Using Obviously Synchronizable Series Expressions Instead of Loops

Richard C. Waters
MIT Artificial Intelligence Laboratory
545 Technology Sq.; Cambridge MA 02139

Abstract
It has been known for a long time that series expressions (functional expressions on data aggregates) are easier to write and maintain than loops. For example,

\[ \text{Product}(\text{sqrt}(\text{PositiveElements}(V))) \]

is shorter and clearer than the equivalent loop

\[ \text{Prod} := 1; \]
\[ \text{for } I := 1 \text{ to LengthOfV do} \]
\[ \text{if } (V[I]) > 0 \text{ then} \]
\[ \text{Prod} := \text{Prod} * \text{sqrt}(V[I]); \]

However, as typically implemented, series expressions are much less efficient than loops. At the cost of placing modest limits on what can be written, obviously synchronizable series expressions solve this problem by guaranteeing that every series expression can be automatically converted into a highly efficient loop. As demonstrated by Lisp and Pascal prototypes, obviously synchronizable series expressions can be added to any programming language.

1. Series Expressions

A series is an ordered linear sequence of elements such as a vector, list, or stream. Most algorithms that can be expressed as loops can also be expressed as functional expressions on series. It is well known that, although series expressions require a shift in one’s point of view, they are more concise, more likely to be self-evidently correct, and easier to modify than loops.

As an example, consider the problem of computing a weighted average of the odd elements of a vector. This can be done using a loop as shown by the following Pascal [10] code fragment.

\[
\begin{align*}
J & := 1; \\
\text{Sum} & := 0; \\
\text{for } I := 1 \text{ to LengthOfV do} & \\
& \text{if odd}(V[I]) \text{ then} \\
& \text{begin} \\
& \quad \text{Count} := \text{Count} + 1; \\
& \quad \text{Sum} := \text{Sum} + V[I] \times \text{Weights}[J]; \\
& \quad J := J + 1 \\
& \text{end}; \\
\text{Result} & := \text{Sum}/\text{Count};
\end{align*}
\]

Alternatively, this average can be computed using a series expression as shown below. This expression makes use of five series functions: \text{Evector} which creates a series of the elements in a vector; \text{TselectF} which creates a series by selecting the elements of an input series that satisfy a predicate; \text{TmapF} which creates the ith element of its output by applying the function given as its first argument to the ith elements of its inputs; \text{Tsum} which computes the sum of the elements of a series; and \text{Length} which returns the number of elements in a series.

\[
\begin{align*}
\text{Weighted} & := \text{TmapF}(\times, \text{TselectF}(\text{odd, Evector}(V)), \\
& \quad \text{Evector}(\text{Weights})); \\
\text{Result} & := \text{Tsum}(\text{Weighted})/\text{Length}(\text{Weighted});
\end{align*}
\]

The key advantage of using a series expression in this situation is that the various common patterns of computation that are mixed together in the loop are distilled into function calls that can be understood in isolation from each other. For example, the pattern of initializing a variable to zero and then repetitively adding numbers into it is distilled into the function \text{Tsum} and the actions involved with looking at the successive elements of a vector is distilled into the function \text{Evector}. Looked at from this perspective, the concept of series functions continues the trend in language design initiated by iterators in CLU [11] and generators in Alphard [25].

Using series expressions is not a new idea. Such expressions have been around for a long time in languages like APL [12], where the example above can be written as:

\[
\begin{align*}
\text{WEIGHTED} & := ((1:2) \times V)/V) \times \text{WEIGHTS} \\
\text{RESULT} & := (+/\text{WEIGHTED}) : (\times \text{WEIGHTED})
\end{align*}
\]

and Common Lisp [15], where one can write:

\[
\begin{align*}
\text{(setq weighted} & := (\text{map 'vector} \times (\text{remove-if-not} ='\text{odd} v) \times \text{weights})) \\
& \quad (\text{setq result} (/ (\text{reduce} \times '\text{weighted}) \times (\text{length} \text{weighted})))
\end{align*}
\]

In addition, it has always been possible to write series expressions in more conventional languages such as Pascal. There are however, two very significant problems.

First, like most Algol-like languages, Pascal does not provide any predefined series functions and does not allow user defined functions to return aggregate data structures such as arrays. The effort required to implement a comprehensive set of series functions operating on series represented
using some kind of list-like, pointer structure quite naturally deters all but the most ardent enthusiasts from using series expressions.

Second, even as implemented in APL and Common Lisp, series expressions are 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 series expressions is the creation of intermediate series objects. This requires a significant amount of space overhead per element (to store them) and time overhead per element (to access them).

The key to evaluating series expressions efficiently is the realization that it is often possible to transform a series expression into a form where the creation of intermediate series is eliminated. For example, it is straightforward to transform the series expressions above into the loop shown.

This transformational approach to the efficient evaluation of series expressions has been used in a number of contexts. For example, it is used by optimizing APL compilers [4, 9], Wadler's Listless Transformer [16] which can improving the efficiency of programs written in a Lisp-like language, and Bellegarde's transformation system [3] which can improve the efficiency of programs written in the functional programming language PP. In addition, Goldberg and Paige [8] have shown that the transformational approach can be used to improve the efficiency of data base queries.

Unfortunately, it is not always possible to transform a series expression into an efficient loop. As a result, all of the systems above fall back on a strategy of transforming some parts of some series expressions while leaving other parts untransformed. The complexities involved with this strategy are illustrated by the fact that Goldberg and Paige have shown (see [8]) that making an optimal choice of which subexpressions of a series expression to transform is NP-hard.

Sad to say, a partial transformation strategy is not anywhere near as satisfactory from the user's perspective as guaranteed complete transformation would be. The difficulty is that a series expression typically remains quite inefficient unless it can be transformed completely. As a result, even using one of the systems above, programmers still hesitate to use series expressions, because although a given series expression may end up being evaluated efficiently, they cannot be sure that this will be the case. This problem is aggravated by the fact that none of the systems above is capable of giving any kind of clear feedback on how to modify a given expression so that it can be evaluated efficiently.

A solution to the problems engendered by series expressions that cannot be evaluated efficiently is to restrict the kinds of series expressions that are allowed so that the transformation of every series expression into an efficient loop is guaranteed. This allows programmers to write series expressions without worrying about whether the expressions will be efficient.

The use of restrictions is illustrated by the high level business data processing languages Hibol [14] and Model [13]. Each of these languages features series expressions prominently. However, each language implicitly places very strong restrictions on the kinds of series expressions that can be written. (In particular, each language only provides a small number of predefined series operations and users are not allowed to define new ones.) The implicit restrictions ensure that the compilers for Hibol and Model can produce very efficient code in most situations. However, since the restrictions employed are much stronger than they have to be, programmers are unnecessarily limited in what they can write.

**Obviously Synchronizable Series**

Obviously synchronizable series expressions (OSS expressions) are a restricted class of series expressions that have the property that every OSS expression can be transformed into an efficient loop. In addition, the restrictions underlying OSS expressions have been carefully chosen so that they are not very much stronger than necessary. The relationship between, loops, series expressions and OSS expressions is shown in Figure 1.

![Figure 1: Series expressions and loops.](image)

Each rectangle in the figure indicates the class of algorithms that can be reasonably expressed using the indicated approach. The emphasis here is on the word 'reasonably'. Given that each of the approaches is capable of simulating a Turing machine, they are all capable of representing any algorithm. However, there can be very wide disparities in the ease of expressing a given algorithm using the three approaches.

The top rectangle in Figure 1 represents the algorithms that can be reasonably expressed as loops. Most of these algorithms can be expressed significantly more conveniently as series expressions. However, there are some looping algorithms that cannot by reasonably expressed using series expressions. The main criterion here is that an algorithm cannot be usefully rendered as a series expression unless it can be broken up into two or more operations that can be understood in isolation from each other. At the other extreme, there are numerous series expressions that do not correspond to any reasonable loop.

The innermost rectangle represents the algorithms that can be conveniently represented using OSS expressions. The
primary point of Figure 1 is that this class of algorithms is quite close to the intersection of the other two classes. In addition, this class corresponds to a large percentage of looping algorithms.

Looking at the last point more carefully, it is interesting to note that the concept of OSS expressions grew out of the observation that there are a number of common looping algorithms that appear again and again. An informal study [18] revealed that approximately 80% of the loops programmers actually write are constructed by combining just a few dozen common looping algorithms. The OSS restrictions are chosen so that all of these algorithms can be represented as series expressions. As a result, it appears that approximately 80% of loops can be trivially rewritten as OSS expressions. Many more can be converted to this form with only minor modification.

Getting Rid of Loops

OSS expressions make it possible to obtain the advantages of using series expressions instead of loops without any loss of efficiency. Their long term potential can be summarized as follows:

OSS expressions are to loops as structured control constructs are to gotos.

Structured control constructs (if...then...else, case, while...do, repeat...until) are not capable of expressing anything that cannot be expressed as gotos. In addition, there are probably a few algorithms for which the use of gotos is preferable. However, 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.

OSS expressions have the exact 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 is a significant class of algorithms that can be expressed better as loops than as OSS 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 OSS 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.

Supporting OSS Expressions

Support for OSS expressions can be provided in any programming language as follows. First, an OSS data type and a set of predefined OSS functions are created. This is a purely additive extension that does not have to change any of the standard features of the language. Second, a preprocessor (or compiler extension) is implemented that checks the restrictions underlying OSS expressions and converts OSS expressions into loops.

A Common Lisp implementation of OSS expressions [22] has been in regular use since December, 1987. As shown in Section 2, a prototype has also been constructed that demonstrates how OSS expressions can be supported as an extension to Pascal.

Sections 3 & 4 describe the restrictions underlying OSS expressions and the algorithms that can be used to transform OSS expressions into loops. In the interest of brevity, the sections confine themselves to presenting only the most important points. A full discussion of these topics can be found in [23].

2. OSS Expressions in Pascal

The user-visible part of introducing support for OSS expressions into a language is the introduction of an OSS data type and predefined OSS functions. For this extension to fit in harmoniously, it needs to be consistent with the syntax of the language.

The OSS data type. The OSS data type is best provided in analogy with vectors as shown below. However, one should be sure to avoid making the mistake of specifying the length of an OSS series as part of its type. (Since OSS series do not require any physical storage this would limit the ability of OSS functions to operate on OSS series of arbitrary length without having any counterbalancing benefits.)

```
type Integers = oss of Integer;
var Inputstream: oss of Char;
```

Once a generic OSS data type has been introduced, OSS variables, functions, procedures, and expressions can be written using exactly the same syntax as ordinary variables, functions, procedures, and expressions. In particular, an OSS variable is simply a variable that holds an OSS series, and an OSS function (or procedure) is a simply function (or procedure) that has either an OSS input or an OSS output. However, one should be sure not to place any limits on the type of the output of an OSS function. (Since OSS functions do not have any existence at run-time—having been transformed into parts of loops—there is no need for such limitations.)

Predefined OSS functions. Just as a set of basic functions and operations must be provided for any other data type, a set of basic OSS functions must be provided. To be most useful, these functions should support a number of specific operations. A number of these operations are illustrated below. A more complete list can be found in [22].

OSS functions divide naturally into three classes. Enumerators produce series without consuming any. Transducers compute series from series. Reducers consume series without producing any. As a mnemonic device, the name of each predefined OSS function begins with a letter code that indicates the type of operation. These letters are intended to be pronounced as separate syllables.
There are two basic kinds of enumerators: ones that create a series containing the elements of an aggregate data structure and ones that create a series based on some formula. For example, \texttt{Eset} and \texttt{Evector} enumerate the elements of a set and a one-dimensional array respectively, while \texttt{Eup} enumerates the integers in a range. (In the examples below, the notation \(...\) is used to represent an OSS series.)

\begin{verbatim}
Eup(1, 3) ⇒ \langle 1, 2, 3 \rangle
Evector('aba') ⇒ \langle 'a', 'b', 'a' \rangle
Eset(['f', 'c', 't']) ⇒ \langle 'f', 'c', 't' \rangle
\end{verbatim}

Useful transducers include \texttt{Tsubseries} which selects a subseries of the elements of a series, \texttt{Tconcatenate} which creates a series by appending two series together, \texttt{Tselect} which selects elements from a series based on a series of boolean values, and \texttt{Tprevious} which takes in a series and shifts it over one element by inserting the indicated value at the front and discarding the last element.

\begin{verbatim}
Tsubseries(['a', 'b', 'z', 'x'], 2, 4) ⇒ \langle 'b', 'x' \rangle
Tconcatenate(['a', 'b', 'z'], \langle 'x' \rangle) ⇒ \langle 'a', 'b', 'x' \rangle
Tselect(true, false, true, \langle 1, 2, 3 \rangle) ⇒ \langle 1, 3 \rangle
Tprevious(['a', 'b', 'c', 't'], 3) ⇒ \langle 't', 'a', 'b' \rangle
\end{verbatim}

There are two basic kinds of reducers: ones that combine the elements of a series together into an aggregate data structure and ones that compute some summary value from these elements. Useful reducers include, \texttt{Rset} which combines the elements of a series into a set, \texttt{Rsum} which adds up the elements of a series, \texttt{Rlength} which computes the length of a series, and \texttt{Rlast} which returns the last element of a series or the indicated default value if there are no elements in the series.

\begin{verbatim}
Rset(['c', 'f', 't', 'r']) ⇒ \langle 'c', 'f', 't' \rangle
Rsum(\langle 1, 2, 3 \rangle) ⇒ 6
Rlength(\langle 'a', 'b', 'c', 't' \rangle) ⇒ 3
Rlast(\langle 'a', 'b', 'c', 't', 'r' \rangle) ⇒ 't'
\end{verbatim}

Higher-Order OSS functions. In addition to OSS functions like the ones above, it is very important to provide higher-order functions supporting general classes of OSS operations. This is essential to allow the user to define a wide variety of new OSS functions. (Each of the higher-order OSS functions ends in the suffix "F", which is pronounced separately.)

The function \texttt{EnumerateF(init, step, test)} supports enumeration in general. A series of elements is enumerated starting with \texttt{init} by repeatedly applying \texttt{step}. The series consists of the values up to, but not including, the first value for which \texttt{test} is true. (The examples below assume that the function \texttt{negative} tests whether or not an integer is less than zero and that the function \texttt{halve} divides an even integer by 2.)

\begin{verbatim}
EnumerateF(3, pred, negative) ⇒ \langle 3, 2, 1, 0 \rangle
EnumerateF(56, halve, odd) ⇒ \langle 56, 28, 14, 7 \rangle
\end{verbatim}

The function \texttt{ReduceF(init, function, items)} supports reduction in general. The elements of the series \texttt{items} are combined together using \texttt{function}. The quantity \texttt{init} is used as an initial seed value for the accumulation.

\begin{verbatim}
ReduceF(0, +, \langle 2, 3, 4 \rangle) ⇒ 9
ReduceF(1, +, \langle 2, 3, 4 \rangle) ⇒ 24
\end{verbatim}

The function \texttt{TmapF(function, items)} supports the generic transduction operation of mapping a function over one or more series. A series is computed by applying \texttt{function} to each element of the input series.

\begin{verbatim}
TmapF(sqr, \langle 2, 3, 4 \rangle) ⇒ \langle 4, 9, 16 \rangle
TmapF(+, \langle 2, 3 \rangle, \langle 4, 5 \rangle) ⇒ \langle 8, 15 \rangle
\end{verbatim}

The function \texttt{TselectF(predicate, items)} supports the generic transduction operation of selecting elements from a series. This function is identical to \texttt{Tselect} except that instead of operating on a series of boolean values, it uses \texttt{function} to determine the elements of \texttt{items} to select.

\begin{verbatim}
TselectF(odd, \langle 1, 2, 3, 4 \rangle) ⇒ \langle 1, 3 \rangle
TselectF(negative, \langle -1, 2, 3, -4 \rangle) ⇒ \langle -1, -4 \rangle
\end{verbatim}

Further Language Extensions

The facilities presented above provide minimal support for OSS expressions. They demonstrate that this can be done in close analogy with the preexisting features of Pascal. However, it is advisable to support a few additional features, which although not essential, make OSS expressions significantly more useful. In spirit, these additional features require somewhat greater extensions to Pascal. However, since OSS expressions are supported entirely by a preprocessor, no actual changes to Pascal are ever required.

Implicit mapping. Mapping is by far the most common series operation—probably accounting for more than half of all series computations. As a result, it is worthwhile to make mapping particular easy to express. This can be done by introducing the concept of implicit mapping. Whenever a non-series operation is applied to an OSS series, it is automatically mapped over the elements of the series. For example, in the expression below, the function \texttt{sqr} is mapped over the series of numbers created by \texttt{Eset}.

\begin{verbatim}
Rsum(sqr(Eset(\langle 2, 4 \rangle))))
\equiv Rsum(TmapF(sqr, Eset(\langle 2, 4 \rangle)))) ⇒ 20
\end{verbatim}

The key virtue of implicit mapping is that it reduces the number of functions that have to be explicitly defined. For example, you can write:

\begin{verbatim}
Rsum(ord(Evector('246'))-ord('0')) ⇒ 12
\end{verbatim}

instead of having to write:

\begin{verbatim}
function digit (C: char): integer;
begin
digit := ord(C)-ord('0')
end;

Rsum(TmapF(digit, Evector('246'))) ⇒ 12
\end{verbatim}
Generic OSS functions. Just like basic Pascal operations such as +, succ, and array referencing, most of the basic OSS operations are naturally applicable to many different types of objects. For example, Reum is naturally applicable to a series of any kind of numbers and TmepF is applicable to any kind of series at all. The higher-order OSS functions are even more flexible in the way they can be applied. They merely require that the types expected by their functional arguments must be consistent with the types of their other arguments. (Like array referencing, TmapF goes further by allowing there to be a variable number of arguments.)

If one were to insist that strong typing had to apply at the level of the predefined OSS functions, each basic OSS function would have to be expanded into a large family of functions that are identical except that they have different type signatures. It is much better to treat the predefined OSS functions as generic functions in analogy with the way the basic Pascal operations of +, succ, and array referencing are treated.

An Example

Within the limits of the space available, the following example attempts to give a feeling for how OSS expressions can best be used. The example revolves around a job queue data abstraction which might be used in an operating system. The basic type definition is shown below. A JobQ is a pointer to a chain of entries that point to records describing jobs. These records have a number of fields including a numerical priority.

```pascal
type JobInfo = record job: Job; rest: JobQ end;

type JobQentry = record; job: JobInfo; rest: JobQ end;

type JobQ = record; job: JobInfo; rest: JobQ end;

type JobRecord = record priority: real; ... end;
```

There are a number of functions defined that operate on job queues. These functions 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.

```pascal
procedure AddToJobQ (J: JobInfo; var Q: JobQ);
var E: JobQentry;
begin
  new(E);
  E.job := J;
  E.rest := Q;
  Q := E
end
```

In addition to ordinary functions that operate on job queues, it is useful to define a number of OSS functions that operate on job queues. For example, the function EJobQtails enumerates all of the tails of a queue (i.e., (Q, Q.rest, Q.rest.rest, ...)). It is implemented using the higher-order OSS function Enumerates and two special purpose functions operating on job queues.

```pascal
function EJobQtails (Q: JobQ): oss of JobQ;
function JobQrest (Q: JobQ): JobQ;
begin
  JobQrest := Q.rest end;
function JobQnull (Q: JobQ): Boolean;
begin
  JobQnull := Q=nil end;

begin
  EJobQtails := Enumerates(Q, JobQrest, JobQnull)
end
```

The function EJobQ enumerates the jobs in a queue. It is implemented using EJobQtails. The expression Qs.job causes the operations of following a pointer and selecting the job field of a JobQentry to be implicitly mapped over the pointers returned by EJobQtails.

```pascal
function EJobQ (Q: JobQ): oss of JobInfo;
var Qs: oss of JobQ;
begin
  Qs := EJobQtails(Q);
  EjobQ := Qs.job
end
```

The function RemoveFromJobQ removes a job from the end of a queue. It is implemented straightforwardly using the OSS function EJobQtails. To start with, RemoveFromJobQ enumerates the tails of the queue and uses the standard OSS functions Rlast and Tprevious to obtain a pointer to the next to last entry in the queue. The rest pointer in this entry is set to nil to remove the last entry from the queue. (If there is no next to last entry, then the queue variable itself is set to nil.) RemoveFromJobQ then locates the last entry in the queue and frees the storage associated with it, returning the contents of its job field. (It is assumed that there must be at least one job in the queue.) From an efficiency standpoint, it is interesting to note that since there is only one instance of EJobQtails, the OSS preprocessor will create code that only traverses the queue once.

```pascal
function RemoveFromJobQ (var Q: JobQ): JobQ;
var Qs: oss of JobQ;
begin
  Qs := EJobQtails(Q);
  NextToLast := Rlast(Tprevious(Qs, nil), nil);
  if NextToLast=nil then Q := nil
  else NextToLast.rest := nil;
  Last := Rlast(Qs, nil);
  RemoveFromJobQ := Last.job;
  dispose(Last)
end
```

As a final example of the use of OSS expressions, consider the program SuperJob below. This function 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 all of the jobs in the queue. If there is no such job, nil is returned. The first five statements in the function compute the mean and deviation of the priorities. The next two statements select the jobs that have sufficiently large priorities. The last line selects the last of these jobs, if any.

```pascal
function SuperJob (Q: JobQ): JobQ;
begin
  JobQnull := Q=nil end;

begin
  EJobQtails := Enumerates(Q, JobQrest, JobQnull)
end
```
function SuperJob (Q: JobQ): JobInfo;
  var Jobs, SuperJobs: oss of JobInfo;
  N: Integer;
  Mean, SecondMoment, Deviation, Limit: Real;
begin
  Jobs := EJobQ(Q);
  N := Rlength(Jobs);
  Mean := Rsum(Jobs.priority)/N;
  SecondMoment := Rsum(sqr(Jobs.priority))/N;
  Deviation := sqrt(SecondMoment-sqr(Mean));
  Limit := Mean+2*Deviation;
  SuperJobs := Tselect(Jobs.priority>limit, Jobs);
  SuperJob := Rlast(SuperJob, nil)
end

The programs above are a good example of the way OSS 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 OSS expressions is to convert the vast majority of programs that are in fact straightforward programs into dirt simple programs, rather than to convert the few programs that are truly complex into less complex 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 OSS expressions.

3. The Restrictions

There are seven restrictions that underlie OSS expressions. However, both theoretically and pragmatically, two of these restrictions are much more important than all of the rest. These two restrictions, and the reasons behind them, are presented below. The full set of restrictions is discussed in detail in [23].

The key source of inefficiency when evaluating series expressions is the creation of physical intermediate series objects. For a series expression to be evaluated so that intermediate storage is not required, the expression must be synchronizable. (A series expression is synchronizable if the functions in it can be evaluated in parallel so that each time an element in a series is created, all of the computations that use this element finish using it before the next element in the same series is computed.)

Series functions must be preorder. For synchronization to be achieved, it must be the case that series elements are always created in the same order that they are used. The best way to ensure that this will be the case is to require that every series function be preorder. (A series function is preorder if it processes the elements of each series input and output in order starting with the first element.)

This restriction is stronger than it needs to be, however, this is not a problem in practice, because most commonly used series functions can be straightforwardly and efficiently implemented as preorder functions.

A key problem remains. Even if all of the functions in a series expression are preorder, it still may not be possible to synchronize the evaluation of the expression. The basic problem is that if an intermediate series is used by two functions, these functions may place incompatible requirements on the times at which the elements must be available. This problem is illustrated by the code fragment below.

function TselectF(odd, Values: JobQ): JobInfo;

var Values, Results: oss of real;

Results := Values/Rsum(Values);

This code creates a series of results by dividing each element in a series by the sum of the elements in the series. The difficulty is that the result of Tsum cannot be determined until after all of the elements of Values have been computed and the division processes cannot begin using the elements of Values until after the sum is known. As a result, all of the elements of Values have to be saved in intermediate storage so that they will be available when the division process needs them.

From the example above, it might appear that a failure of synchronizability can only arise when a reducer is involved. However, it can also be caused by a transducer as shown below. Here the difficulty is that TselectF does not necessarily produce an output element every time it reads an input element. Whenever an input element is skipped, the element has to be saved in intermediate storage so that it will be available for eventual use by the division process.

Results := Values/TselectF(odd, Values);

Requiring series expressions to be trees. As shown by Wadler [17], one way to prevent failures of synchronizability from arising is to require that every series expression be a tree. (An expression is a tree if every output value is used by at most one other function.)

Since no series is used in more than one place in a tree, there cannot be any competing requirements as to when elements should be computed. The evaluation of a series expression that is a tree can always be synchronized using lazy evaluation [7]. Series elements are simply computed when, and only when, the function that uses them wants to use them.

Unfortunately, it would not be reasonable to actually require that every series expression be a tree, because this requirement is much more restrictive than necessary. In particular, it is often possible to synchronize the evaluation of an expression even if some of the intermediate values are used in several places. As a result, from the point of view of readability and efficiency, it is unreasonable to require that a quantity that is used in n places must always be computed n times.

Requiring series functions to be on-line. Another way to prevent failures of synchronizability from arising is to require that every series function be on-line. (More or less following the standard definition of the term [1], a series function is on-line if it has a series output, and the i-th element of the output is always computed after reading the i-th element of every series input and before reading the next element of any input. Functions that are not on-line are said to be off-line.)

343
If every function in a series expression is on-line then, if an intermediate series is used by more than one function, these functions cannot disagree as to when the elements should be computed. The evaluation of a series expression consisting solely of on-line functions can always be synchronized by evaluating everything in lock step. On each step, the functions are evaluated one at a time in an order that is compatible with the data flow between them. Each time a function is evaluated, it is simply run until it produces a new output element.

Unfortunately, requiring that every series function be on-line would be even more unreasonable than requiring that every series expression be a tree. To start with, this would imply that series expressions could not contain reducers. In addition, although many common transducers (e.g., `mapF`) are on-line, many others (e.g., `selectF`) are not.

A compromise. Fortunately, it is possible to state a very useful restriction that is essentially a compromise between the two overly strong restrictions examined above. The essence of this compromise is the realization that typical expressions have some parts that are tree-like and other parts that are not. The on-line requirement only has to be applied to those parts of an expression that fail to be tree-like.

To make the compromise above precise, it is necessary to generalize the concept of on-line processing so that it applies to individual input and output ports instead of whole functions. To start with, every non-series port is off-line. In addition, a series port is off-line, if the processing at that particular port causes the containing function to be off-line. For example, the output of the function `select` is off-line while its two inputs are on-line. (The inputs are on-line, because the processing of the inputs always occurs in lock step—every time an input is read through one input, it is also read through the other.)

It is necessary to state precisely what it means for part of an expression to be tree-like. This idea is defined somewhat indirectly by introducing the concept of what it means for an input or output port to be isolated. (A port is isolated if the data flow graph corresponding to the containing expression can be cut into two pieces so that all of the data flow arcs connected to the port are cut and no other series data flow arcs are cut.)

The concept of isolation is illustrated by Figure 2. In the figure, the boxes represent the functions in a series expression. The boxes are connected by thick lines indicating series data flow and thin lines indicating non-series data flow. The vertical dashed lines indicate two possible ways in which the graph can be cut. The cut on the left shows that the output of `e`, the input of `f`, and the bottom-most input of `h` are isolated. The cut in the middle shows that the output of `f` is isolated. The other ports in the figure are not isolated. For example, the input of `g` is not isolated, because any cut that severs the series data flow from `f` to `g` must also sever the series data flow from `f` to `h`.

Given these definitions, the compromise above can be stated as follows. If every off-line port of a series expression is isolated then the expression is synchronizable.

The essential reason why this is true is that whenever a port is isolated, the containing expression can be cut into two pieces that can be synchronized separately. The evaluation of the pieces can then be recombined using lazy evaluation. The ground case of this divide and conquer strategy calls for the synchronous evaluation of expressions that contain only on-line ports. This can be done by evaluating everything in lock step.

**Enforcing the restrictions.** Most of the restrictions underlying OSS expressions can be enforced by setting things up so that it is impossible for a programmer to violate them. For example, since every predefined OSS function (including the higher-order ones) is preorder and the composition of two preorder functions is preorder, it is impossible for a programmer to construct an OSS function that is not preorder.

Other restrictions (in particular the off-line port isolation restriction) have to be checked explicitly by the OSS preprocessor. However, simple OSS expressions are very unlikely to violate any of the restrictions. In particular, it is impossible for an OSS expression to violate any of the restrictions unless it contains an OSS variable.

In addition, it is possible for the preprocessor to automatically fix expressions that violated the restrictions. As a result, programmers can write OSS expressions without having to worry about the restrictions at all.

However, even though violations of the off-line port isolation restriction can be automatically fixed by the preprocessor, warnings are issued whenever a violation is detected. This is done on the theory that it is important for programmers to get accurate feedback in situations where the evaluation of an OSS expression is not going to be as efficient as it might appear. In addition, it is often possible for the user to fix the problem in a way that is much more efficient than the default action taken by the preprocessor.

4. Algorithms

The primary contribution of the work reported here is the set of restrictions that underlie OSS expressions. Given these restrictions, the algorithms that actually convert OSS expressions into loops are surprisingly straightforward. Inasmuch as this is the case, the discussion below gives only a brief sketch of the algorithms. Complete details can be found in [23].

The preprocessor has four parts: a parser which locates
OSS expressions, an implicit mapper which determines what subexpressions should be mapped, a transformer which converts the OSS expressions into highly efficient loops, and an unparscr, which creates Pascal code corresponding to the result.

Parsing. Pascal programs are parsed using the standard grammar for Pascal augmented with the extensions discussed in Section 2. The parse tree is scanned to locate OSS expressions along with the definitions of OSS functions and variables. The OSS expressions are then passed to the implicit mapper.

Implicit mapping. The implicit mapper takes a parsed OSS expression and turns it into a data flow graph which represents the way the various functions are connected by data flow. Both nesting of subexpressions and the use of variables lead to links in this graph. Based on an inspection of the graph, implicit mapping is introduced whenever a non-OSS function receives an OSS input.

Transformation. The transformation process operates in several steps. In the first step, the data flow graph produced by the implicit mapper is partitioned into subexpressions by cutting it at each off-line port. As part of this process, the transformer checks that the expression obeys the off-line port isolation restriction. Whenever a violation is detected a warning is issued and the problem is fixed by duplicating sections of the expression until isolation is achieved.

Once partitioning is complete, the functions in each partition subexpression are combined together into a single operation. To facilitate this process, each OSS function is represented as a loop fragment consisting of several parts including: a prolog containing statements that are executed before the computation starts; a body containing statements that are repetitively executed; and an epilog containing statements that are executed after the loop terminates.

The fragments corresponding to the functions in a partition subexpression are combined by simply concatenating the parts of the corresponding fragments. The data flow between the functions is implemented by renaming the variables carrying input values and output values. This method of combination leads to lock-step evaluation of the various fragments and is much the same as an application of the standard compiler optimization technique of loop fusion [2].

Once the partition subexpressions have been reduced to single operations, they are combined using a method that supports lazy evaluation. For example, when two operations are connected by an OSS series data flow terminating on an off-line input, the body of the source fragment is inserted into the body of the destination fragment at the place where the value is to be used. This allows the destination fragment to control exactly when the source will be evaluated. (The off-line port isolation restriction guarantees that the body of the source fragment will only have to be inserted into one place.)

Once all of the fragments representing the functions in an OSS expression have been combined into a single fragment, this fragment is converted into a loop.

Unparsing. Once a loop is created, it is inserted into the parse tree in place of the corresponding OSS expression. Once every OSS expression has been converted into a loop, the unparscr generates a Pascal program corresponding to the altered parse tree. This program can then be passed to a Pascal compiler.

Systems based on similar algorithms. The OSS macro package as a whole (and the transformer component in particular) is descended from an earlier macro package called Lets [19, 20]. Lets is similar to the OSS macro package in many ways, however, it is less powerful and less clear in its focus, because it is based on an unnecessarily strict set of ad hoc restrictions. (A system intermediate between Lets and OSS expressions as presented in this paper is described in [21].)

The same basic approach to representing and combining series functions was independently developed by Wile [24]. However, he does not explicitly address the question of restrictions and his approach does not guarantee that every intermediate series can be eliminated.

A quite similar approach is also used internally by the Loop macro [5]. However, Loop is quite different from the OSS macro package. To start with, it uses an idiosyncratic English-like syntax rather than representing computations as combinations of functions operating on series. In addition, the Loop macro is implicitly based on restrictions that are much stronger than they need to be.

5. Conclusion

There are three principal perspectives from which to view OSS expressions. First, OSS expressions can be looked at as embodying most of the series processing capabilities of languages such as APL and Lisp in such a way that these capabilities can be included in other programming languages without requiring the use of unusual syntax or causing inefficiency. In addition, because a larger number of higher-order series functions are provided, OSS expressions support a wider range of series operations than either APL or Lisp.

An alternate perspective focuses on the fact that programmers can be confident in the fact that every OSS expression will be evaluated efficiently. This is an example of what Cai and Paige have referred to as "binding performance at language design time" [6]. This does not mean that everything programmers write is miraculously converted into efficient code. Rather, it means that programmers are encouraged (via prompt and accurate feedback) to think of ways of expressing what they want in a way that can be computed efficiently. In particular, unlike APL or Lisp, programmers are never tempted to think that all series expressions are equally efficient.

A final perspective is summarized by the statement that OSS expressions are to loops as structured control constructs are to gotos. By using OSS expressions, it is possible to banish loops from most programs. Given that expressions are much easier to understand and modify than loops, this is potentially a step forward at least as important as banishing gotos.
Acknowledgments. The development of obviously synchronizable series expressions benefited from the assistance of a number of people. In particular, C. Rich, A. Meyer, Y. Feldman, D. Chapman, and P. Anagnostopoulos made suggestions that led to a number of very significant improvements in the clarity and power of obviously synchronizable series expressions. Support for the research reported here was provided by IBM, NYNEX, Siemens, National Science Foundation grant IRI-1811644, and the Advanced Research Projects Agency of the Department of Defense under ONR contract N00014-85-K-0124.

References

Programs as Data Objects, Lecture Notes in Computer Science 217. H. Ganziger and N. Jones (eds), Springer-Verlag, New York, 1985