional programming, the greater familiarity of the program `FileSumLoop` may make it appear preferable. However, the program `FileSum` has two important advantages. First, the patterns of computation that are mixed together in the loop in `FileSumLoop` are pulled apart. Second, each of these subcomputations is distilled into a subroutine. For example, the pattern of initializing a variable to `0` and then repetitively accumulating a result by addition is distilled into `CollectSum`.

Because the subcomputations are pulled apart, they can be understood in isolation. The action of the expression as a whole is the composition of the actions of the subcomputations. This makes `FileSum` more self-evidently correct than `FileSumLoop`. The separation of the subcomputations also means that they can be altered in isolation. This makes `FileSum` easier to modify. The distillation of the subcomputations into subroutines makes `FileSum` shorter and enhances the reusability of the subcomputations. It also enhances reliability in two ways. Since the subcomputations are being explicitly reused instead of regenerated by the programmer from memory, there is less chance of error. In addition, since each subroutine can be reused many times, it is practical to work very hard to ensure that the algorithm used in the subroutine is robust.

Unfortunately, there are two problems that inhibit most programmers from writing programs like `FileSum`. First, most programming languages provide very few predefined procedures that operate on sequences as aggregates, rather than merely operating on their individual elements. Second, even in languages such as APL and Common Lisp, where a wide range of sequence operations are available, sequence expressions are typically so inefficient (2 to 10 times slower than equivalent loops), that programmers are forced to use loops whenever efficiency matters.

The primary source of inefficiency when evaluating sequence expressions is the physical creation of intermediate sequence structures. This requires a significant amount of space overhead (for storing elements) and time overhead (for accessing elements and paging). The key to solving the efficiency problem is the realization that it is often possible to transform sequence expressions into a form where the creation of intermediate sequence structures is eliminated. For example, it is straightforward to transform the expression in `FileSum` into the loop in `FileSumLoop`.

A transformational approach to the efficient evaluation of sequence expressions has been used in a number of contexts. For example, it is used by optimizing APL compilers [12, 22], Wadler's Listless Transformer [39, 40] which can improving the efficiency of programs written in a Lisp-like language, and Bellegarde's transformation system [7, 8] which can improve the efficiency of programs written in the functional programming language FP [5]. In addition, Goldberg and Paige [18] have shown that the transformational approach can be used to improve the efficiency of data base queries.

Unfortunately, it is not possible to completely transform every sequence expression into an efficient loop. There are two basic ways to deal with this problem. First, one can hide the issue from the programmer and simply transform what can be transformed. Second, one can develop a set of restrictions defining what can be transformed and communicate with the programmer about the transformability of individual sequence expressions.

The hidden approach, which is followed by all the systems above, has the advantage that programmers can benefit from increased efficiency in some situations without having to think about efficiency in any situation. However, it makes it difficult for programmers to think about efficiency when they want to, because they have no way of knowing for sure whether a given sequence expression will be completely transformed. This is significant, because sequence expressions typically remain quite inefficient if any part of them fails to be transformed. In addition, quite simple changes in an algorithm often suffice to change an untransformable expression into a transformable one. As a result, it is not really a favor to hide the issue of transformability from programmers.

The most important contribution of the research reported here is a set of restrictions that can serve as a basis for the communicative approach to the transformation of sequence expressions into loops. As discussed in Section 2, these restrictions identify a class of *optimizable* sequence expressions that can always be completely transformed. The restrictions are novel in two ways. First, they are explicit. While every system that optimizes sequence expressions implicitly embodies some set of restrictions, the restrictions used are not explicit except in the work of Wadler [41]. Second, the restrictions in Section 2 are less strict than most other sets of restrictions. In particular, they are less strict than Wadler's restrictions.

Sections 3–6 show how the communicative approach can be used to add comprehensive and efficient support for sequence expressions into any programming language. This is done by adding a new sequence data type called *series* and a preprocessor that can transform optimizable series expressions into loops. The support for series utilizes the optimizability restrictions in two ways. One of the key restrictions is enforced by selecting the set of predefined series operations so that the restriction cannot be violated. The rest of the restrictions are explicitly checked by the preprocessor. Non-optimizable expressions are flagged with warning messages and left unoptimized. If users take the time to make each series expression optimizable, they can have complete confidence that every series expression is efficient. This is facilitated by the fact that simple series expressions that only use each series once can always be optimized.

Section 3 presents the series data type and a broad suite of associated functions. Currently, the most comprehensive support for series is in Common Lisp. This implementation [47, 48] is presented in Section 4, along with an extended Lisp example showing how series expressions can be used. A prototype implementation [29, 46] shows that series expressions can also be added into Pascal. This implementation is presented in Section 5, along with an extended Pascal example of how series expressions can be used. Readers are encouraged to focus on whichever of Section 4 or 5 discusses the most familiar language.

Section 6 presents the algorithms used to transform optimizable series expressions into loops. It should be noted that these algorithms are not capable of optimizing expressions computing series of series. However, it should also be noted that series of series are not necessary when expressing looping algorithms as series expressions. In particular, while algorithms involving nested loops can be expressed as expressions computing series of series, they do not have to be expressed that way.

Section 7 concludes by comparing series expressions with related concepts. The comparison includes both other implementations of sequences and other approaches to expressing loops in ways that are easy to understand and modify.

1.1 Getting Rid of Loops

To fully appreciate the practical impact of series expressions in general and optimizable ones in particular, one must return to the perspective of sequence expressions as a notational variant for loops. The program `FileSum` is an example based on the Pascal implementation of series. The series expression in it is optimizable and is transformed into a loop essentially identical to the one in `FileSumLoop`. As a result, it is not merely the case that `FileSumLoop` and `FileSum` compute the same result using the same abstract algorithm; the two programs denote exactly the same detailed computation. Using the expression in `FileSum`, one gains the advantages of functional form without paying any price in terms of efficiency or anything else, because there is no change in anything other than the form.

The value of optimizable series expressions as an alternate notation for loops is directly related to the percentage of loops that can be profitably replaced by them. Any loop can be expressed as an optimizable series expression by converting the subcomputations used in it into series operations and composing them together. (At worst, the entire loop becomes a single series operation.) The value of doing this depends on how many fragments the loop can be decomposed into and how many of these fragments correspond to familiar computations. In general, the change is advantageous as long as there is at least one familiar fragment, because at the least, there is value in separating the familiar from the unfamiliar.

An informal study [42] revealed that approximately 80% of the loops programmers typically write are constructed solely by combining just a few dozen familiar looping fragments. (A somewhat similar study is reported in [16].) Experience with the Lisp implementation of series indicates that at least 95% of loops contain some familiar computation. Given this, the practical benefit of optimizable series expressions can be summarized as follows:

Optimizable series expressions are to loops
as structured control constructs are to gotos.

Structured control constructs (`if...then...else`, `case`, `while ..do`, etc.) are not capable of expressing anything that cannot be expressed using gotos. In addition, there are probably a few algorithms for which the use of gotos is preferable. Nevertheless, in almost every situation, structured control constructs are much better to use than gotos. They are better, not because they allow more algorithms to be expressed, but because they allow the same algorithms to be expressed in a way that is much easier to understand and modify.

Optimizable series expressions have exactly the same advantage. They do not allow algorithms to be expressed that cannot be expressed as loops. However, they allow algorithms to be expressed in a much better way. The only place where the analogy with structured control constructs breaks down is that while one can argue that gotos are never needed, there are definitely some algorithms that can be expressed better as loops than as optimizable series expressions.

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 optimizable 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.

2. OPTIMIZABLE SEQUENCE EXPRESSIONS

As noted above, the primary source of inefficiency when evaluating sequence expressions is the creation of physical intermediate sequences. There are two aspects to this. First, a subexpression may waste time computing sequence elements that are not used by the rest of the expression. Second, even when all the elements of an intermediate sequence are used, constructing a physical data structure containing the elements wastes a significant amount of time and space.

The problem of computing unused sequence elements can be overcome by using lazy evaluation [17] to ensure that sequence elements are not computed until they are needed. (This also makes it easy to support unbounded sequences.) However, lazy evaluation does little to reduce the waste associated with constructing physical intermediate sequences. In particular, in situations where the elements of a sequence are all used, lazy evaluation wastes time and does not save any space. Time is wasted, because coordination overhead is required to decide when to compute the elements. The same space is used, because a physical intermediate sequence is still created. (Each element has to be stored somewhere after it is computed; otherwise, a later reuse of the element would require recomputation.)

All of the waste associated with constructing physical intermediate sequences and much of the waste associated with computing unused sequence elements can be eliminated by *pipelining* the evaluation of a sequence expression.

Definition 1 (pipelined) The evaluation of a sequence expression E is *pipelined* if and only if the following two conditions hold for every sequence S computed by any subexpression of E. First, each element of S is computed at most once. Second, when an element is computed, it is used wherever it needs to be used and then discarded before any other element of S is computed.

The primary implication of Definition 1 is that, while some of the procedures called by E may buffer sequence elements within themselves, no buffering is needed when transferring sequence elements between the subexpressions of E. Rather, each sequence is transmitted one element at a time between the procedure that creates it and the procedures that use it.

Consider the program `CosMax` below (which like most of the examples in this paper, is written using the Pascal implementation of series discussed in Section 5). The first statement in the body of `CosMax` computes a sequence `Vals` of the numbers in a file `Data`. The second statement computes a sequence of scaled values by dividing each value by its cosine. The third statement computes the maximum of the scaled values.

```
function CosMax (var Data: FileOfReal): Real;
  var Vals,ScaledVals: series of Real;
begin
  Vals := ScanFile(Data);
  ScaledVals := MapFn(/, Vals, MapFn(Cos, Vals));
  CosMax := CollectMax(ScaledVals)
end
```

The body of `CosMax` can be evaluated in many ways, some of which are pipelined and some of which are not. The standard non-pipelined method of evaluation proceeds in four steps. The numbers in the file are read and stored in a physical intermediate data structure. The cosine of each number is computed and stored in a second physical intermediate data structure. The result of dividing each number by its cosine is stored in a third physical intermediate data structure. The maximum element of the third intermediate structure is determined.

In contrast, the standard pipelined method of evaluation operates on the input numbers one at a time as follows. Each time a number is read from the file, it is immediately divided by its cosine and the scaled value is used to update an accumulator keeping track of the maximum value encountered. Since neither the number, its cosine, nor the scaled valued is needed for later processing, they are discarded before the next number is read.

If the basic schedule for what to evaluate when can be determined at compile time (as opposed to at run time) then the pipelined evaluation of a sequence expression is as efficient as the evaluation of an equivalent loop. Unfortunately, pipelining (at compile time or at run time) is not always possible. For example, consider the program `NormalizedMax` below. This program is the same as `CosMax` except that each input number is divided by the sum of the input numbers, rather than by its own cosine. (`Series` creates a sequence indefinitely repeating the value of its argument.)

In `NormalizedMax`, pipelined evaluation is impossible, because the two uses of `Vals` place contradictory constraints on the way pipelined evaluation must proceed. In particular, pipelining requires that each time an element of `Vals` is computed, it must be used immediately—i.e., both as part of computing the sum and as the numerator of a division. However, the division cannot be performed until after the sum has been computed and the computation of the sum cannot be completed until after all the elements of `Vals` have been computed. As a result, one either has to save the elements of `Vals` in an intermediate data structure, or recompute them when it is time to do the divisions.

```
function NormalizedMax (var Data: FileOfReal): Real;
  var Vals,ScaledVals: series of Real;
begin
  Vals := ScanFile(Data);
  ScaledVals := MapFn(/, Vals, Series(CollectSum(Vals)));
  NormalizedMax := CollectMax(ScaledVals)
end
```

Given the existence of programs like `NormalizedMax`, it is clear that any system that supports pipelining can only do so for a restricted class of sequence expressions. Whatever the system, it is valuable for these restrictions to be made explicit. If in addition, the programmer is given feedback about which expressions fail to meet the restrictions, two further advantages are obtained. The programmer is given a clear picture of which expressions are efficient and which are not, and the programmer has the opportunity to change inefficient expressions so that they can be pipelined.

It would be nice to have a set of necessary and sufficient restrictions specifying exactly which sequence expressions can be pipelined. However, there are a number of reasons why a somewhat stricter set of restrictions is of greater pragmatic benefit. First, the restrictions must be associated with practical algorithms that can check whether the restrictions hold and can actually perform the pipelining. Second, the restrictions must be sufficiently straightforward that programmers can succeed in fixing expressions that violate them.

The primary contribution of the work presented here is a set of four restrictions that are practical and straightforward without being excessively strict. These restrictions define a class of *optimizable* sequence expressions that can be pipelined at compile time by transforming them into loops, using the algorithms in Section 6.

2.1 Straight-Line Expressions

In the interest of simplicity, optimizable sequence expressions are required to be straight-line computations, not subject to any conditional or looping control flow. While it is likely that looping control flow in sequence expressions must be prohibited, simple conditional control flow could probably be allowed. However, this would complicate the algorithms in Section 6 in a number of ways. Conveniently, much of what can be done using conditional control flow can be done using sequence operations like `ChooseIf` instead.

Restriction 1 (straight-line) Optimizable sequence expressions must be straight-line computations.

One aspect of the simplicity engendered by the straight-line restriction is that it allows sequence expressions to be represented using simple data-flow graphs. For example, Figure 1 depicts the data-flow graph corresponding to the body of the program `NormalizedMax`. In the figure, procedure calls are represented by boxes. The inputs and outputs of each procedure are represented by dots on the left and right edges (respectively) of the boxes. Data flow is represented by arrows between the ports. Simple arrows indicate the flow of sequences. Cross hatched arrows indicate the flow of other values. Because the only expressions of interest are

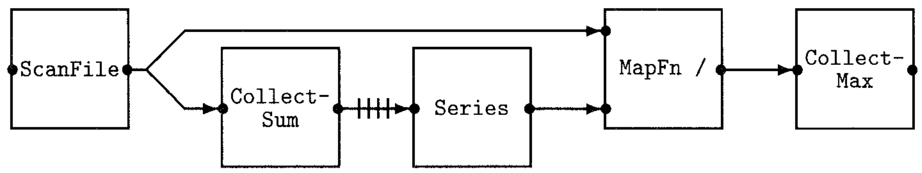

Fig. 1. The sequence expression in NormalizedMax.

straight-line ones, there is no need to consider control flow.

2.2 Static Analyzability

As with most other optimization processes, it is not possible to pipeline a sequence expression at compile time, unless it can be determined at compile time exactly what computation is being performed.

Restriction 2 (static analyzability) Optimizable sequence expressions must be *statically analyzable*. A sequence expression is statically analyzable if and only if each sequence value is computed and consumed by explicit calls in the expression on previously defined sequence procedures whose bodies are statically analyzable.

Static analyzability guarantees that it will always be clear exactly how each sequence is being computed and used. Unfortunately, it also implies that every sequence computed in an optimizable sequence expression must be used solely inside the same sequence expression. Further, a sequence cannot be stored in a sequence or in any other kind of data structure. Arbitrary storage of sequences in data structures is bound to block compile-time pipelining. However, certain limited cases could be allowed. For instance, one can sometimes determine how a sequence contained in another sequence is being computed. The practicality of this has been demonstrated by compilers for APL [12], Hibol [35], and Model [33].

The static analyzability restriction allows user-defined sequence procedures to be used. However, it requires that each sequence procedure be defined before its first use and that this definition be available to the compiler at the point of use. This is required by any process that calls for the inline compilation of procedure calls.

2.3 The Preorder Restriction

Suppose that a procedure call $\mathcal{F}$ uses a sequence computed by another procedure call $\mathcal{G}$. For the computation of these two calls to be pipelinable, two conditions must be satisfied. First, it must be the case that the sequence elements are created and consumed one at a time. Second, the elements must be consumed in the same order they are created. A good way to ensure that this will always be the case is to pick some fixed order and require that every procedure process every sequence one element at a time in that order. Given a desire to support unbounded sequences, a good order to pick is the natural order of the elements starting with the first.

Restriction 3 (preorder) Every procedure called by an optimizable sequence expression must be *preorder*. A procedure is preorder if and only if it processes the elements of each of its sequence inputs and outputs one at a time in ascending order starting with the first element.

It is important to note that the restriction above applies to procedures, rather than functions. In this paper, the word *procedure* is used to refer to a particular algorithm that implements a mathematical function, while the word *function* is reserved for referring to a mathematical function. The term *preorder* applies to procedures, rather than functions, because it is a property of the way a computation is performed, not of the mathematical relationship between the input and the output.

For example, consider computing the sum of the elements in a sequence. Summation can be implemented in many different ways, some of which are preorder and some of which are not. The procedure `CollectSum` operates in preorder, reading the sequence elements one at a time and adding each one into a running sum. However, one could choose to read the elements in reverse order, or two at a time, or in some other non-preorder way.

Any sequence function can be implemented as a preorder procedure. Therefore, the preorder restriction does not limit what functions can be used, but rather only how they can be implemented. In addition, most of the time, a preorder implementation is no less efficient than any other implementation. However, preorder processing sometimes requires more buffering of input elements than would otherwise be required. This issue is discussed further at the end of this section.

2.4 The On-Line Cycle Restriction

Before looking at the last restriction optimizable sequence expressions must satisfy, it is useful to consider the following property.

Definition 2 (on-line and off-line) An input or output port of a procedure is *on-line* if and only if it reads or writes a sequence and operates in lock step with all the other on-line ports of the procedure as follows: The initial element of each on-line input is read, then the initial element of each on-line output is written, then the second element of each on-line input is read, then the second element of each on-line output is written, and so on for the rest of the elements. If the sequence ports of a procedure are all on-line, the procedure as a whole is on-line. If a port or procedure is not on-line, it is off-line.

Definition 2 extends the standard definition of the term *on-line* [1, 23] so that it applies to individual ports as well as whole procedures. Like the definition of the term preorder, Definition 2 applies to procedures, rather than functions.

For example, consider the operation of mapping a procedure $\mathcal{F}$ over the elements of one or more sequences. Mapping can be implemented in many different ways, some of which are on-line and some of which are not. The procedure `MapFn` operates in an on-line way, reading the first element of each input sequence, applying $\mathcal{F}$ to these values, writing the result as the first output element, and so on until the inputs are exhausted. However, one could choose to implement mapping in an off-line way by accelerating the reading of one or more inputs or delaying the writing of the output.

There is a close relationship between on-line processing and preorder processing. In particular, every on-line procedure is preorder and every preorder procedure that has only one sequence input or output is trivially on-line. However, when more than one sequence port is involved, on-line processing is more constrained than preorder processing. In particular, while every mathematical function can be implemented as a preorder procedure, there are many functions that cannot be implemented as on-line procedures.

For example, consider the operation of selecting the elements of a sequence that satisfy a predicate $\mathcal{P}$. It is impossible for this to be implemented as an on-line procedure. The problem is that since some of the elements of the input do not become elements of the output, the writing of the elements of the output cannot remain in lock step with the reading of the elements of the input—as soon as an input element is skipped, the output gets out of phase.

Consider a related function that takes two sequences, a sequence of boolean values and some other sequence, and returns the elements of the second sequence that correspond to true values in the first sequence. It is also impossible for this function to be implemented as an on-line procedure. However, note that the output is the only problem. There is no difficulty in having the two inputs be on-line. Definition 2 specifies what it means for individual ports to be on-line in order to capture the fact that functions like the one proposed in this paragraph can be implemented in ways that are *partly* on-line.

Returning to a discussion of pipelinability in general, and the example programs `CosMax` and `NormalizedMax` used in the beginning of this section in particular, the concept of on-line processing is the key to understanding why the computation in `CosMax` can be pipelined while the computation in `NormalizedMax` cannot.

Recall that the evaluation of the sequence expression in `NormalizedMax` (see Figure 2) cannot be pipelined, because pipelining imposes contradictory constraints on the way the output of `ScanFile` must be computed. The procedure `CollectSum` requires that all the elements of its input be produced before the sum can be returned and `Series` requires that its input be available before it can start producing its output. However, `MapFn` requires that the first element of its two sequence inputs be simultaneously available. For pipelining to work, this implies that the first element of the output of `Series` (and therefore the output of `CollectSum`) must be available before the second element of the output of `ScanFile` is computed. Unfortunately, if the output of `ScanFile` contains more than one element, this is impossible.

The essence of the problem above can be broken down into two parts. First, there is a cycle of interacting constraints. Second, these constraints are inconsistent with each other.

The expression in Figure 2 is associated with a cycle of interacting constraints, because it contains a non-directed data-flow cycle. In particular, the data-flow arcs connecting `ScanFile`, `CollectSum`, `Series`, and `MapFn` form a non-directed cycle.

If an expression does not contain any non-directed data flow cycles, then all of the constraints on pipelining are completely independent and cannot be inconsistent. However, while the presence of a data flow cycle implies that the constraints are not independent, it does not necessarily imply that they are inconsistent. For example, the sequence expression in `CosMax` (shown in Figure 3) is pipelinable even though it contains a non-directed data flow cycle. It is possible to guarantee that data flow cycles will not lead to inconsistencies by requiring that they be *on-line.*

Restriction 4 (on-line cycle) Every non-directed data flow cycle in an optimizable sequence expression must be *on-line.* A non-directed data-flow cycle is on-line if and only if, whenever it passes through two ports of the same procedure call, the two ports are both on-line.

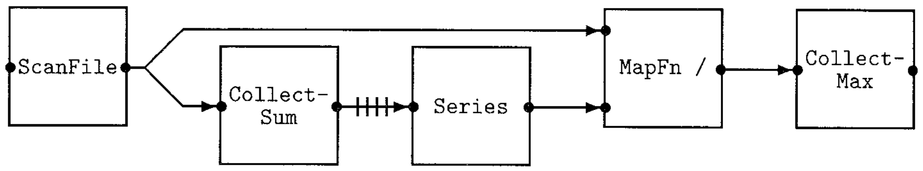

Fig. 2. The sequence expression in NormalizedMax.

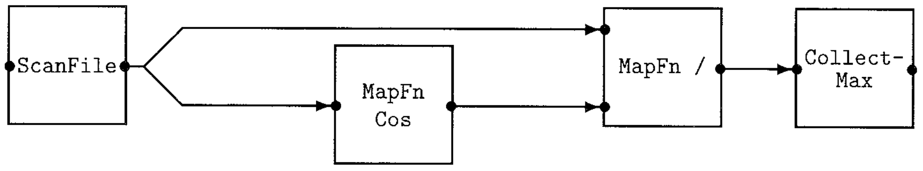

Fig. 3. The sequence expression in CosMax.

The non-directed data-flow cycle in Figure 3 is on-line, because the input and output of `MapFn Cos` along with both inputs of `MapFn /` are on-line. It does not matter that the output of `ScanFile` is on-line, because the cycle does not pass through any other port of `ScanFile`. In contrast, the non-directed data-flow cycle in Figure 2 is off-line, both because the output of `CollectSum` is off-line and because the input of `Series` is off-line.

To understand why on-line cycles guarantee pipelinability, it is useful to think of the cycles in Figures 2 and 3 as consisting of two branches (an upper branch and a lower branch). If pipelining is to work, the processing of elements along the two branches must be synchronized. If the cycle is on-line, the lock-step synchronization each step of the way guarantees that each branch will be completely synchronized and therefore the two branches will balance. In contrast, if (as in Figure 2) one branch contains an off-line port while the other does not, one branch will be synchronized while the other is not, and the two branches will not be balanced. (In either case, it does not matter whether the output of `ScanFile` is on-line, because the relationship between the processing at this port and the processing at the other ports of `ScanFile` is not relevant to either branch of the cycle.)

The output of `CollectSum` is off-line because it is a non-sequence port. The program `PositiveMax` below illustrates that off-line sequence ports also block pipelining. This program is identical to `CosMax` (see Figure 3) except that `MapFn Cos` is replaced with `ChooseIf Positive`. The scaled numbers in `PositiveMax` are computed by dividing the *i*th element of `Vals` by the *i*th positive element of `Vals`.

```
function PositiveMax (var Data: FileOfReal): Real;
  var Vals,ScaledVals: series of Real;
  function Positive (X: Real): Boolean;
    begin Positive := X>0.0 end;
begin
  Vals := ScanFile(Data);
  ScaledVals := MapFn(/, Vals, ChooseIf(Positive, Vals));
  PositiveMax := CollectMax(ScaledVals)
end
```

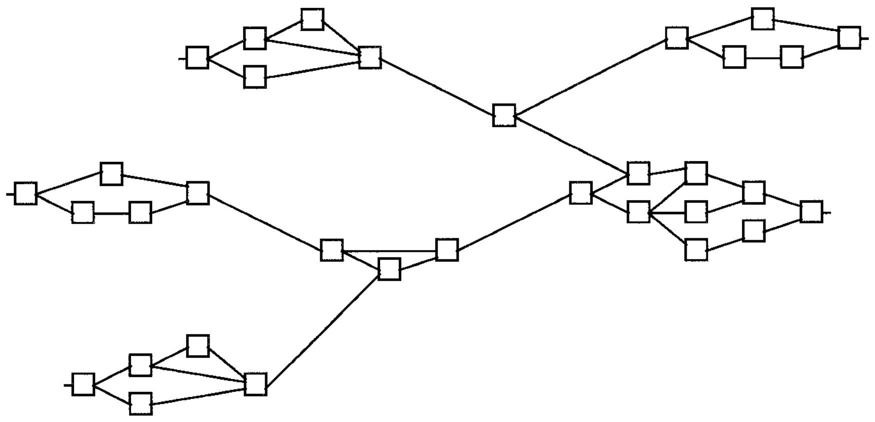

Fig. 4. The qualitative effect of the on-line cycle restriction.

As in the program `NormalizedMax`, the two branches of the computation cannot be reliably synchronized, because the output of `ChooseIf Positive` will get out of phase with `Vals` as soon as a non-positive element is encountered. In particular, suppose that the first and third elements of `Vals` are positive while the second is not. In this situation, the second output element of `ChooseIf` will not be available until after the third element of `Vals` has been computed. Depending on the input, the unbalanced delay in `PositiveMax` may be very small or even zero. However, compile-time pipelining is only possible when it is known that there will never be any unbalanced delay.

The on-line cycle restriction is stronger than it has to be, because it is possible for both branches of a cycle to contain off-line ports and therefore both be desynchronized and yet still balance, because they are identically desynchronized. However, the only common situation where balanced desynchronization occurs in practice is indirectly handled by the restrictions above. If the non-sequence output in the lower branch of the cycle in Figure 2 were balanced by a second non-sequence output in the upper branch, then it would be possible to view the expression as a whole as two separate expressions (one on the left and one on the right) which could be individually optimized.

The on-line cycle restriction forces optimizable sequence expressions to take on the qualitative form illustrated in Figure 4. In particular, the restriction forces a two-level structure of connectivity. At the bottom level, there are a number of clusters of procedure calls. Within each cluster, arbitrarily complex interconnection can be used; however, all of the ports participating in this data flow must be on-line. In contrast, the overall inputs and outputs of the on-line clusters can be off-line; however, the top-level data flow connecting the clusters cannot contain any non-directed data-flow cycles.

From a pragmatic perspective, the limits imposed by the on-line cycle restriction are softened by the fact that several of the most commonly used sequence procedures compute sequences from sequences in a completely on-line way. Nevertheless, the on-line cycle restriction is by far the most stringent of the restrictions on optimizable sequence expressions.

2 5 Summary of Restrictions

The four restrictions discussed above can be summarized as follows. A sequence expression is *optimizable* if and only if:

(1) It is a straight-line computation; and
(2) It is statically analyzable; and
(3) Every procedure called by it is preorder; and
(4) Every non-directed data-flow cycle in it is on-line.

The first restriction greatly simplifies the algorithms in Section 6, but is undoubtedly stronger than necessary. It is hoped that it will be weakened in the future. A restriction analogous to the second one is required for any optimization that is to be applied at compile time as opposed to run time. The third restriction ensures that isolated pairs of sequence procedure calls can be pipelined. The fourth restriction ensures that every optimizable sequence expression will have the form shown in Figure 4. Together, the restrictions ensure that the following divide-and-conquer approach can be used to pipeline an optimizable sequence expression.

If an optimizable sequence expression consists of a single on-line cluster, pipelining can be achieved by simply evaluating every procedure call in lock step, one element at a time. If there is more than one cluster, then (since the top-level data flow is cycle-free) the expression as a whole can always be divided into two non-overlapping subexpressions that are connected by a single data flow. The expression as a whole can be pipelined by pipelining the evaluation of the two subexpressions separately and using a simplified form of lazy evaluation to interleave the evaluation of the subexpressions in a pipelined fashion. The lazy evaluation is simplified because the method for determining which subexpression to evaluate when is very simple. The source subexpression needs to be evaluated when and only when the single destination subexpression needs to read a new value computed by it.

2.6 Obeying the Restrictions

In the implementations discussed in the following sections, the preorder restriction is implicitly enforced by ensuring that every predefined procedure is preorder and ensuring that there is no way to define a non-preorder procedure. (Note that the composition of two preorder procedures is preorder.)

The other three restrictions are explicitly checked. Whenever an expression satisfies these restrictions, the algorithms in Section 6 are used to transform the expression into an efficient loop. When they are not satisfied, a warning is issued. In the current Pascal implementation, these warnings are fatal errors. However, in the Lisp implementation, expressions that do not satisfy the restrictions are simply left as is and evaluated/compiled without optimization.

It is straightforward to follow the straight-line and static analyzability restrictions. However, the on-line cycle restriction is significantly more complex. Experience suggests that the best approach for programmers to take is to write expressions without worrying about the on-line cycle restriction and then fix any expressions for which warning messages are reported. The virtues of this approach are enhanced by the fact that simple expressions are very unlikely to violate any of the restrictions. In particular, it can be shown that if every sequence procedure in an expression has only one output and sequence outputs are not stored in variables, then the on-line cycle restriction cannot be violated.

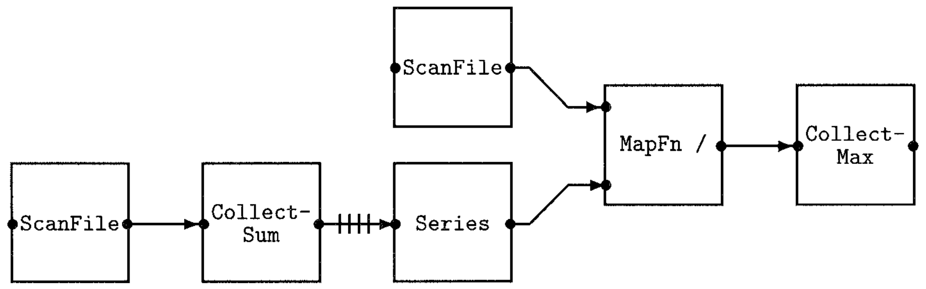

Fig. 5. The sequence expression in NormalizedMaxA.

Violations of the on-line cycle restriction can always be fixed by using code copying to break the offending cycle. For instance, the program `NormalizedMax` can be brought into compliance with the on-line cycle restriction by duplicating the call on `ScanFile` as in `NormalizedMaxA`. This breaks the cycle and converts the expression into one that is optimizable (see Figure 5).

```
function NormalizedMaxA (var Data: FileOfReal): Real;
  var ScaledVals: series of Real;
      Sum: Real;
begin
  Sum := CollectSum(ScanFile(Data));
  ScaledVals := MapFn(/, ScanFile(Data), Series(Sum));
  NormalizedMaxA := CollectMax(ScaledVals)
end
```

It would be possible to automatically introduce code copying to resolve conflicts with the on-line cycle restriction. However, this can lead to significant inefficiencies. It is better to leave it up to the programmer to figure out how to fix conflicts. For example, the procedure `NormalizedMax` can be brought into compliance more efficiently, by realizing that the operations of computing the maximum and dividing by the sum commute as shown in `NormalizedMaxB`.

```
function NormalizedMaxB (var Data: FileOfReal): Real;
  var Vals: series of Real;
begin
  Vals := ScanFile(Data);
  NormalizedMaxB := CollectMax(Vals)/CollectSum(Vals)
end
```

This example brings up an important secondary goal underlying the restrictions presented here. This goal is to make it easy for programmers to reliably tell which expressions are efficient and which are not. It would be better to make every expression efficient. However, given that this is not possible, programmers need accurate information in order to decide what to do.

2.7 Other Approaches to Restrictions

The four optimizability restrictions are the result of research going back thirteen years. The basic concept of representing common looping subcomputations as operations on sequences was present in some of the earliest work on the Programmer's Apprentice [42, 34]. The first attempt to state a formal set of restrictions appears in [43, 44]. An intermediate set of restrictions is presented in [45].

Optimizability restrictions are implicit in all the work on sequence expression optimizers. However, these restrictions are typically implicit in the way the optimizers work, rather than being explicitly stated. The only other research featuring explicit restrictions is that of Wadler [41].

Wadler's work differs from the work presented here in three ways. First, while Wadler's restrictions imply the ones presented here, they are needlessly more limiting. Second, he uses his restrictions in a different way. Third, as discussed in the next subsection, the motivation behind his restrictions is different.

In [41] Wadler only addresses the situation where a sequence procedure with a single output is composed with a second sequence procedure. His results can be straightforwardly generalized to the case of statically-analyzable, straight-line expressions, in which each procedure has only one output and no output value is used more than once. However, they are not applicable to more complex situations.

The limitation to statically-analyzable, straight-line expressions is no different from the limitations presented here. However, the implicit requirement that each procedure have only one output and no output be used more than once is significantly more limiting. It has the effect of outlawing off-line non-directed data-flow cycles, however, it goes way beyond this by outlawing all non-directed data-flow cycles. Since it is often possible to evaluate a sequence expression very efficiently even though it contains non-directed data-flow cycles, this is an overly severe restriction. It is unreasonable (both from the point of view of readability and efficiency) to require that every intermediate value that is used twice be computed twice.

The restriction that Wadler explicitly states (i.e., that procedures must be *preorder listless*) is basically equivalent to the preorder restriction stated here, except that, as discussed in the next subsection, it also requires that each sequence procedure operate using a bounded amount of internal storage. (Wadler's definition of the term preorder by itself is different from the one used here.)

Wadler uses his restrictions in the background to identify situations where computations involving a pre-existing data type (lists) can be optimized. In contrast, the approach used here places Restrictions 1–4 in the foreground as a primary feature of a new data type. This more active approach is followed for three reasons. First, since lists cannot represent unbounded sequences, focusing on lists unduly limits the kind of procedures that can be expressed. Second, lists (and vectors) have evolved a style of use and a suite of associated operations that are appropriate for that use. These work well for their intended use, but are not as useful as they could be from the perspective of writing efficient sequence expressions. Developing a new data type makes it possible to create a new suite of operations that is more appropriate for this purpose. Third, an important part of the approach being advocated here is the error messages that report unoptimizable expressions. These are important if programmers are to achieve efficiency. However, they would be irritating and counterproductive if they were constantly being reported for list expressions that the programmers did not intended to be optimizable. By adding a new data type, programmers can benefit from the restrictions when they choose to follow them, without being inhibited from using lists and vectors in standard ways.

2.8 Avoiding Unnecessary Storage

Wadler's work and the work presented here share the goal of evaluating sequence expressions efficiently. However, the basic framework of the two approaches is different, because they use different methods to assess whether efficient evaluation has been achieved.

Consider again the basic situation of an expression $\mathcal{F}(\mathcal{G}(x))$, where a procedure $\mathcal{F}$ is applied to a sequence computed by another procedure $\mathcal{G}$. Suppose further that $\mathcal{F}$ and $\mathcal{G}$ are individually as efficient as possible. Traditional methods of evaluating $\mathcal{F}(\mathcal{G}(x))$ are nevertheless inefficient, because they create an intermediate sequence object representing the value of $\mathcal{G}(x)$.

Wadler's work and the work presented here use the same basic two-step approach for eliminating the creation of intermediate sequences. First, they place limits on the way procedures can operate (i.e., by requiring them to be preorder). This forces you to use preorder procedures $\mathcal{F}'$ and $\mathcal{G}'$ that compute the same results as $\mathcal{F}$ and $\mathcal{G}$, but which may be individually less efficient. Second, they provide algorithms for combining $\mathcal{F}'$ and $\mathcal{G}'$ at compile time into a joint procedure $\mathcal{F}'\circ\mathcal{G}'$ that does not cause the creation of an intermediate sequence. A key question in both approaches is how is one to assess whether $\mathcal{F}'\circ\mathcal{G}'(x)$ is really more efficient than $\mathcal{F}(\mathcal{G}(x))$?

The work presented here uses pipelining as its test of efficiency. The basic idea is that if $\mathcal{F}'\circ\mathcal{G}'(x)$ is pipelined, then the resources used to transmit values from the computation associated with $\mathcal{G}'$ to the computation associated with $\mathcal{F}'$ is reduced to almost zero. If $\mathcal{F}'$ and $\mathcal{G}'$ are as efficient as $\mathcal{F}$ and $\mathcal{G}$, this guarantees that $\mathcal{F}'\circ\mathcal{G}'(x)$ will be significantly more efficient than $\mathcal{F}(\mathcal{G}(x))$. Unfortunately, the mere fact that $\mathcal{F}'\circ\mathcal{G}'(x)$ is pipelined does not imply that that $\mathcal{F}'$ and $\mathcal{G}'$ are as efficient as $\mathcal{F}$ and $\mathcal{G}$. In fact, since $\mathcal{F}'$ and $\mathcal{G}'$ must be preorder, they are sometimes forced to be much less efficient than $\mathcal{F}$ and $\mathcal{G}$.

From an efficiency standpoint, the fundamental problem with preorder processing is that it sometimes requires input elements to be buffered in a way that is not required by other methods of computation. For example, consider computing a sequence that contains the same elements as an input sequence, but in reverse order. Reversal can be implemented very efficiently by a procedure that reads the input elements one at a time beginning with the last, while writing the output elements starting with the first. This procedure need not use any buffering of elements, however, it is not preorder. Since a preorder procedure must process both its inputs and outputs in order starting with the first element, any preorder implementation of reverse has to store all the input elements inside itself before it can begin writing the output elements.

Typically, preorder processing requires the buffering of inputs whenever the function being implemented calls for the reordering of the elements of an input sequence. Conveniently, reversal and sorting are the only common sequence functions where this is the case.

In the expression $\mathcal{F}(\mathcal{G}(x))$, if $\mathcal{F}$ implements reversal, then the preorder implementation $\mathcal{F}'$ must be so much less efficient than $\mathcal{F}$ that $\mathcal{F}'\circ\mathcal{G}'(x)$ cannot be any more efficient than $\mathcal{F}(\mathcal{G}(x))$. The simultaneous storage of the elements computed by $\mathcal{G}$ is merely moved from an intermediate sequence computed by $\mathcal{G}$ to a buffer inside $\mathcal{F}'$. While this situation does not violate the letter of the definition of pipelining, it violates the spirit of what is trying to be achieved.

Wadler uses a limit on the storage used by $\mathcal{F}'\circ\mathcal{G}'(x)$ as his test of efficiency. He requires that $\mathcal{F}'\circ\mathcal{G}'(x)$ operate using bounded internal storage—i.e., storage over and above the storage needed to represent the overall input and output. Since sequences are potentially unbounded in length, this guarantees that an intermediate sequence is not being created during the computation. The central theorem in [41] shows that if an additional restriction requiring that procedures use a bounded amount of internal storage is added to the preorder restriction, then $\mathcal{F}' \circ \mathcal{G}'(x)$ is guaranteed to use a bounded amount of internal storage.

An important virtue of the bounded internal storage restriction is that it goes beyond eliminating the physical representation of intermediate sequences by also ruling out the simultaneous storage of the elements of an intermediate sequence in a buffer inside a sequence procedure. Unfortunately, although the bounded internal storage restriction has the advantage of closing this loophole, it has a number of disadvantages.

The biggest problem is that the bounded internal storage restriction is really somewhat beside the point. The real question is whether a preorder sequence procedure $\mathcal{F}'$ is as efficient as its non-preorder counterpart $\mathcal{F}$. This correlates only very weakly with the question of whether $\mathcal{F}'$ uses bounded storage. It is true that if $\mathcal{F}'$ uses unbounded storage and $\mathcal{F}$ does not, then it is virtually certain that $\mathcal{F}'$ is much less efficient. However, if they both use bounded storage it is still possible for $\mathcal{F}'$ to be much less efficient. Further, if unbounded storage is necessary in both cases, $\mathcal{F}'$ could be just as efficient. In this situation, there would be no need to outlaw $\mathcal{F}'$, because while it is very likely expensive to compute, this expense is inherent rather than engendered by the optimization transformations being applied.

An additional problem with the bounded internal storage restriction is that there is no practical way to detect whether it is being obeyed. To start with, there is no useful way to determine whether user-defined sequence procedures require unbounded storage. As a result, user-defined sequence procedures would have to be outlawed. Worse than this, many of the most common higher-order sequence procedures are capable of using unbounded storage if given certain kinds of procedures as their arguments. Draconian limits would have to be set here as well.

Given the problems it causes and the fact that it has nothing to do with pipelinability *per se*, it was decided that it was better not to include the bounded internal storage restriction in the definition of an optimizable sequence expression. However, in the interest of overall efficiency, the set of predefined procedures (see Section 3) is confined to functions that have preorder implementations that are (nearly) as efficient as the best non-preorder implementation. In particular, the functions reversal and sorting are omitted. This gets things off on the right foot, without preventing users from defining whatever functions they feel are useful.

3. SERIES EXPRESSIONS

Vectors, lists, and other such data structures differ in how closely they model the mathematical concept of a sequence and in the range of associated operations. The series data type embodies a set of design decisions that combine full support for the mathematical concept of a sequence, a wide range of operations, and high efficiency. As illustrated in the next two sections, support for series can be straightforwardly added to any programming language. High efficiency is obtained by using a preprocessor or compiler extension like the one presented in Section 6.

Informally speaking, series are like vectors except that they can be unbounded in length. Formally, series are defined by the way they can be operated on. The remainder of this section presents an illustrative selection of these operations as mathematical functions. (See [9] for an in depth discussion of the mathematical properties of many of these functions when applied to finite sequences.) The next two sections present procedures implementing these functions in specific programming languages.

In the following, lowercase letters (r, x) denote arbitrary values, uppercase letters (R, S) denote series, and calligraphic letters ($\mathcal{F}$, $\mathcal{P}$) denote functions and predicates. The notation $\langle x, y, \ldots, z\rangle$ denotes a literal series containing the elements x, y, and so on up to z. The notation $r : R$ denotes the series formed by appending the element r to the front of the series R—e.g., $6 : \langle 7, 8\rangle = \langle 6, 7, 8\rangle$.

Series functions can be divided into three categories: *collectors* compute non-series values from series, *scanners* compute series from non-series values, and *transducers* compute series from series.

There are two kinds of collectors. Some collectors create an aggregate data structure containing the elements of a series, for instance, a hash table containing the series elements. Other collectors create a summary value computed by some formula from the elements of a series, for instance, their sum. The Lisp implementation of series supports 18 collectors. However, one of these (collection, shown below) can be used to define the rest.

Collection is a higher-order function—a function that takes functions as arguments. The first argument z is used as the initial value of an accumulator. The second argument is a binary function $\mathcal{F}$, which is used to combine the successive elements of an input series into the accumulator. (Typically, z is chosen to be a left identity of $\mathcal{F}$.) The final value of the accumulator is returned as the value of collection. The example computes the sum of a series.

$$\begin{aligned}\text{collection}(z,\ \mathcal{F},\ \langle\rangle) &= z\\ \text{collection}(z,\ \mathcal{F},\ r : R) &= \text{collection}(\mathcal{F}(z,\ r),\ \mathcal{F},\ R)\end{aligned}$$

$$\text{collection}(0,\ \lambda xy\,.\,x{+}y,\ \langle 1,2,3\rangle) = 6$$

There are also two kinds of scanners. Some scanners create a series of the elements in an aggregate data structure, for instance, a series of the elements in a hash table. Other scanners create a series based on some formula, for instance, the successive powers of a number. The Lisp implementation of series supports 15 scanners. However, one of these (the higher-order function scanning, see below) can be used to define the rest.

The first argument of scanning is an initial value z, which becomes the first element of the series created. The second argument is a stepping function $\mathcal{F}$, which is used to compute each output element after the first, from the previous element. The third argument is a predicate $\mathcal{P}$, which is used to determine where the series should end. The series created contains the elements z, $\mathcal{F}(z)$, $\mathcal{F}(\mathcal{F}(z))$, and so on, up to but not including the first value satisfying $\mathcal{P}$. If no value satisfies $\mathcal{P}$, the series is unbounded in length. The example computes the powers of 2 less than 100.

$$\text{scanning}(z,\ \mathcal{F},\ \mathcal{P}) = \begin{cases}\langle\rangle & \text{if } \mathcal{P}(z)\\ z : \text{scanning}(\mathcal{F}(z),\ \mathcal{F},\ \mathcal{P}) & \text{otherwise}\end{cases}$$

$$\text{scanning}(1,\ \lambda x\,.\,x{+}x,\ \lambda x\,.\,x{\geq}100) = \langle 1,2,4,8,16,32,64\rangle$$

Transducers are more complex than collectors or scanners. In particular, there is no one transducer that serves as a basis for the rest. Nevertheless, four key higher-order transducers support wide classes of common transduction operations.

Collecting is the same as collection except that it returns a series of partial results, rather than just a final value. The length of the output is the same as the length of the input. The example computes a series of partial sums.

$$\begin{aligned}\text{collecting}(z,\ \mathcal{F},\ \langle\rangle) &= \langle\rangle\\ \text{collecting}(z,\ \mathcal{F},\ r : R) &= \mathcal{F}(z,\ r) : \text{collecting}(\mathcal{F}(z,\ r),\ \mathcal{F},\ R)\end{aligned}$$

$$\text{collecting}(0,\ \lambda\, xy\,.\,x{+}y,\ \langle 1,2,3\rangle) = \langle 1,3,6\rangle$$

By far the most commonly used series function is mapping, which maps a function $\mathcal{F}$ over some number of series, producing a series of the results. Each element of the output is computed by applying $\mathcal{F}$ to the corresponding elements of the inputs. The length of the output is the same as the length of the shortest input. The example adds the corresponding elements in two series.

$$\begin{aligned}\text{mapping}(\mathcal{F},\ \ldots,\ \langle\rangle,\ \ldots) &= \langle\rangle\\ \text{mapping}(\mathcal{F},\ r_1 : R_1,\ \ldots,\ r_n : R_n) &= \mathcal{F}(r_1,\ \ldots,\ r_n) : \text{mapping}(\mathcal{F},\ R_1,\ \ldots,\ R_n)\end{aligned}$$

$$\text{mapping}(\lambda\, xy\,.\,x{+}y,\ \langle 1,2,3\rangle,\ \langle 4,5,6,7\rangle) = \langle 5,7,9\rangle$$

Truncating cuts off a series by testing each element with a predicate and discarding all the remaining elements as soon as an element satisfying the predicate is encountered. The example truncates a series before the first negative element.

$$\begin{aligned}\text{truncating}(\mathcal{P},\ \langle\rangle) &= \langle\rangle\\ \text{truncating}(\mathcal{P},\ r : R) &= \begin{cases}\langle\rangle & \text{if } \mathcal{P}(r)\\ r : \text{truncating}(\mathcal{P},\ R) & \text{otherwise}\end{cases}\\ \text{truncating}(\lambda\, x\,.\,x{<}0,\ \langle 0,3,2,-7,1,-1\rangle) &= \langle 0,3,2\rangle\end{aligned}$$

Choosing selects the elements of a series that satisfy a predicate. The example picks out the negative elements of the input.

$$\begin{aligned}\text{choosing}(\mathcal{P},\ \langle\rangle) &= \langle\rangle\\ \text{choosing}(\mathcal{P},\ r : R) &= \begin{cases} r : \text{choosing}(\mathcal{P},\ R) & \text{if } \mathcal{P}(r)\\ \text{choosing}(\mathcal{P},\ R) & \text{otherwise}\end{cases}\\ \text{choosing}(\lambda\, x\,.\,x{<}0,\ \langle 0,3,2,-7,1,-1\rangle) &= \langle -7,-1\rangle\end{aligned}$$

Mingling combines two series into one under the control of a comparison predicate. The comparison is performed as indicated to ensure that the combination will be stable—if two elements are not ordered by the comparison predicate, the element from R precedes the element from S in the result. The example shows the combination of two sorted series into a sorted result.

$$\begin{aligned}\text{mingling}(\langle\rangle,\ S,\ \mathcal{P}) &= S\\ \text{mingling}(R,\ \langle\rangle,\ \mathcal{P}) &= R\\ \text{mingling}(r : R,\ s : S,\ \mathcal{P}) &= \begin{cases} r : \text{mingling}(R,\ s : S,\ \mathcal{P}) & \text{if } \neg\mathcal{P}(s,\ r)\\ s : \text{mingling}(r : R,\ S,\ \mathcal{P}) & \text{otherwise}\end{cases}\\ \text{mingling}(\langle 1,3,7\rangle,\ \langle 2,4,5\rangle,\ \lambda\, xy\,.\,x{<}y) &= \langle 1,2,3,4,5,7\rangle\end{aligned}$$

In addition to the higher-order transducers above, some other transducers are important as well. Catenating appends two series end to end.

$$\begin{aligned}\text{catenating}(\langle\rangle,\ S) &= S\\ \text{catenating}(r:R,\ S) &= r:\text{catenating}(R,\ S)\end{aligned}$$

$$\text{catenating}(\langle 6,7,8\rangle,\ \langle 9,10\rangle) = \langle 6,7,8,9,10\rangle$$

Spreading is a quasi-inverse of choosing. Spreading takes a series of non-negative integers R and a series of values S, and creates a series containing the elements of S. In the output, the elements of S are spread out by interspersing them with copies of z. If the ith element of R is r, then the ith element of S is preceded by r copies of z. Taken together with the above example of choosing, the example below illustrates the relationship between choosing and spreading.

$$\begin{aligned}\text{spreading}(\langle\rangle,\ S,\ z) &= \langle\rangle\\ \text{spreading}(R,\ \langle\rangle,\ z) &= \langle\rangle\\ \text{spreading}(r:R,\ s:S,\ z) &= \underbrace{z:\ldots:z}_{r\text{ copies}}:s:\text{spreading}(R,\ S,\ z)\\ \text{spreading}(\langle 3,1\rangle,\ \langle -7,-1\rangle,\ 0) &= \langle 0,0,0,-7,0,-1\rangle\end{aligned}$$

Sectioning creates a subseries of the elements of its series input. The subseries begins with the element indexed by $m \geq 0$ and continues up to but not including the element indexed by $n \geq 0$. (The first element in a series has the index 0.) The n input is optional. If n is omitted or fails to be less than the length of the series input, the output continues until the input runs out of elements. The example extracts a section out of the middle of a series.

$$\begin{aligned}\text{sectioning}(\langle\rangle,\ m,\ n) &= \langle\rangle\\ \text{sectioning}(R,\ m,\ 0) &= \langle\rangle\\ \text{sectioning}(r:R,\ 0,\ n) &= r:\text{sectioning}(R,\ 0,\ n{-}1)\\ \text{sectioning}(r:R,\ m,\ n) &= \text{sectioning}(R,\ m{-}1,\ n{-}1)\end{aligned}$$

$$\text{sectioning}(\langle 1,1,2,2,3,3,4,4\rangle,\ 2,\ 5) = \langle 2,2,3\rangle$$

The function chunking is different from the ones above, because it can produce more than one output. It has the effect of breaking a series R into (possibly overlapping) chunks of width $w > 0$. Successive chunks are displaced $d > 0$ elements to the right, in the manner of a moving window. (For example, if $R = \langle 1,5,3,7\rangle$, $w = 2$, and $d = 1$ then the chunks are 1 5, 5 3, and 3 7.) Chunking produces w output series where the first output contains the first element of each chunk, the second output contains the second element of each chunk, etc. Thus, the ith chunk is composed of the ith elements of the w outputs. The number of chunks (and therefore the length of each output) is $\lfloor 1 + (|R|-w)/d \rfloor$, where $|R|$ denotes the length of the input series R.

$$\begin{aligned}\text{chunking}(1,\ d,\ \langle\rangle) &= \langle\rangle\\ \text{chunking}(1,\ d,\ r:R) &= r:\text{chunking}(1,\ d,\ \text{sectioning}(R,\ d{-}1))\\ \text{chunking}(w,\ d,\ R) &= S_1,\ \ldots,\ S_w \quad \text{where}\\ S_k &= \text{chunking}(1,\ d,\ \text{sectioning}(R,\ k{-}1,\ |R|{+}k{-}w))\end{aligned}$$

$$\text{chunking}(2,\ 1,\ \langle 1,5,3,7\rangle) = \langle 1,5,3\rangle,\ \ \langle 5,3,7\rangle$$
$$\text{mapping}(\lambda\, xy\,.\,(x{+}y)/2,\ \langle 1,5,3\rangle,\ \langle 5,3,7\rangle) = \langle 3,4,5\rangle$$

By itself, chunking may appear somewhat unusual, however, it is quite useful in combination with other transducers. For instance, the last part of the example shows how the results of chunking could be used as the basis for computing a moving average. (Programming languages differ in the mechanisms that could be used to channel the outputs of chunking to the inputs of mapping.)

The preceding list of functions can be viewed as a recommendation for the kind of functions that can profitably be supported in conjunction with a sequence data type. As discussed in Section 7, some languages (e.g., APL) support most of these functions; others (e.g., Pascal) support almost none of them.

The list of functions is also interesting for what it does not contain. To start with, it does not contain functions for accessing arbitrary series elements or altering the value of series elements. This reflects the fact that, unlike vectors or lists, series are not intended to be used as mutable data storage.

In addition, the list only includes functions that can be implemented just as efficiently as preorder procedures as any other way. (The only common functions ruled out by this criteria are ones like reversal and sorting that rearrange the order of the elements of their input.) The list also favors functions that can be implemented as on-line procedures, because these are more useful in optimizable expressions. (The only functions in the list that require off-line implementations are choosing, mingling, catenating, spreading, sectioning, and chunking.)

4. A COMMON LISP IMPLEMENTATION

Series can be added into essentially any programming language by adding an implementation of the series data structure and defining a set of procedures supporting the series functions in Section 3. The optimization of series expressions can be supported by a preprocessor (see Section 6). It is in the nature of Lisp, that both of these things are easy to do using a macro package. Such a macro package has been in regular use for a number of years and is generally available (see [47, 48]).

Series. In the Lisp implementation, unoptimized series are implemented lazily using closures. A series has a procedural part and a data part. The procedural part is a generator [19, 30] capable of computing the elements one by one. The data part records the elements computed so far.

The elements of a series are accessed using a second *accessing* generator that enumerates the elements in the data part of the series and then uses the procedural part to compute additional elements. Each time the accessing generator is called, it returns another element in the series. The accessing generator takes a procedure argument specifying what to do when the series runs out of elements.

The preceding two-level generation scheme ensures that elements are not computed until needed, no element is computed twice, and each user of a series can access all the elements. For those familiar with Lisp, Figure 6 illustrates the implementation of series data structures. The same basic implementation approach is used in the language Seque [20].

The closure implementation of series is effective and straightforward; however, it is not very efficient. No effort has been expended on producing a more efficient implementation, because the focus of series expressions is on the situations where

```
(setq first5           ; Implementation of ⟨1,2,3,4,5⟩.
      (let ((x 0))
        (list #'(lambda (at-end)
                  (if (< x 5) (setq x (+ x 1)) (funcall at-end))))))

(defun generator (s)   ; Returns a generator for the elements of a series.
  (let ((g (car s)))
    #'(lambda (at-end)
        (when (null (cdr s))
          (setf (cdr s)
                (block nil
                  (list (funcall g #'(lambda () (return T)))))))
        (if (not (eq (cdr s) T))
            (car (setq s (cdr s)))
            (funcall at-end)))))

(defun choose-if (p s) ; Implementation of choosing(P, S).
  (let ((gen (generator s)))
    (list #'(lambda (end-action)
              (loop (let ((x (funcall gen end-action)))
                      (if (funcall p x) (return x))))))))

(defun collect-sum (s) ; Implementation of collection(0, λxy.x+y, S).
  (let ((gen (generator s))
        (sum 0))
    (loop (let ((x (funcall gen #'(lambda () (return sum)))))
            (setq sum (+ sum x))))))

(collect-sum (choose-if #'oddp first5)) ⇒ 9
```

Fig. 6. Illustration of the Lisp implementation of unoptimized series.

they can be optimized, eliminating the physical representation of series altogether. In situations where optimization is impossible, it is usually better to represent a sequence as a vector or list than as a series.

The protocol for obtaining an accessing generator for the elements of a series and thence the elements themselves is not an exported part of the series implementation. Users must manipulate series using the procedures below. This is important in the interest of optimizability in general and static analyzability in particular.

Series procedures. The series functions described in Section 3 are all supported by Lisp procedures as shown in Figure 7. In addition, the # macro character syntax #Z(*x y* ... *z*) is provided for reading and printing literal series.

```
(catenate #Z(1 2) (choose-if #'oddp #Z(8 -7 -6 1))) ⇒ #Z(1 2 -7 1)
(subseries (mingle #Z(1 5 9) #Z(2 6 8) #'<) 2 4) ⇒ #Z(5 6)
(multiple-value-bind (xs ys) (chunk 2 1 #Z(1 5 3 7))
  (map-fn T #'(lambda (x y) (/ (+ x y) 2)) xs ys)) ⇒ #Z(3 4 5)
```

The higher-order procedures implementing scanning, collecting, mapping, truncating, and collection are extended so that they can accept multiple series arguments and produce multiple values. (Series of tuples could be used to get the same effect in any given situation. However, using multiple series values is usually more convenient and almost always more efficient than using tuples.)

The examples below illustrate the Lisp procedure scan-fn, which supports scanning. In the second example, a two-valued stepping procedure is used and two series are returned (an unbounded series of the natural numbers and a series of their partial sums). While scanning is in progress, two internal state values are maintained. The stepping procedure must accept as many values as it returns. Each of these

Series Function	Lisp Implementation
collection(z, $\mathcal{F}$, R)	`(collect-fn` *type* $\mathcal{Z}$ $\mathcal{F}$ $R_1 \ldots R_n$`)`
scanning(z, $\mathcal{F}$, $\mathcal{P}$)	`(scan-fn` *type* $\mathcal{Z}$ $\mathcal{F}$ $\mathcal{P}$`)`
collecting(z, $\mathcal{F}$, R)	`(collecting-fn` *type* $\mathcal{Z}$ $\mathcal{F}$ $R_1 \ldots R_n$`)`
mapping($\mathcal{F}$, $R_1, \ldots, R_n$)	`(map-fn` *type* $\mathcal{F}$ $R_1 \ldots R_n$`)`
truncating($\mathcal{P}$, R)	`(until-if` $\mathcal{P}$ $R_1 \ldots R_n$`)`
choosing($\mathcal{P}$, R)	`(choose-if` $\mathcal{P}$ R`)`
mingling(R, S, $\mathcal{P}$)	`(mingle` R S $\mathcal{P}$`)`
catenating(R, S)	`(catenate` R S`)`
spreading(R, S, z)	`(spread` R S z`)`
sectioning(R, m, n)	`(subseries` R m n`)`
chunking(w, d, R)	`(chunk` w d R`)`

Fig. 7. Lisp support for series functions.

values is treated as a separate state variable.

```
(scan-fn 'list #'(lambda () '(a b c d)) #'cddr #'null)
  ⇒ #Z((a b c d) (c d))

(scan-fn '(values integer integer)
         #'(lambda () (values 1 1))
         #'(lambda (i sum)
             (setq i (+ i 1)) (values i (+ sum i))))
  ⇒ #Z(1 2 3 4 ...) and #Z(1 3 6 10 ...)
```

Three other features of `scan-fn` are worthy of note. First, a new first argument is introduced, which specifies the type (or types) of the values returned by the stepping procedure. Given the lack of typing information in Lisp, this argument is necessary to ensure that the number of arguments returned by the stepping procedure can be determined at compile time. Second, the initial value is replaced by a procedure that returns the initial values. This is convenient in situations where multiple initial values are needed. Third, the predicate argument is made optional. Omitting it is the same as supplying a predicate that is not true of any value. The first and second extensions are applied to `collect-fn`, `collecting-fn`, and `map-fn` as well as `scan-fn`.

As a convenience to the user, a number of specific scanners are provided in addition to `scan-fn`. These include: `series` which creates a series indefinitely repeating a given value, `scan` which enumerates the elements in a list, vector, or string, `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 object to be scanned. If omitted, the type defaults to `list`.

```
(series "test") ⇒ #Z("test" "test" "test" ...)
(scan '(a b c)) ⇒ #Z(a b c)
(scan 'vector '#(a b c)) ⇒ #Z(a b c)
(scan 'string "Tuz") ⇒ #Z(#\T #\u #\z)
(scan-range :from 1 :upto 3) ⇒ #Z(1 2 3)
(scan-plist '(a 1 b 2)) ⇒ #Z(a b) and #Z(1 2)
```

Similarly, a number of specific collectors are provided including: `collect` which combines the elements of a series into a list, vector, or string, `collect-sum` which adds up the elements of a series, `collect-length` which returns the number of elements in a series, and `collect-first` which returns the last element of a series (or nil if the series is empty). The first argument of collect specifies the type of object to be produced. If omitted, the type defaults to list.

```
(collect #Z(a b c)) ⇒ (a b c)
(collect 'simple-vector #Z(a b c)) ⇒ #(a b c)
(collect 'string #Z(#\T #\u #\z)) ⇒ "Tuz"
(collect-sum #Z(1 3 2)) ⇒ 6
(collect-length #Z("fee" "fi" "fo" "fum")) ⇒ 4
(collect-first #Z("fee" "fi" "fo" "fum")) ⇒ "fee"
(collect-first #Z()) ⇒ nil
```

Finally, a number of additional transducers are provided including: previous (based on collecting-fn) which takes in a series and shifts it over one element by inserting the indicated value at the front and discarding the last element, choose (based on choose-if) which selects the elements of its second argument that correspond to non-null elements of its first argument, and positions (also based on choose-if) which returns the positions of the non-null elements in a series. If given only one argument, choose returns the non-null elements of this series.

```
(previous #Z("fee" "fi" "fo" "fum") "  ") ⇒ #Z("  " "fee" "fi" "fo")
(choose #Z(T nil T) #Z(1 2 3)) ⇒ #Z(1 3)
(choose #Z(nil 3 4 nil)) ⇒ #Z(3 4)
(positions (map-fn #'oddp #Z(1 2 3 5 6 8))) ⇒ #Z(0 2 3)
```

Convenient support for mapping. In cognizance of the ubiquitous nature of mapping, the Lisp series implementation provides three mechanisms that make it easy to express particular kinds of mapping. The # macro character syntax #M$\mathcal{F}$ converts a procedure $\mathcal{F}$ into a transducer that maps $\mathcal{F}$.

```
(#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 computation over one or more series without having to write a literal lambda expression. It has the same basic syntax as let. 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 nature 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.

Nested loops. The equivalent of a nested loop is expressed by simply using a series expression in a procedure that is mapped over a series. This is typically done

```
(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))))))
```

Fig. 8. Converting between lists and bit sets.

using mapping. In the example, a list of sums is computed based on a list of lists of numbers.

```
(let ((data '((1 2 3) (4 5 6) (7 8))))
  (collect
    (mapping ((number-list (scan data)))
      (collect-sum (scan number-list))))) ⇒ (6 15 15)
```

User-defined series procedures. As shown by the definitions of collect-sum and mapping below, the standard Lisp forms defun and defmacro can be used to define new series procedures. However, the series macro package must be informed when a series procedure is being defined with defun. This is done by using the declaration optimizable-series-function. No special declaration is required when using defmacro.

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

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

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 procedures 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.

Sets over a small universe can be represented very efficiently as binary integers where each 1 bit in the integer represents an element in the set. Here, sets represented 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.

The procedures in Figure 8 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

```
(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))
```

Fig. 9. Operations on series of bit sets.

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 rightmost bit.)

Given a bit set and its associated universe, the procedure `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 that correspond to 1s in the integer representing the bit set. (When no `:upto` argument is supplied, `scan-range` counts up forever.)

The procedure `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 procedure `bit-position` is called to determine which bit position should be set to 1. The procedure `ash` is used to create an integer with the correct bit set to 1. The procedure `collect-fn` is used to combine the integers corresponding to the individual elements together into a bit set corresponding to the list.

The procedure `bit-position` takes an item and a universe and returns the bit position corresponding to the item. The procedure operates in one of two ways depending on whether or not the item is in the universe. The first line of the procedure 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`.

If the item is not in the universe, the second line of the procedure 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 9 shows the definition of two collectors that operate on series of bit sets. The first procedure computes the union of a series of bit sets, while the second computes the 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 10 shows the data structure that represents a block of code. Each block B has several pieces of information associated with it. Two of these pieces of information are the blocks that can branch to B and the blocks B 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 procedure `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

```
(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 in the block.
  live         ;Variables that must be available 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 universe)
  (iterate ((block (scan program-graph)))
    (setf (temp block)
          (make-temp-bsets
            :used (list->bset (used block) universe)
            :written (list->bset (written block) universe)
            :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 universe)
  (iterate ((block (scan program-graph)))
    (setf (live block)
          (bset->list (bset-live (temp block)) universe))
    (setf (temp block) nil)))
```

Fig. 10. Live variable analysis.

it can potentially be used by a following block.) Finally, there is a temporary data field, which is used by procedures (such as `determine-live`) that perform computations involved with the blocks.

The remainder of Figure 10 shows the procedure `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 procedure uses bit sets. The procedure 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 10 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 procedure `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 procedure `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`.

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 the predecessors of the current block will be examined to see whether the new estimate for the current block requires their live estimates to 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. Figure 10 is a particularly good example of the way series expressions are intended to be used in three ways. First, all the series expressions are optimizable. Second, series expressions are used in a number of places to express computations that would otherwise be expressed less clearly as loops or less efficiently using operations on lists or vectors. Third, the main relaxation algorithm in `perform-relaxation` is expressed as a loop. This is done, because the data flow in this algorithm prevents it from being decomposed into two or more fragments. This highlights the fact that optimizable 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.

5. A PASCAL IMPLEMENTATION

Series can be added to Pascal in much the same way as they are added to Lisp. A prototype system has been constructed that demonstrates this [29, 46]. However, the prototype is written in Lisp rather than Pascal and only supports optimizable series expressions. A fatal error is issued whenever optimization is blocked. Although less complete than the approach of the Lisp implementation, this still allows loops to be replaced by optimizable series expressions.

Series. The Pascal series preprocessor supports the declaration of series in analogy with array declarations as shown below.

```
type Integers = series of Integer;
var InputData: series of Real;
```

In line with the general philosophy of Pascal, it is required that all the elements of a series have the same type. However, the length of a series is not part of its

Series Function	Pascal Implementation
collection(z, $\mathcal{F}$, R)	`CollectFn(`z, $\mathcal{F}$, R`)`
scanning(z, $\mathcal{F}$, $\mathcal{P}$)	`ScanFn(`z, $\mathcal{F}$, $\mathcal{P}$`)`
collecting(z, $\mathcal{F}$, R)	`CollectingFn(`z, $\mathcal{F}$, R`)`
mapping($\mathcal{F}$, R_1, ..., R_n)	`MapFn(`$\mathcal{F}$, R_1, .., R_n`)`
truncating($\mathcal{P}$, R)	`TruncateIf(`$\mathcal{P}$, R`)`
choosing($\mathcal{P}$, R)	`ChooseIf(`$\mathcal{P}$, R`)`
mingling(R, S, $\mathcal{P}$)	`Mingle(`R, S, $\mathcal{P}$`)`
catenating(R, S)	`Catenate(`R, S`)`
spreading(R, S, z)	`Spread(`R, S, z`)`
sectioning(R, n, m)	`Subseries(`R, n, m`)`
chunking(w, d, R)	`Chunk(`w, d, R, S_1, .., S_m`)`

Fig 11. Pascal support for series functions

type. This is important to facilitate the definition of series procedures operating on series of arbitrary length.

Series procedures. The series functions described in Section 3 are all supported by Pascal procedures as shown in Figure 11. In general, the Pascal procedures have the same names as the corresponding Lisp procedures with any hyphens removed (e.g., `scan-fn` becomes `ScanFn`). (The examples in [29, 46] show an obsolete set of names linked to an earlier Lisp implementation of series.) Since Pascal does not support the concept of a function procedure that returns multiple values, the outputs of chunking are turned into arguments.

```
Catenate(⟨1,2⟩, ChooseIf(Odd, ⟨8,-7,6,-1⟩)) ⇒ ⟨1,2,-7,1⟩
Subseries(Mingle(⟨1,5,9⟩, ⟨2,6,8⟩, <), 2, 4) ⇒ ⟨5,6⟩
Chunk(2, 1, ⟨1,5,3,7⟩, Xs, Ys) ⇒ Xs := ⟨1,5,3⟩ and Ys := ⟨5,3,7⟩
MapFn(Average, Xs, Ys) ⇒ ⟨3,4,5⟩
function Average (x,y: Integer): Integer;
begin
  Average := (x+y) div 2
end;
```

(Since the Pascal implementation does not provide a syntax for series literals, the syntax from Section 2 is used in the examples in this section. Note that while a notation for literal series is very convenient in small examples, it is of relatively little importance in other situations.)

The Pascal implementation does not extend the higher-order procedures over their specifications in Section 3 for two reasons. Given the strong typing in Pascal, the preprocessor can obtain type information without needing type arguments. Since, Pascal does not support the concept of multiple return values, some other method needs to be employed to avoid the need for tuples.

The procedures in Figure 11 do not follow the usual Pascal restrictions on the parameters of procedures. Some of the procedures allow the number of arguments they receive to vary and they all allow considerable flexibility in the types of their arguments. This is important because the series procedures are inherently generic in character. For instance, `MapFn` is naturally applicable to any number and any type of series as long as the element types are compatible with the procedure being mapped.

Due to their generic nature, the procedures in Figure 11 could not be implemented as user-defined procedures in Pascal. However, as an extension to the language, they do not violate the spirit of Pascal. In particular, the predeclared Pascal procedures are generic in exactly the same way. Several (e.g., `Read` and `Write`) allow variable numbers of arguments and most of them are applicable to more than one type of object. Using a more flexible language such as Ada [51], it would be possible to implement (at least most of) the higher-order series functions as user-defined procedures.

All of the specific scanners, collectors, and transducers from the Lisp implementation that are applicable to Pascal are supported by the Pascal implementation as well. Given the strong typing in Pascal, `Scan` and `Collect` do not need type arguments. Since Pascal has sets, but not lists, these functions apply to sets and not lists. In keeping with the general style of Pascal, `Collect` takes the destination vector/string/set as its first argument rather than returning an aggregate value.

```
Series('test') ⇒ ⟨'test','test','test', ...⟩
Scan('Tuz') ⇒ ⟨'T','u','z'⟩
Scan([Mon,Wed,Fri]) ⇒ ⟨Mon,Wed,Fri⟩
ScanRange(1, 3) ⇒ ⟨1,2,3⟩

Collect(X, ⟨'T','u','z'⟩)          { Places 'Tuz' in X. }
CollectSum(⟨1,3,2⟩) ⇒ 6
CollectLength(⟨'fee','fi','fo','fum'⟩) ⇒ 4
CollectLast(⟨'fee','fi','fo','fum'⟩) ⇒ 'fum'
CollectLast(⟨⟩, 'none') ⇒ 'none'

Previous(⟨'fee','fi','fo','fum'⟩, '   ') ⇒ ⟨'   ','fee','fi','fo'⟩
Choose(⟨true,false,true⟩, ⟨1,2,3⟩) ⇒ ⟨1,3⟩
Positions(MapFn(Odd, ⟨1,2,3,5,6,8⟩)) ⇒ ⟨0,2,3⟩
```

Implicit mapping. To avoid making syntactic extensions, the Pascal implementation does not support constructs analogous to the Lisp forms `mapping` and `iterate`. However, it supports a related concept that is in many ways even more useful. Whenever a non-series procedure is applied to a series, it is automatically mapped over the elements of the series. For example, in the expression below, `Sqr` is automatically mapped over the series of numbers created by scanning the set.

```
CollectSum(Sqr(Scan([2,4])))
  ≡ CollectSum(MapFn(Sqr, Scan([2,4]))) ⇒ 20
```

The key virtue of implicit mapping is that it reduces the number of helping procedures that have to be defined. For instance, in the example of a moving average on the previous page, you can write the following instead of defining a procedure `Average` and explicitly mapping it.

```
Chunk(2, 1, ⟨1,5,3,7⟩, Xs, Ys); (Xs+Ys) div 2 ⇒ ⟨3,4,5⟩
```

The concept of implicit mapping is completely separate from the other concepts associated with series expressions. As such, it could easily be dispensed with. However, as shown by experience with APL and the other languages that support it, implicit mapping is extremely useful. (Although the lack of reliable compile-time type information introduces a number of complications, implicit mapping is being added to the Lisp implementation.)

User-defined series procedures. As shown in the examples below, series procedures in Pascal are simply procedures that either have series inputs or return series values. As with series in general, all such definitions are handled directly by the preprocessor. There is no need for any special kind of declaration. Pascal does not support the concept of macros.

Example. The following example illustrates how series expressions can best be used in Pascal. As in the last section, all of the expressions are optimizable. The example revolves around a job queue data abstraction that might be used in an operating system. The basic type definitions are shown below. A `JobQ` is a pointer to a chain of jobs. A `Job` is a pointer to a descriptive record containing a number of fields including a numerical priority.

```
type JobQ = ^JobQentry;
type JobQentry = record; JobInfo: Job; Rest: JobQ end;
type Job = ^JobRecord;
type JobRecord = record Priority: Real; ... end;
```

There are a number of procedures defined that operate on job queues. These procedures include putting a new job onto a queue (shown below) and removing a job from a queue (discussed near the end of this section). To add a job onto a queue, one merely needs to allocate a new queue entry and attach it to the front of the queue.

```
procedure AddToJobQ (J: Job; var Q: JobQ);
  var E: JobQ;
begin
  new(E);
  E^.JobInfo := J;
  E^.Rest := Q;
  Q := E
end
```

In addition to ordinary procedures that operate on job queues, it is useful to define a number of series procedures that operate on job queues. In particular, as with any aggregate data structure, it is useful to have procedures `ScanJobQ` and `CollectJobQ` that convert job queues to series of jobs and vice versa. It also turns out to be useful to have a procedure `ScanJobQtails` that enumerates all the tails of a queue (i.e., ⟨`Q, Q^.Rest, Q^.Rest^.Rest, ...`⟩). As shown below, `ScanJobQtails` can be implemented using the higher-order series procedure `ScanFn` and two locally-defined procedures operating on job queues.

```
function ScanJobQtails (Q: JobQ): series of JobQ;
  function JobQrest (Q: JobQ): JobQ;
    begin JobQrest := Q^.Rest end;
  function JobQnull (Q: JobQ): Boolean;
    begin JobQnull := Q=nil end;
begin
  ScanJobQtails := ScanFn(Q, JobQrest, JobQnull)
end
```

Among other things, `ScanJobQtails` can be used to implement `ScanJobQ` as shown below. The expression `ScanJobQtails(Q)^.JobInfo` causes the operations of following a pointer and selecting the `JobInfo` field of a `JobQentry` to be implicitly mapped over the job queue pointers returned by `ScanJobQtails`.

```
function ScanJobQ (Q: JobQ): series of Job;
begin
  ScanJobQ := ScanJobQtails(Q)^.JobInfo
end
```

The procedure RemoveFromJobQ removes a job from the end of a queue. It can be implemented using ScanJobQtails as shown below. To start with, RemoveFromJobQ enumerates the tails of the queue and uses CollectLast and Previous to obtain pointers to the last and next to last entries in the queue. The job in the last queue entry is returned as the result of RemoveFromJobQ. (It is assumed that there is at least one job in the queue.) The rest pointer in the next to last entry is set to nil, in order to remove the last entry from the queue. (If there is no next to last entry, the queue variable itself is set to nil.) The storage associated with the last entry is then freed.

```
function RemoveFromJobQ (var Q: JobQ): Job;
  var Qs: series of JobQ;
      NextToLast, Last: JobQ;
begin
  Qs := ScanJobQtails(Q);
  Last := CollectLast(Qs, nil);
  NextToLast := CollectLast(Previous(Qs, nil), nil);
  RemoveFromJobQ := Last^.JobInfo;
  if NextToLast=nil
     then Q := nil
     else NextToLast^.Rest := nil;
  dispose(Last)
end
```

The first three statements in the body of RemoveFromJobQ form a series expression, while the remaining statements in the body are non-series expressions. From an efficiency standpoint, it should be noted that since there is only one instance of ScanJobQtails, the series expression is converted into a loop that only traverses the queue once.

As a final example of the use of optimizable series expressions, consider the procedure SuperJob below. This procedure inspects a job queue and returns the last (i.e., longest queued) job in the queue whose priority is more than two standard deviations larger than the average priority of the jobs in the queue. If there is no such job, nil is returned. The first four statements form a series expression that computes the basic information needed to calculate the mean and deviation of the priorities of the jobs. The last three statements form a second series expression that selects the jobs that have sufficiently large priorities and returns the last of these jobs, if any. The queue has to be scanned twice, because the selection cannot begin until after the mean and deviation have been computed.

```
function SuperJob (Q: JobQ): Job;
  var Jobs, SuperJobs: series of Job;
      N: Integer;
      Mean, SecondMoment, Deviation, Limit: Real;
begin
  Jobs := ScanJobQ(Q);
  N := CollectLength(Jobs);
  Mean := CollectSum(Jobs.Priority)/N;
  SecondMoment := CollectSum(Sqr(Jobs.Priority))/N;
  Deviation := Sqrt(SecondMoment-Sqr(Mean));
  Limit := Mean+2*Deviation;
  Jobs := ScanJobQ(Q);
  SuperJobs := Choose(Jobs.Priority>Limit, Jobs);
  SuperJob := CollectLast(SuperJobs, nil)
end
```

The programs above are a good example of the way series expressions are intended to be used. To start with, all of the programs are straightforward in nature. This reflects the fact that the primary goal of series expressions is to convert the vast majority of programs that are in fact straightforward programs into dirt simple programs. When a program is straightforward, it is usually easy to write it in a loop-free form without having to use anything other than very simple series expressions.

6. THE OPTIMIZATION ALGORITHMS

A preprocessor or compiler extension that transforms optimizable series expressions into loops can be implemented in three stages: *parsing* which locates optimizable expressions and converts them into equivalent data flow graphs, *pipelining* which collapses a graph into a single node representing an equivalent loop, and *unparsing* which converts this node into appropriate program code and inserts the code in place of the original expression.

Below, the Pascal implementation of series is used as a concrete illustration. The Common Lisp implementation works in the same way, except that the characteristics of Lisp simplify the parsing and unparsing stages.

Parsing. When the preprocessor is applied to a program, it begins by parsing the program. Series expressions are located by inspecting the types of the procedures called by the program. While this is being done, the static analyzability and straight-line computation restrictions are checked and any violations reported.

In a language like Pascal where complete compile-time type information is available, implicit mapping can be supported by noting places where non-series procedures are applied to series. Each such application is replaced by an appropriate use of `MapFn`.

The final action of the parsing stage is to create a data flow graph corresponding to each optimizable series expression located. Since each of these expressions is a straight-line computation, this is easy to do. Each procedure call becomes a node in the graph and the data flow between the nodes is derived from the way the procedure calls are nested and the way variables are used.

Pipelining. The operation of the pipelining stage is illustrated in Figure 12. The series expression in the procedure `SumSqrs` (which computes the sum of the squares of the odd elements of a vector) is transformed into the loop shown in the procedure `SumSqrsPipelined`. The readability of the loop code is reduced by the fact that it contains a number of internally generated variables. However, the code is quite efficient. The only significant problem is that the pipeliner sometimes uses more variables than strictly necessary (e.g., `Result5`). However, this need not lead to inefficiency during execution as long as a compiler capable of simple optimizations is available.

The pipelining process operates in several steps. In the first step, the divide and conquer strategy discussed in Section 2.5 is used to partition the data flow graph for a series expression into clusters where all the data flow connects on-line ports. While doing this, the pipeliner checks that the expression obeys the on-line cycle restriction.

Once partitioning is complete, the procedures in each cluster are combined into a single procedure. The resulting procedures are then combined based on the data flow between the subexpressions. To support the combination process, each series

```
      function SumSqrs (V: array [1..N] of Integer): Integer;
      begin
        SumSqrs := CollectSum(Sqr(ChooseIf(Odd, Scan(V))))
      end
```

$$\Downarrow$$

```
      function SumSqrsPipelined (V: array [1..N] of Integer): Integer;
        label 0,1;
        var Element12, Index15, Result5, Sum2: Integer;
      begin
[1]     Index15 := 0;
[4]     Sum2 := 0;
[1]  1: Index15 := 1+Index15;
[1]     if Index15>N then goto 0;
[1]     Element12 := V[Index15];
[2]     if not Odd(Element12) then goto 1;
[3]     Result5 := Sqr(Element12);
[4]     Sum2 := Sum2+Result5;
        goto 1;
     0: SumSqrsPipelined := Sum2
      end


[1] -- Scan of a vector ---------------------------------

      inputs- Vector: array [K..L] of ElementType;
     outputs- Element: Series of ElementType;
        vars- Index: Integer;
      prolog- Index: 1-K;
        body- Index := 1+Index;
              if Index>L then goto 0;
              Element := Vector[Index];


[2] -- ChooseIf -----------------------------------------

      inputs- function P(X: ElementType): Boolean;
              Item: Series of ElementType;
     outputs- Item: Series of ElementType;
      labels- 2;
        body- 2: NextIn(Item);
                 if not P(Item) then goto 2;


[3] -- Implicit mapping of Sqr -------------------------

      inputs- Item: Series of ElementType;
     outputs- Result: Series of ElementType;
        body- Result := Sqr(Item);


[4] -- CollectSum ---------------------------------------

      inputs- Number: Series of ElementType;
     outputs- Sum: ElementType;
      prolog- Sum := 0;
        body- Sum := Sum+Number;
```

Fig. 12. Transforming optimizable series expressions into loops.

procedure is represented as a loop fragment with one or more of the following parts:

`inputs`-	Input variables.
`outputs`-	Output variables.
`vars`-	Auxiliary variables used by the computation.
`labels`-	Labels used by the computation.
`prolog`-	Statements that are executed before the computation starts.
`body`-	Statements that are repetitively executed.
`epilog`-	Statements that are executed after the loop terminates.

The bottom part of Figure 12 shows the fragments that represent the procedures called by the series expression in `SumSqrs`. These fragments are combined to create the loop in `SumSqrsPipelined`. The numbers in the left hand margin indicate which fragment each line of the loop comes from. Two different combination algorithms are used: one corresponding to data flow between on-line ports and one corresponding to data flow touching off-line ports.

When two procedures are connected by data flow between on-line ports (e.g, the data flow from the output of the implicit mapping of `Sqr` to the input of `CollectSum`), the procedures are combined by simply concatenating the various parts of the corresponding fragments together. In addition, the variables and labels in the fragments are renamed so that there will be no possibility of conflicts. The data flow between the procedures is implemented by renaming the input variable of the destination so that it is the same as the output variable of the source. (The process above is much the same as an application of the standard compiler optimization technique of loop fusion [3].)

When two procedures are connected by series data flow terminating on an off-line input (e.g., the data flow from the output of `Scan` to the series input of `ChooseIf` in the figure), the fragment representing the destination procedure contains an instance of the form `NextIn`, which specifies when elements of the input should be computed. The two fragments are combined exactly as in the on-line combination algorithm except that the body of the source fragment is substituted in place of the call on `NextIn`, rather than being concatenated with the body of the destination fragment. (This process essentially compiles in support for a simple case of lazy evaluation [17].)

Unparsing. The result of pipelining is a single loop fragment that corresponds to the series expression as a whole. In the unparsing stage, this fragment is converted into a loop as indicated below. The combination process eliminates the inputs. The outputs are connected up to the surrounding code when the loop is substituted into the program in place of the original series expression. The other parts of the fragment appear directly in the loop.

```
 label 0,1,labels;
   var vars;
 begin
   prolog;
1: body;
   goto 1;
0: epilog;
```

Once each series expression has been replaced by a loop, the resulting code can be passed to a standard Pascal compiler.

Side-effects. The correctness preserving nature of the transformations above has only been verified under the assumption that there are no side-effects involved. However, it is believed that if unoptimized series are implemented as illustrated in the beginning of Section 4, then the transformations are also correctness preserving even in the presence of side-effects as long as they are applied to a series expression that only computes a single overall output.

The reason for this is that the transformed code exactly mimics the lazy evaluation of the untransformed expression. For instance, the off-line port combination algorithm involves code motion; however, this motion simply moves the generation of the series elements to the place where lazy evaluation requires the elements of the series to be first computed.

The above can be made more formal from the point of view of path analysis [11]. Path analysis seeks to determine at compile time where in a program each lazy value will be first used and where it will be reused. This information can be used to optimize lazy evaluation in two ways. If there is an identifiable place of first use of a given value, then ordinary evaluation can be used instead of lazy evaluation for that value. If there is an identifiable last use for a value, the value does not have to be stored beyond that time.

The restrictions in Section 2 guarantee that for each series, there is an identifiable place where each element of the series is first used and that, for each element, the last use precedes the computation of the next element. The transformations above merely position the computation of the elements at their place of first use and omit their long term storage.

The above notwithstanding, one should realize that side-effects are still problematical, because lazy evaluation makes it difficult for programmers to figure out what the net results of side-effects will be. Some situations can be readily understood. For example, one can depend on the fact that mapping will apply the mapped function $\mathcal{F}$ first to the first element of the input, then to the second, and so on, in strict temporal order. Thus, if $\mathcal{F}$ interacts with itself or the environment outside of the containing series expression X via side-effects (e.g., by doing input or output), but does not interact with anything else in X, the result is easy enough to understand. More complex uses of side-effects should be avoided.

6.1 Systems Based on Similar Algorithms

The algorithms described above have evolved into their current form over thirteen years. The first generally available implementation was a Lisp macro package called LetS [43, 44]. The current Lisp implementation [47] is available in portable Common Lisp.

The same basic approach to representing and combining sequence procedures was independently developed by Wile [49]. However, he does not explicitly address the question of restrictions and his approach does not guarantee that every intermediate sequence can be eliminated. Much the same can be said about optimizing APL compilers [12].

A quite similar approach is also used internally by the Loop macro [13]. However, as discussed in the next section, the Loop macro is externally very different from series expressions. In particular, it uses an idiosyncratic English-like syntax rather than representing computations as compositions of procedures operating on series.

7. COMPARISONS

There are two primary vantage points from which to compare series expressions with related concepts. The most obvious comparison is with other support for sequence expressions. From this perspective, the key feature of series expressions is that they support most of the operations supported by the vector operations of APL [32], the sequence operations of Common Lisp [37], and the stream operations of Seque [20] (along with a few additional ones) while being more efficient.

Another way to view series expressions is that they are a logical continuation of the trend in programming-language design toward supporting the reuse of loop fragments. From this point of view, series expressions extend the approach taken by iterators in CLU [28] and the Lisp Loop macro [13]. The key feature of series expressions in this context is that they support the reuse of a wider variety of fragments and are easier to understand and modify, without being any less efficient.

To lend depth to the comments above, the remainder of this section presents detailed comparisons between series expressions and five other approaches. Each of these comparisons features the example below. This example shows the definition of a procedure that computes the sum of the positive elements of a vector. It also illustrates how a new series procedure can be defined.

```
function SumPositive (V: array [1..N] of Integer): Integer;
  function Positive (X: Integer): Boolean;
    begin Positive := X>0 end;
begin
  SumPositive := CollectSum(ChooseIf(Positive, Scan(V)))
end

function CollectSum (S: series of Integer): Integer;
begin
  CollectSum := CollectFn(0, +, S)
end
```

7.1 Other Support for Sequence Expressions

There are many programming languages that provide support for sequence expressions, e.g., [5, 6, 20, 26, 33, 35, 36, 37]. So many, that it would not be practical to make detailed comparisons between each one of these languages and series expressions. Three representative languages are discussed below.

APL. One of the oldest and must used languages that takes a functional approach is APL [24, 32]. A style of writing APL has evolved where vector expressions are used instead of loops. The correspondence between the series functions discussed in Section 3 and the APL vector operators is summarized in Figure 13. The APL concept of the extension of scalar operations to vectors corresponds to implicit mapping.

As illustrated below, both the vector summation algorithm and user-defined sequence procedures can be compactly represented in APL. Since sequences are directly represented as vectors, there is no need for an explicit Scan operation.

```
     ∇ SUM←SUMPOSITIVEAPL V
[1]    SUM←COLLECTSUM((V>0)/V)
     ∇

     ∇ SUM←COLLECTSUM NUMBERS
[1]    SUM←+/NUMBERS
     ∇
```

Series Function	APL Operation	name
collection(z, $\mathcal{F}$, R)	$\mathcal{F} / R$	reduction
scanning(z, $\mathcal{F}$, $\mathcal{P}$)	*missing*	
collecting(z, $\mathcal{F}$, R)	$\mathcal{F} \backslash R$	scanning
mapping($\mathcal{F}$, R_1, ..., R_n)	$R_1\ \mathcal{F}\ R_2$	extension of scalar operations
truncating($\mathcal{P}$, R)	$(((\mathcal{P}\ R)\ \iota\ 1)-1) \uparrow R$	take
choosing($\mathcal{P}$, R)	$(\mathcal{P}\ R) / R$	compression
mingling(R, S, $\mathcal{P}$)	*missing*	
catenating(R, S)	$R\ ,\ S$	catenation
spreading(R, S, z)	*idiom based on*	expansion
sectioning(R, m, n)	$(n-m) \uparrow m \downarrow R$	take and drop
chunking(w, d, R)	*missing*	
simple idioms	index generation, membership, inner product, etc.	
missing	reversal, rotation, grade up/down, modifying elements	

Fig. 13. The correspondence between series functions and APL operations.

The key differences between APL and series expressions are that APL vectors cannot represent unbounded sequences, the set of APL operations is somewhat different, and users are not given any feedback about what is efficient and what is not.

Although all the series operations could have been supported in APL, there is no direct built-in support for scanning, mingling, or chunking. In addition, APL does not support higher-order operations as well as it might initially appear. For instance, the reduction operator appears to be a higher-order operation. However, at least in standard APL, the operation to be reduced must be one of the predefined scalar operations—user-defined operations cannot be used. As a result, the reduction operator is actually just part of a naming scheme for a small set of specific collectors. (This has the collateral benefit of allowing the initial identity value to be implicit.)

APL supports four operations (reversal, rotation, grade up, and grade down) that are not supported by series expressions, because they cannot be implemented in a preorder fashion without introducing significant inefficiency. APL also allows the modification of the elements of a sequence. When using series expressions, one has to rely on other constructs in the host language when performing any of these operations.

For instance, to sort a series in the Lisp implementation of series, one must first collect the series into a list or vector and then sort the resulting structure. Explicitly creating an intermediate structure makes the expensive nature of sorting more obvious, however, it does not make sorting more expensive, because sorting always requires that some physical representation of the sequence be created.

A few APL compilers [12, 22] are capable of producing efficient code in most of the situations where series expressions can be optimized; however, most are not. As a result, optimizable series expression are typically much more efficient. Further, even when the compiler supports optimization, programmers are not given any feedback about whether optimization has occurred. Rather, programmers are (at least implicitly) encouraged not to think about such issues.

An area where APL is fundamentally more powerful than series expressions is that the standard intermediate structure in APL is the array. APL has a number of powerful array operators (not shown in Figure 13) and a few APL compilers can optimize some array expressions. In contrast, while it is possible to have series of

Series Function	Sequence Operation
collection(z, $\mathcal{F}$, R)	`(reduce` $\mathcal{F}$ R`)`
scanning(z, $\mathcal{F}$, $\mathcal{P}$)	*missing*
collecting(z, $\mathcal{F}$, R)	*missing*
mapping($\mathcal{F}$, R_1, ... R_n)	`(map` *type* $\mathcal{F}$ R_1 ... R_n`)`
truncating($\mathcal{P}$, R)	`(subseq` R `0 (position-if` $\mathcal{P}$ R`))`
choosing($\mathcal{P}$, R)	`(remove-if-not` $\mathcal{P}$ R`)`
mingling(R, S, $\mathcal{P}$)	`(merge` *type* R S $\mathcal{P}$`)`
catenating(R, S)	`(concatenate` *type* R S`)`
spreading(R, S, z)	*missing*
sectioning(R, m, n)	`(subseq` R m n`)`
chunking(w, d, R)	*missing*
simple idioms	`elt`, `length`, `count`, `find`, `some`, etc
missing	`reverse`, `sort`, modifying elements

Fig. 14 The correspondence between series functions and sequence operations.

series, there are no special series operations for operating on them, and they are never optimized.

Finally, a superficial but striking difference between series expressions and APL is that series expressions use standard subroutine calling notation while APL uses a special set of concise, but cryptic, operators.

Common Lisp Sequence Operations. While many (if not most) Lisp programmers use loops extensively, a style of writing Lisp has evolved where expressions computing intermediate lists and vectors are used instead of loops. Unfortunately, until recently, Lisp supported an impoverished set of predefined sequence operations—it supported `mapcar`, but not much else. When Common Lisp was designed [37], this defect was rectified by introducing a relatively comprehensive suite of sequence operations.

In Common Lisp, the term 'sequence' is used to refer to either a list or a vector. However, since both of these structures are limited to representing bounded sequences, Lisp sequences are not a complete implementation of mathematical sequences. The correspondence between the basic series functions and the Lisp sequence operations is summarized in Figure 14. Lisp does not support implicit mapping.

The example below shows how the Common Lisp sequence operations can be used to express the vector summation algorithm and a user-defined sequence procedure. Since a vector is a Lisp sequence, there is no need for an explicit `scan` procedure.

```
(defun sum-positive-sequence (v)
  (collect-sum (remove-if-not #'plusp v)))

(defun collect-sum (numbers)
  (reduce #'+ numbers))
```

Except for the fact that Lisp sequences cannot represent unbounded sequences, there is no reason why all the series functions could not be supported by sequence operations. However, there is no direct built-in support for scanning, collecting, spreading, or chunking. In addition, the identity value to use for `reduce` (collection) is specified in an odd way. The procedure argument must be implemented in such a way that when called with zero arguments it returns the identity value.

Current Lisp compilers do not optimize sequence expressions. As a result, optimizable series expressions are much more efficient. In light of the lack of optimization, it is not surprising that Lisp provides no feedback about optimizability. As in APL, there is no bias toward preorder functions and modification of sequence elements is allowed. It is also common to have sequences of sequences, however, Lisp does not provide any special operations for manipulating them.

An interesting aspect of the Lisp sequence operations is that they typically support a number of keyword arguments that modify their behaviors. For example, consider the sequence operation `count-if`, which takes a predicate and a sequence, and returns a count of the number of elements in the sequence that satisfy the predicate.

```
(count-if #'plusp '(1 -2 3 4 -5)) ⇒ 3
```

The Lisp operation `count-if` takes two keyword arguments `:start` and `:end`, which can be used to specify a subsection of the input in which counting is to occur. In addition, a keyword argument `:key` can be used to specify an access procedure that will be used to fetch the part of each sequence element that should be tested by the predicate. Finally, an operation `count-if-not` exists, which is the same as `count-if` except that it automatically negates the values returned by the predicate.

As illustrated by the example below, none of these options is strictly necessary. The `:start` and `:end` keywords can be dispensed with by using `subseq`. The `:key` keyword and `count-if-not` can be dispensed with by specifying complex predicates.

```
(count-if-not #'plusp '((1) (-2) (3) (4) (-5))
              :start 0 :end 3 :key #'car)
   ≡ (count-if #'(lambda (element) (not (plusp (car element))))
               (subseq '((1) (-2) (3) (4) (-5)) 0 3)) ⇒ 1
```

Nevertheless, the various options described above are important for two reasons. First, they promote efficiency. (Using `subseq` instead of the `:start` and `:end` keywords is inefficient, because it creates an intermediate sequence.) Second, they increase the probability that predefined operations can be used as procedure arguments instead of `lambda` expressions. This makes uses of `count-if` more concise and easier to read.

Using series expressions, neither of these issues comes up. In the Lisp series expression below, the use of `subseries` does not lead to inefficiency, since pipelining eliminates the physical creation of its output series. Convenient support for mapping makes it possible to avoid the need for an explicit `lambda` expression. The desired test and key is simply mapped over the series in question (again without inefficiency). Finally, `count-if` itself can be dispensed with by using a combination of `choose` and `collect-length`. The approach taken by series expressions allows the individual procedures to be simpler and makes programs more functional in appearance.

```
(let ((elements (subseries (scan '((1) (-2) (3) (4) (-5))) 0 3)))
  (collect-length (choose (#Mnot (#Mplusp (#Mcar elements)))))) ⇒ 1
```

Seque. Under the name of streams, sequences are the central data type of the language Seque [20]. Using the same basic lazy evaluation technique discussed in the beginning of Section 4, Seque supports both bounded and unbounded sequences. The correspondence between the series functions discussed in Section 3 and the

Series Function	Seque Operation	name
collection(z, $\mathcal{F}$, R)	`Red`(R, $\mathcal{F}$) `!` `Length`(R)	reduction
scanning(z, $\mathcal{F}$, $\mathcal{P}$)	*idioms based on*	generators
collecting(z, $\mathcal{F}$, R)	`Red`(R, $\mathcal{F}$)	reduction
mapping($\mathcal{F}$, R_1, ..., R_n)	[$\mathcal{F}$(R_1 `!` i, .. , R_n `!` i)]	derived stream
truncating($\mathcal{P}$, R)	R \\\\ [`if` $\mathcal{P}$(R `!` i) `then` $i-1$] `!` 1	post-truncation
choosing($\mathcal{P}$, R)	[`if` $\mathcal{P}$(R `!` i) `then` R `!` i]	filtering
mingling(R, S, $\mathcal{P}$)	*missing*	
catenating(R, S)	R `->` S	concatenation
spreading(R, S, z)	*missing*	
sectioning(R, m, n)	$R\{m-1, n-2\}$	sectioning
chunking(w, d, R)	*missing*	
simple idioms	`Length`, referencing, operations over streams, etc.	
missing	modifying elements	

Fig. 15. The correspondence between series functions and Seque operations

stream operations provided by Seque is summarized in Figure 15. As in APL, many of these operations are provided by means of special syntax. In addition, implicit mapping is supported for many non-stream operations when they are applied to streams.

As illustrated below, both the vector summation algorithm and user-defined sequence procedures can be easily represented in Seque. Since streams are a distinct data type from vectors, an operation `!` equivalent to `Scan` is required.

```
procedure SumPositive (V)
  return CollectSum([:!V: lambda(e) if e>0 then e])
end

procedure CollectSum (S)
  L := Length(S)
  return if L=0 then 0 else Red(S, "+")!L
end
```

Since unbounded sequences are supported, it would be easy to completely support all the series functions in Seque. However, there is no direct support for mingling, spreading, or chunking. In addition, collection is only indirectly supported by collecting, this leads to awkwardness when collection is applied to an empty sequence, because there is no specification of the correct default value to return. There is also no direct support for the higher-order function scanning. However, there is an impressive array of facilities for defining scanning functions, both in Seque and in the language Icon [19], which Seque is based on. It is interesting to note that like the series operations, all of Seque's stream operations are preorder.

Seque does not attempt to optimize the evaluation of stream expressions by eliminating the computation of unnecessary intermediate streams. As a result, series expressions are never less efficient and often much more efficient. Seque programmer's are encouraged to think in terms of streams of streams and to make use of assigning to the elements of streams without any regard for the consequences on efficiency.

Summary. In comparison with the languages above, series expressions have three principal advantages. They support a wider range of operations than any one of the languages. Except in comparison with the best of APL compilers, they are much more efficient. They give clear feedback about what is efficient and what is not. As part of this, they make the use of procedures that cannot be efficiently implemented as preorder procedures more awkward, by forcing the programmer to use the facilities of the host language.

7.2 Looping Notations

The fundamental virtues of series expressions in comparison with looping constructs are illustrated by the discussion in the beginning of Section 1. However, this discussion is colored by the fact that it illustrates the use of only the most basic kind of looping construct. More complex looping constructs support several of the features of series expressions. In particular, they allow the equivalent of scanners and collectors (but not transducers) to be expressed as localized forms rather than as a statements dispersed in a loop.

Most programming languages contain a `for` construct, which makes it easy to express loops that are based on enumerating a range of integers. Some languages go beyond this by providing special looping forms corresponding to a few additional scanners. For example, Common Lisp provides a form `dolist` that makes it easy to implement a loop based on scanning the elements of a list. A couple of languages go further still by supporting a relatively wide range of standard looping fragments. Some of the oldest and most comprehensive support for this is in Lisp.

The Lisp Loop macro. The Lisp Loop macro [13] (which is based on the iterative statements in the InterLisp Clisp facility [38]) introduces two concepts into Lisp. First, it supports a looping construct analogous to `for` that uses an Algol-like keyword syntax. Second, it goes way beyond most `for` constructs by supporting a wide range of looping fragments analogous to scanners and collectors. Because of its non-Lisp syntax, the Loop macro has always been controversial. However, because of the utility of its predefined looping fragments, it has gained wide use.

The example below shows a program that uses the Loop macro to implement the vector summation algorithm. (The Loop macro does not support the definition of new collector-like fragments.) The code produced by the Loop macro is more or less identical to the code produced when optimizable series expressions are pipelined. As a result, the two approaches are equally efficient.

```
(defun sum-positive-loop-macro (v)
  (loop for item being each vector-element of v
        when (plusp item)
          sum item))
```

Loop supports a keyword `when` that is similar to the transducer choosing (see the example). A call on the Loop macro can also contain an arbitrary body that is mapped over the values computed by the scanner-like fragments (this is not shown in the example). However, the Loop macro does not support any other transducer-like looping fragments.

Although Loop does not support the definition of new collector-like fragments, it does support the definition scanner-like fragments. However, as illustrated by the following example (which shows the definition of the keyword `vector-element` used in `sum-positive-loop-macro`), these facilities are quite cumbersome. The user has to define a procedure that deals with parsing parts of the Loop syntax and that returns a list of six parts analogous to the parts of the loop fragments discussed in Section 6. (Recently, the Common Lisp standardization committee decided to adopt most of the Loop macro as part of Common Lisp. However, on the grounds that it was too complex, they decided not to include the scanner-defining capabilities.)

```
(define-loop-path vector-element scan-vector (of))

(defun scan-vector (path-name variable data-type prep-phrases
                    inclusive? allowed-prepositions data)
    (declare (ignore path-name data-type inclusive?
                     allowed-prepositions data))
  (let ((vector (gensym))
        (i (gensym))
        (end (gensym)))
    `(((,vector) (,i 0) (,end) (,variable))
      ((setq ,vector ,(cadar prep-phrases))
       (setq ,end (- (length ,vector) 1)))
      (> ,i ,end)
      (,variable (aref ,vector ,i))
      nil
      (,i (+ ,i 1)))))
```

A subtle difficulty with the Loop macro is that there are no restrictions on the computation that can be in the body and there is no attempt to prevent the body from interfering with the computation specified by the looping fragments. As a result, programmers cannot depend on the fact that these fragments will necessarily do what they are intended to do.

The concept of Generators and Gatherers presented in [30], provides essentially the same capabilities as the Loop Macro, but with a more functional syntax and simpler defining forms.

Iterators in CLU. Among Algol-like languages, some of the most powerful support for the use of looping fragments is provided by CLU [28]. In CLU, scanner-like fragments called *iterators* can be used in `for` loops to generate a series of elements that are processed by the body of the `for`. CLU provides a number of predefined scanners, including one corresponding to scanning a vector, and users can define new ones. (Alphard [50] supports a construct called a generator that is essentially identical to a CLU iterator. More recently, [10] shows how generators can be supported in Ada using generic packages.)

As an illustration, the example below shows how the vector summation algorithm can be expressed in CLU. It also shows the definition of a user-defined scanner. This is done by writing a coroutine that yields the scanned elements one at a time.

```
SUM_POSITIVE_CLU = proc(V: ARRAY[INT]) returns(INT)
    SUM: INT := 0
    for ITEM: INT in SCAN_VECTOR(V) do
        if ITEM>0 then SUM := SUM+ITEM end
    end
    return(SUM)
end SUM_POSITIVE_CLU

SCAN_VECTOR = iter(V: ARRAY[INT]) yields(INT)
  I: INT := ARRAY[INT]$LOW(V)
  END: INT := ARRAY[INT]$HIGH(V)
  while I<=END do
    yield(V[I])
    I := I+1
  end
end SCAN_VECTOR
```

Taken together, CLU iterators and the `for` statement are essentially the same as the Loop macro except for three things. Nothing besides mapping and scanning is supported. (In the example above, the operations of choosing and summing are both represented in non-local ways in the body of the loop.) Each `for` can only contain one iterator instead of many. CLU's method for defining iterators as coroutines is significantly easier to use than the Loop macro's scanner-defining form.

A method for supporting multiple iterators in a CLU `for` statement is described in [15]. Going beyond this, [14] describes how one could support collectors (again restricted to only one in each loop). While both of these papers merely present proposals rather than describing actual implementations, there is no doubt that everything supported by the Lisp Loop macro could be straightforwardly supported in an Algol-like language.

Summary. The key difference between the looping constructs above and series expressions is that while the looping constructs support looping fragments corresponding to (potentially unbounded) scanners and collectors and are highly efficient, they do not support the concept of a sequence data structure nor the idea of treating loop fragments as procedures. This preserves the iterative feel of the constructs, however, it is significantly limiting in several ways.

The lack of a sequence data type prevents the constructs from supporting anything other than a few simple transducers. (It is not clear how one could support transducers in general without having some kind of object that they can act upon.)

The fact that the loop fragments are not procedures means that the way the fragments can be used is intimately tied up with the syntax of the constructs. One has to learn a new language of combination rather than simply using standard functional composition. In addition, this new language of combination is much more restricted than functional composition. For instance, the only thing that can be done with a scanner-like fragment is to use it in a call on `loop` or `for` and map some computation over the values scanned.

An interesting thing to note about the looping constructs above is the way they avoid getting involved with a discussion of the restrictions in Section 2. By not allowing a sequence data structure to be stored in a variable, only supporting preorder fragments, largely ignoring transducers (particularly off-line ones), and limiting the way fragments can be combined, the constructs implicitly enforce these restrictions without having to talk about them. Unfortunately, the total restrictions they embody are much stronger than the ones in Section 2. This unnecessarily limits what can be expressed.

8. BENEFITS

There are three principal perspectives from which to view series expressions. To start with, series expressions can be looked at as embodying relatively complete support for sequence expressions in such a way that they can be included in any programming language without removing any preexisting features of the language, requiring the use of unusual syntax, or causing inefficiency. This support includes most of the operations provided by languages such as APL, Lisp, and Seque along with a few additional ones.

An alternate perspective focuses on the fact that programmers are given clear and immediate feedback about the efficiency of the series expressions they write. Series expressions that do not violate the restrictions in Section 2 are guaranteed to be as efficient as they look. By means of error messages, programmers are encouraged to think of efficient methods for computing the results they want. In particular, unlike APL, Lisp, or Seque, programmers are never tempted to think that all sequence expressions are equally efficient.

A final perspective is summarized by the statement that "optimizable series expressions are to loops as structured control constructs are to gotos." By using optimizable series expressions, it is possible to banish loops from most programs. Given that expressions are much easier to understand and modify than loops, this has the potential for being a step forward at least as important as banishing gotos.

In the context of this final perspective, it is worthy of note that it has been shown [42, 34] that it is possible to analyze loops and automatically convert them into series expressions. As a result, it should be possible to construct a tool that automatically converts the loops in pre-existing programs into series expressions. This would allow programmers to obtain the full benefits of using series expressions when maintaining old code.

While the idea has not yet been explored, optimizable series expressions might also be helpful in the context of parallelism. Even though it is oriented toward sequential machines, the pipelining applied to optimizable series expressions is very much the same as the 'software pipelining' of loops for execution on very large instruction word machines [27] and for execution by the processors of systolic arrays [2, 21]. If programs were written using series expressions, the process of analyzing the programs to determine a good schedule for the pipelined computation might be simplified. In addition, the restrictions in Section 2 appear relevant, because buffering of elements also causes inefficiency in a parallel context.

The application of optimizable series expressions to non-pipelined parallelism is less clear. The emphasis in such situations is on locating opportunities for evaluating subcomputations completely in parallel with no data flow between them. This is appropriate for mapping, but not for most of the other series operations. Nevertheless, using optimizable series expressions might make it easier to detect where such parallelism exists. For example, this might make it easier to vectorize [4] programs.

ACKNOWLEDGMENTS

The concept of optimizable series expressions 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. In addition, the anonymous reviewers made suggestions that significantly improved both the concept and the presentation.

This paper describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support was provided by the Defense Advanced Research Projects Agency (under Naval Research contracts N00014-75-C-0643, N00014-80-C-0505, N00014-85-K-0124, and N00014-88-K-0487), the National Science Foundation (under grants MCS-7912179, MCS-8117633, and IRI-8616644), and the IBM, NYNEX, Siemens, Sperry, and Microelectronics and Computer Technology corporations. The views and conclusions contained in this paper are those of the author and should not be interpreted as representing the policies, expressed or implied, of these organizations.

REFERENCES

1. AHO, A , HOPCRAFT, J , AND ULLMAN, J. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

2. AIKEN, A., AND NICOLAU, A. Optimal loop parallelization. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation* (Atlanta, GA, June 1988). ACM, New York, 1988, 308–317.
3. ALLEN, F., AND COCKE, J. A catalogue of optimizing transformations. In *Design and Optimization of Compilers.* R. Rustin, Ed., Prentice Hall, New York, 1971.
4. ALLEN, R., AND KENNEDY, K Automatic translation of Fortran programs to vector form. *ACM Trans. Program. Lang. and Syst. 9*, 4 (Oct. 1987), 491–542.
5. BACKUS, J. Can programming be liberated from the Von Neuman style? A functional style and its algebra of programs. *Commun. ACM 21*, 8 (Aug. 1978), 613–641.
6. BARSTOW, D. Automatic programming for streams. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence* (Aug. 1985). 232–237.
7. BELLEGARDE, F. Rewriting systems on FP expressions that reduce the number of sequences they yield. In *Proceedings ACM Symposium on Lisp and Functional Programming* (Aug. 1984). ACM, New York, 1984, 63–73.
8. BELLEGARDE, F. Convergent term rewriting systems can be used for program transformation. In *Proceedings Workshop on Programs as Data Objects.* Springer-Verlag, New York, 1985, 24–41.
9. BIRD, R. An introduction to the theory of lists. In *Logic of Programming and Calculi of Discrete Design.* M. Broy, Ed., NATO ASI series, Springer Verlag, New York, 1986, 5-42.
10. BISHOP, J The effect of data abstraction on loop programming techniques. *IEEE Trans. Softw. Eng. 16*, 4 (April 1990), 389–402.
11 BLOSS, A., HUDAK, P., AND YOUNG, J. Code optimizations for lazy evaluation. *Lisp and Symbolic Comput. 1*, 2 (Sept. 1988), 147–164.
12. BUDD, T. *An APL Compiler.* Springer-Verlag, New York, 1988.
13. BURKE, G., AND MOON, D Loop iteration macro. Massachusetts Institute of Technology Rep. LCS/TM-169, July 1980.
14. CAMERON, R. Efficient high-level iteration with accumulators. *ACM Trans. Program. Lang. and Syst. 11*, 2 (Feb. 1989), 194–211.
15. ECKART, J. Iteration and abstract data types. *ACM SIGPLAN Not. 22*, 4 (April 1987), 103–110.
16. EMERY, J. Small-scale software components. *ACM SIGSOFT Softw. Eng. Not. 4*, 4 (Oct. 1979), 18–21.
17. FRIEDMAN, D., AND WISE, D. CONS should not evaluate its arguments. University of Indiana Computer Science Department Rep. 44, Nov. 1975.
18. GOLDBERG, A., AND PAIGE, R. Stream processing. Rutgers University Laboratory for Computer Systems Research Rep. LCSR-TR-46, Aug. 1983.
19. GRISWOLD, R., AND GRISWOLD, M. *The Icon Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 1983.
20. GRISWOLD, R., AND O'BAGY, J. Seque: A programming language for manipulating sequences. *Comput. Lang. 13*, 1 (Jan. 1988), 13–22.
21. GROSS, T., AND SUSSMAN, A. Mapping a single-assignment language onto the Warp systolic array. In *Functional Programming Languages and Computer Architecture.* G. Kahn, Ed., Springer-Verlag, New York, 1987, 347–363.
22. GUIBAS, L., AND WYATT, D. Compilation and delayed evaluation in APL. In *Proceedings 1978 ACM Conference on the Principles of Programming Languages* (Sept. 1978). ACM, New York, 1978.
23. HARTMANIS, J., LEWIS, P., AND STEARNS, R. Classification of computations by time and memory requirements. In *Proceedings IFIP Congress 65.* Spartan Books, Washington DC, 1965, 31–35.
24. IVERSON, K. Operators. *ACM Trans. Program. Lang. and Syst. 1*, 2 (Oct. 1979), 161–176
25. JENSEN, K., AND WIRTH, N. *Pascal User Manual and Report.* Springer-Verlag, New York, 1985.
26. KAHN, G., AND MACQUEEN, D. Coroutines and networks of parallel processes. In *Proceedings 1977 IFIP Congress.* North-Holland, Amsterdam, 1977.
27 LAM, M. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings SIGPLAN '88 Conference on Programming Language Design and Implementation* (Atlanta, GA, June 1988). ACM, New York, 1988, 318–328.

28. LISKOV, B., *et. al* *CLU Reference Manual.* Springer-Verlag. New York, 1981.

29 ORWANT, J. Support of obviously synchronizable series expressions in Pascal. Massachusetts Institute of Technology Rep. AI/WP-312, Sept. 1988.

30. PERDUE, C , AND WATERS, R. Generators and gatherers. In *Common Lisp: the language*, 2nd Ed. G Steele Jr., Ed., Digital Press, Burlington, MA, 1990, 956–959

31. PINGALI, K., AND ARVIND Efficient demand-driven evaluation, part 1. *ACM Trans Program. Lang and Syst* 7, 2 (April 1985), 311–333.

32. POLIVKA, R., AND PAKIN, S. *APL. The Language and Its Usage.* Prentice-Hall, Englewood Cliffs, NJ, 1975.

33. PRYWES, N., PNUELI, A., AND SHASTRY, S Use of a non-procedural specification language and associated program generator in software development. *ACM Trans Program Lang and Syst. 1*, 2 (Oct. 1979), 196–217.

34. RICH, C., AND WATERS, R. *The Programmer's Apprentice.* Addison-Wesley, Reading MA, 1990.

35. RUTH, G , ALTER, S , AND MARTIN, W. A very high level language for business data processing. Massachusetts Institute of Technology Rep LCS/TR-254, 1981.

36. SCHWARTZ, J *et. al.* *Programming With Sets. An Introduction To SETL* Springer-Verlag, New York, 1986.

37. STEELE, G JR *Common Lisp: The Language.* Digital Press, Maynard, MA, 1984.

38 TEITELMAN, W *Interlisp Reference Manual.* Xerox PARC, 1978.

39. WADLER, P. Applicative languages, program transformation, and list operators. In *Proceedings ACM Conference on Functional Programming Languages and Computer Architecture* (Oct. 1981). ACM, New York, 1981, 25–32

40 WADLER, P Listlessness is better than laziness; Lazy evaluation and garbage collection at compile-time. In *Proceedings ACM Symp. on Lisp and Functional Programming* (Aug. 1984). ACM, New York, 1984, 45–52.

41 WADLER, P Listlessness is better than laziness II: Composing listless functions. In *Proceedings workshop on Programs as Data Objects.* Springer-Verlag, New York, 1985.

42. WATERS, R A method for analyzing loop programs. *IEEE Trans Softw. Eng. 5*, 3 (May 1979), 237–247.

43 WATERS, R. LetS. An expressional loop notation Massachusetts Institute of Technology Rep AIM-680a, Oct. 1982.

44 WATERS, R. Expressional loops. In *Proceedings 1984 ACM Conference on the Principles of Programming Languages* (Jan. 1984). ACM, New York, 1984, 1–10

45. WATERS, R Efficient interpretation of synchronizable series expressions. In *Proceedings ACM SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques. ACM SIGPLAN Not. 22*, 7 (July 1987), 74–85.

46. WATERS, R Using obviously synchronizable series expressions instead of loops. In *Proceedings 1988 International Conference on Computer Languages* (Miami, FL, Oct. 1988). IEEE Computer Society Press, New York, 1988, 338–346.

47. WATERS, R. Optimization of series expressions: part I: A user's manual for the series macro package. Massachusetts Institute of Technology Rep. AIM-1082, Dec 1989

48. WATERS R Series. In *Common Lisp: The Language*, 2nd Ed. G. Steele Jr., Ed., Digital Press, Burlington, MA, 1990, 923–955.

49. WILE, D. Generator expressions. USC Information Sciences Institute Rep. ISI/RR-83-116, 1983

50. WULF, W., LONDON, R , AND SHAW, M An introduction to the construction and verification of Alphard programs. *IEEE Trans. Softw. Eng. 2*, 4 (Dec. 1976), 253–265.

51 Military Standard Ada Programming Language. ANSI/MIL-STD-1815A-1983, U.S. Government Printing Office, Feb. 1983.

Received December 1987; revised November 1989 and May 1990; accepted July 1990.