CSAIL

Computer Science and Artificial Intelligence Laboratory

Massachusetts Institute of Technology

Stream Data Types for Signal Processing

Jack Dennis

1994, October

Computation Structures Group
Memo 362

The Stata Center, 32 Vassar Street, Cambridge, Massachusetts 02139

LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Stream Data Types for Signal Processing

Computation Structures Group Memo 362
October 27, 1994

Jack B. Dennis

This memo is a chapter for the book, *Advances in Dataflow Architecture and Multithreading*, J.-L. Gaudiot and L. Bic, editors. To be published by IEEE Computer Society Press.

The research reported in this docment was performed, in part, using facilities of the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-92-J-1310.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Stream Data Types for Signal Processing

Jack B. Dennis
Associate Member, MIT Laboratory for Computer Science
Cambridge, Massachusetts, USA

October 27, 1994

Abstract

Streams of integers and streams of integer arrays are natural representations for the signals processed in speech analysis, image analysis, and seismic exploration, among other computer applications. In this paper we show how typical signal processing operations may be expressed in functional programming languages as tail-recursive functions using stream data types. Several programming styles are considered, with emphasis on illustating the support for streams proposed for the Sisal 2 language. A signal processing application often decomposes as a set of modules that transform signals (data streams), where pairs of modules are connected by links and operate in producer/consumer mode. Such compositions of modules are readily expressed in a functional programming language as a composition of recursive functions operating on streams. One issue in compiling such programs into efficient machine code is the recognition of recursion schemes that may be transformed into non-recursive dataflow graphs. Another issue is recognizing when finite buffers may be used between processing modules without introducing the possibility of deadlock. These issues are treated for an important class of signal processing programs and it is suggested that multiprocessor computers with fine-grain scheduling capability will prove to be attractive for these computations.

1 Introduction

A *stream* is a sequence of values which may be infinite (unending); a stream of integers is a natural representation for a signal that has been converted into digital form. Interconnecting modules that process streams of data is a powerful means for combining program parts to build larger modules and is well matched to the needs of signal processing tasks. However, stream data types have seen little use in practical signal processing applications because programming languages generally do not provide support for streams, and because implementations of sufficiently high performance to meet the demands of applications are not available.

In this article we illustrate the use of stream data types, as has been proposed for the Sisal 2 functional programming language, to express typical signal processing operations as recursive functions on streams. We show how the producer/consumer type of concurrency that occurs naturally in signal processing may be exposed and exploited by transforming the recursive schemes into (non-recursive) dataflow graphs. In this form, a multiprocessor computer built of multithreaded procesing units is an attactive implementation vehicle.

Sisal 2[18] is a proposed extension of the Sisal language[16] and is a functional programming language intended to support high performance execution of scientific codes on highly parallel computers. Sisal was developed at the Lawrence Livermore National Laboratory and has been used to express a variety of substantial scientific application codes. Sisal evolved from the Val language developed by the Computation Structures Group at the M.I.T. Laboratory for Computer Science[2].

2 Stream Data Types and Operations

In Sisal a stream data type `T2` may be created for any type `T1` by writing

```
type T2 = stream [ T1 ]
```

This means that values of type `T2` are streams (sequences of indefinite length) of elements of type `T1`. Three basic operations are provided for stream data types. Sisal provides the operations `stream_first` and `stream_rest` for accessing the first element of a stream and for defining a stream consisting of the remaining elements (all but the first) of the given stream. The Sisal concatentate operation, denoted by `||`, may be used to form a stream as a combination of given streams, for example

```
s2 := stream T [x]  || s1;
```

in which

```
stream T [x]
```

defines a stream of a single element `x` of type `T`. The result `s2` has `x` as its first element followed by the elements of stream `s1`. These operations are related by

```
s = stream T [ stream_first (s) ] || stream_rest (s)
```

Elements of a stream may also be accessed using subscript notation, as in the familiar syntax for array elements. The first element of a stream always has the index 1, so, for example

```
s[1] = stream_first (s)
```

and

```
s[2] = stream_first ( stream_first (s) )
```

3 Recursive Stream Processing Functions in Sisal

It is natural to write stream processing algorithms as recursive functions that define a result stream as the result of concatenating a new element with the stream produced by a recursive application of the function. The examples used below are based on simplified algorithms taken from a large-scale defense application studied by the Boeing Company. In a later section we will show how the algorithms may be combined to define a complete process suitable for execution by a massively parallel computer.

3.1 Example: Averaging Samples of a Signal

The first example is the program in Figure 1. Each element of the result stream is the average value of two adjacent elements of the input stream. (It is a simple a finite impluse response (FIR) filter.) This is a straightforward use of tail recursion to represent the incremental construction of a stream of integers from a given stream. The tail-recursive operator in this case is the concatenation of one element at the head of the result stream.

There is little difficulty in uderstanding that this function definition correctly defines the result sequence in terms of an input sequence. However, if this function were evaluated using conventional implementations of recursion, the execution would perform an endless loop, generating a forever growing set of stack frames! On the other hand, this function is a special form of tail recursion that can be translated into a static dataflow graph that has a small, fixed storage requirement (see below).

```
type Signal = stream [integer];

function AveragePairs ( D: Signal returns Signal )

   stream integer [ (D[0] + D[1]) / 2 ]
         ||  AveragePairs ( stream_rest (D) )

end function
```

Figure 1: Stream function to average pairs of stream elements.

This example has the property that (after an initial transient) one element is added to the output stream for each new element accessed in the input stream. The next example does not have this property.

3.2 Example: A Rate Changer.

A frequent requirement in signal processing is to convert a signal to a different sampling rate. The stream function in Figure 2 produces four samples for each group of three samples in the input stream, thereby increasing the sampling rate by the factor 4/3. Each sample of the result is obtained by linear interpolation between the adjacent samples of the input stream. The function `Tail (p, s)` returns the stream obtained by removing p head elements from the stream s and may be implemented by p `stream_rest` operations.

3.3 Dataflow Graphs for Stream Processing Functions

The Sisal programs for stream processing functions do not indicate explicitly the concurrency that should be exploited; this is determined by the language implementation. Present compilers for Sisal have not emphasized high performance in stream processing becuase the kernels of scientific applications are mainly array-defining modules that make no use of stream data types.

The most general implementation to support stream data will require dynamic memory management, leading to considerable overhead cost on conventional computer systems. However, many signal processing applications, including the examples in this article, can be implemented using only statically allocated storage. We show this by converting the recursive stream

```
type Signal = stream [integer];

function FourForThree ( D: Signal returns Signal )

   let
      n1 := D[1];
      n2 := ( D[1] + 3 * D[2] ) / 4;
      n3 := ( D[2] + D[3] ) / 2;
      n4 := ( 3 * D[3] + D[4] ) / 4;
   in
      stream integer [ n1, n2, n3, n4 ]
            ||  FourForThree ( Tail (3, D) )
   end let

end function
```

Figure 2: The rate changer function written in Sisal.

functions into (static) dataflow graphs[7]. (Use of dataflow diagrams in signal processing goes back at least to[13] and has been studied extensively by Lee[12, 15]. Recent work includes[11].) We give a general transformation scheme in the next section.

The `AveragePairs` function may be described graphically as in Figure 3. The function body has three parts: one that extracts some head elements from the input stream; one that performs a computation on these element values; the third component is the concatenate operator that may be regarded as *affixing* the computed element at the head of the result stream and following it with the result stream from a recursive call of the function. The graphical scheme shown in Figure 3 is a form of recursive dataflow graph[7, 19].

Figure 4 shows an equivalent static dataflow graph. The identity and gate actors in the box labelled Group extract successive groups (pairs) of elements from the input stream and present them to the Compute component. The Compute component is exactly the same as its counterpart in the recursive scheme, except it must be able to process successive sets of data (by pipelining, perhaps). In this example, the output stream consists of the successive elements computed.

Figure 5 shows a dataflow graph for the rate changer stream function. Again, the switch actors on the left access groups of four elements from the

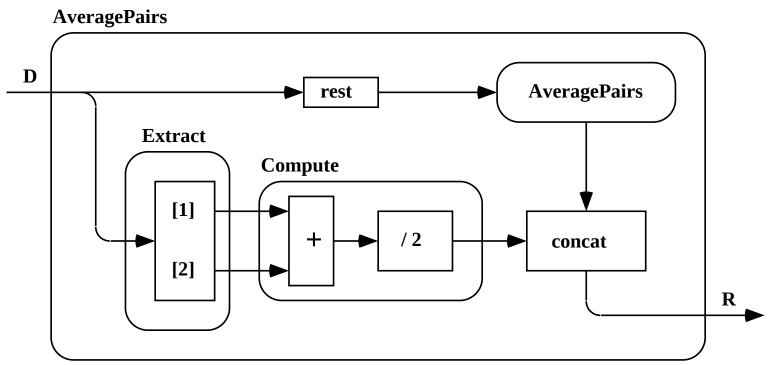

Figure 3: Recursive scheme of the averaging function.

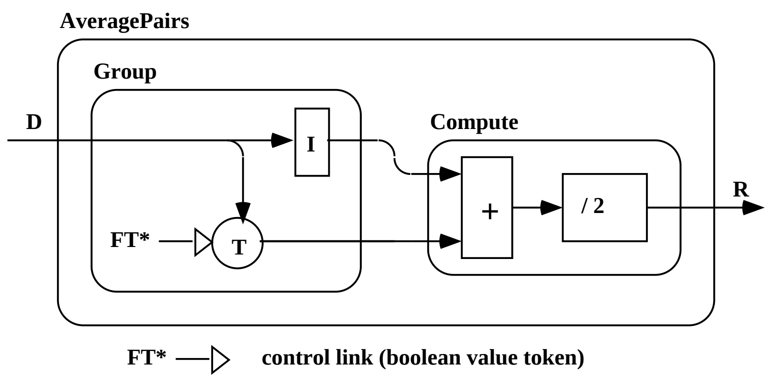

Figure 4: Dataflow graph for the averaging function.

input stream at positions separated by three elements. The merge actors on the right place the four computed values in the output stream. The control inputs to the gate, switch, and merge actors are specified by regular expressions on the alphabet {true, false} in lieu of showing configurations of dataflow actors that generate them. The figure shows the Compute box as a coefficient matrix. Each group (vector) of four input samples is multiplied by the matrix to yield the corresponding 4-vector of output samples.

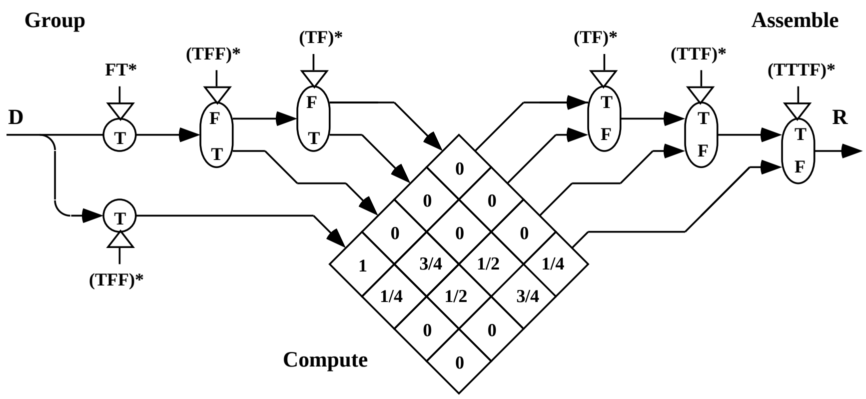

Figure 5: Dataflow graph for the rate changer function.

4 Translation of Stream Functions into Dataflow Graphs

A general scheme for recursive stream processing functions is shown in Figure 6. We consider only tail-recursive functions in which the body, consisting of the Compute and Auxiliary boxes, is *well-behaved*, that is, it produces a single set of output values for each set of input values and the body is not history-sensitive. Besides the stream input `D`, the function may have a tuple `S` of additional inputs of arbitrary type. The pattern of operation of this general scheme is defined by three integers p, q, and m. At each level of recursion, the function accesses m elements at the head of the input stream and emits q elements of the result stream `R`. The remainder of the result stream is the result of applying the stream function recursively to the input stream with p head elements removed. The function body contains an arbitrary function Compute with m inputs and q outputs, which defines elements of the result stream. The arbitrary function Auxiliary defines the tuple $\texttt{S}'$ of additional input values for the next deeper level of operation.

The transformed (dataflow) scheme is shown in Figure 7. The Group box corresponds to the Extract box in Figure 6. It forms groups of m elements from the input stream, starting at indices $1, 1+p, 1+2p, \ldots$ and presents them to the Compute and Auxiliary boxes. The Assemble box takes successive groups of q elements defined by the Compute box and concatenates them to form the output stream. The additional inputs to the Compute and

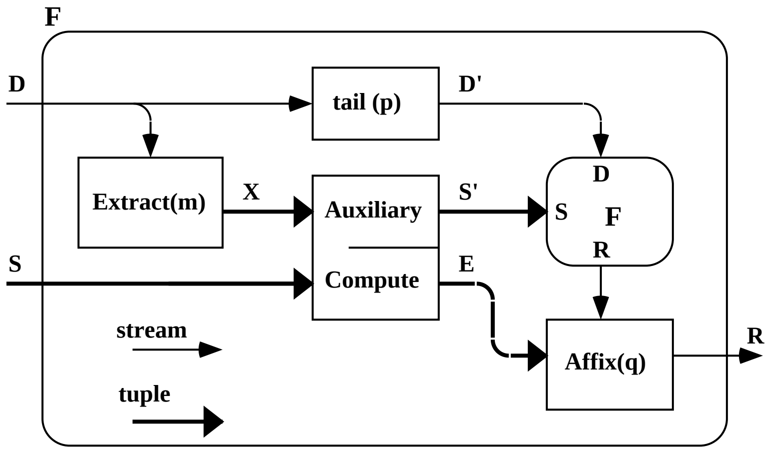

Figure 6: General form of recursive stream function.

Auxiliary boxes are initially supplied from the additional schema inputs, but come from the outputs of the Auxiliary box on subsequent iterations.

The construction of the Group and Assemble modules is illustrated by the specific constructions shown for the rate changer in Figure 5. Proof of equivalence may be done by an induction, provided in the appendix, showing that the successive sets of values computed in the dataflow scheme are identical to the sequences of sets of values occurring at successive levels of recursion in the recursive scheme. The proof extends to stream processing modules that have several input and output data streams.

5 Composition of Stream Functions

Complete signal processing tasks often take the form of a set of processing modules, each generating a stream of values that is passed to other modules for further processing. Thus the overall computation may be described by an acyclic graph in which the nodes are stream processing modules such as those we have presented, and each link indicates a producer/consumer relationship between a pair of modules. It is well-known that such interconnections of modules may lead to deadlock if the temporary storage for stream elements in each link is bounded in capacity.

If each node in an acyclic composition of stream processing modules has

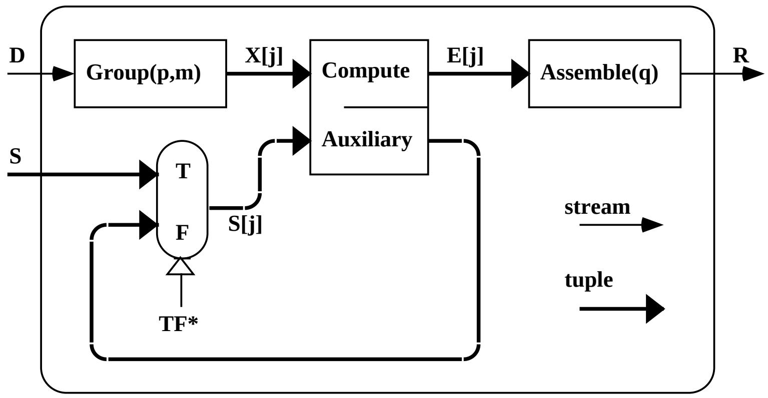

Figure 7: General form of the transformed stream function.

the structure given in Figure 6, then each node may be characterized by a *gain* that is the ratio q/p of tokens produced to tokens consumed. The gain for a (directed) path in the graph is the product of the gains for each node in the path. A necessary and sufficient condition that an acylic composition of such stream processing functions be free of deadlock is that for any pair of nodes a and b, all directed paths from a to b must have the same gain. A test of this condition may be incorporated into a Sisal compiler to warn the programmer if his program will deadlock.

6 Stream Computations in Other Languages

Functional programming languages are characterized by "referential transparency", a piece of program text has the same meaning regardless of the context in which it appears, and freedom from "side-effects", the notion that arguments and results of a program module are distinguished and argument values to a module instantiation do not change. These concepts provide functional programing languages with two advantages. The first is that programming is easier and more productive because programs are simpler, closer to mathematics, and easier to understand. The second is that functional programs are far easier for compilers to analyze into parts that may be executed concurrently on parallel computers.

The idea of programming with streams is old, having been described by Landin in 1965[14]. However, most presentations of programming with streams are in the context of languages influenced by their implementations on conventional sequential computers.

In some languages the concept of stream is introduced as an application of lazy lists. A stream is represented by a linear list structure so that the CAR of the list is its first element and the CDR of the list represents the stream consisting of the remaining elements. The problem with the usual implementaion of lists is that a stream represented as a list will not be accessible to a using program module until the list is completely constructed by the list generating module. This leads to needless memory demands and the impossibility of handling infinite streams, which are the usual form of data in signal processing.

A solution to this dilemma is to represent the remaining elements of a stream by an object variously called a "future" or a "promise to compute on demand". For example, the pair averaging function may be written in Scheme[1] as

```
(define (average-pairs D)
   (cons
      (divide (plus (car D) car (force (cdr D))))
      (delay (average-pairs (cdr D)))
   )
)
```

The `delay` operator defers the recursive call of `average-pairs` until access to the `cdr` of the list element is attempted by a consuming steam function. The `force` operator must be used to call for evaluation of a list component that may not have been computed yet. Treating all lists as composed of components to be evaluated on demand (the "lenient CONS") was suggested by Friedman and Wise[10]. The functional programming language Miranda[3] has embodied this concept of universal lazy evaluation so that use of special operators is not necessary to avoid waste of memory in stream processing. The pair averaging function may be written in Miranda as

```
average-pairs ( e1 : e2 : d )  =
        ( e1 + e ) / 2 : average-pairs ( e2 : d )
```

In this illustration, the colon stands for the associative list constructor (CONS) and pattern matching is used to detect when sufficient elements of the input list are available to define more output.

A major drawback of Scheme and Miranda is the difficulty of exploiting the opportunities for parallelism offered by acyclic compositions of stream processing modules. Correct interpretation of Scheme programs calls for a co-routine-like execution which must be honored because Scheme (and Lisp) are not free of side-effects. Thus there is always a single locus of control. It seems that Miranda implementations are similar because this author is not aware of any efforts to develop parallel implementations of Miranda, and permitting "eager beaver" evaluation to achieve concurrent execution would alter Miranda semantics.

In contrast, Sisal is one of few languages that introduce streams as an explicit type generator, is free of side-effects, and is intended to support parallel implementations. The functional language Id[17] has similar goals, but does not include a stream type generator. Instead it provides support for lazy lists and eager evaluation. The operational mechanism to support streams by this combination of lazy and eager evaluation has been studied by Dennis and Weng[6, 5].

7 Image Processing: Streams of Arrays

The elements of the stream being processed need not be simple scalar values. The next two examples illustrate how operations on images may be represented in a way that allows massively parallel processing of image data. Typical image information takes the form of a sequence of frames or scans. It is often convenient to view the input data as an array of streams where each stream contains data for a particular line in successive frames or scans.

7.1 A Two-Dimension Filter

The function `TwoDimFilter` shown in Figure 8 represents a two-dimension filter by a single Sisal function. The filter is defined by a three-by-three array `Filter` which is applied at each position in the image data for which an output value is desired. The input is an array of streams indexed from 1 to w. The output is an array of streams indexed from 2 to $w - 1$. (The boundary elements are omitted from the result data to avoid applying the filter function to non-existing array positions.)

As written, this function leads to duplicate computation of many intermediate values. This may be avoided, but requires more complex code[4] which would not suit the purposes of the present exposition.

```
type ImageStream = array [ stream [integer] ];

function TwoDimFilter (
      D: ImageStream, w: integer
      returns ImageStream )

   let
      Filter := array [-1:
         array [-1: 1, 2, 1 ],
         array [-1: 2, 3, 2 ],
         array [-1: 1, 2, 1 ],
         ];
      Dn := for i in 2, w-1
      return array of
         for g in -1, +1 cross h in -1, +1
         return value of sum Filter[g, h] * D[g+i, h+2]
         end for
      end for
      Dt := for i in 1, w
      return array of
         stream_rest ( D[i] )
      end for
      Dr := TwoDimFilter ( Dt, w );
   in
      for i in 2, w-1
      return array of
         stream integer [ Dn[i] ] || Dr[i]
      end for
   end let

end function
```

Figure 8: Two-dimension background filter in Sisal.

7.2 A Peak Detector Algorithm

Figure 9 shows a PeakDetect function that identifies all elements of the (image) data that have a value that is at least equal to the values of all immediate neighbors and exceeds their average by a given threshhold `Th`. The two conditions are tested separately and combined to determine the result. The input is an array of integer streams indexed from 2 to $w-1$. The output stream is an array of boolean streams indexed from 3 to $w-2$. The peak detection fuction is similar in structure to the filter function; each element of the result is true if and only if the data surronding the corresponding input pixel satisfies the specified conditions. As in the case of the two-dimension filter, a more complex code may be constructed that avoids recomputation of intermediate results.

8 Composition of Stream Functions

The stream functions we have described may be combined as shown in Figure 10 to form a complete process and may be written in Sisal as in Figure 11. This process may be partitioned advantageously for multiprocessing by dividing the streams into blocks allocated to each of several processors. This corresponds to slicing the diagram vertically and allocating each slice to a separate processing element.

9 Conclusions

The examples presented have shown how signal processing operations may be expressed elegantly using the stream data types of the Sisal functional programming language. A `stream_tail` operation that truncates the head of a stream by a specified number of elements would be a useful addition to the language. A transformation into dataflow graphs was given from which efficient implementations of compositions of stream functions may be derived. Dataflow computers and multithreaded processors capable of efficient fine-grain scheduling of threads would be attactive targets for this approach to high performance signal processing[8, 9].

The work reported here applies the results of research conducted by the Computation Structures Group of the MIT Laboratory for Computer Science to practical signal processing algorithms. The algorithms are taken from a real surveillance task, but simplified to permit easier presentation

```
type ImageStream = array [ stream [integer] ];
type MarkStream = array [ stream [boolean] ];

function PeakDetect (
   D: ImageStream, w: integer
   returns MarkStream )

   let
      Pk := for i in 3, w-2
         P := D[i, 2]
         C :=  for g in -1, +1 cross h in -1, +1
         return value of product
            if (g = 0 & h = 0) then true
            else ( D[g+i, h+2] <= P )
            endif
         end for
         S :=  for g in -1, +1 cross h in -1, +1
         return value of sum
            if (g = 0 & h = 0) then 0
            else D[g+i, h+2]
            endif
         end for
         return array of C & ( 8 * P > S + 8 * Th )
      end for
      Dt := for i in 2, w-1
      return array of
         stream_rest ( D[i] )
      end for
      Pr := PeakDetect ( Dt, w );
   in
      for i in 3, w-2
      return array of
         stream boolean [ Pk[i] ] || Pr[i]
      end for
   end let

end function
```

Figure 9: The peak detector function in Sisal.

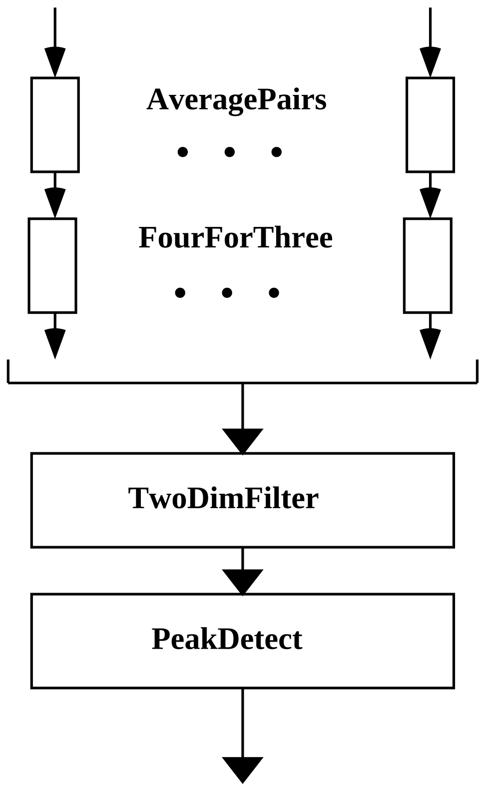

Figure 10: Diagram of the composition of stream functions.

in a brief paper. The complete original algorithms were expressed in a variant of the Val language[2] in a study performed by Dataflow Computer Corporation under contract to Boeing. The report of this work[4] included a suggested multithreaded processor design, manually derived machine code, and performance calculations for the Boeing application.

A Appendix: Proof of the Transformation

We will show that the recursive scheme and the dataflow scheme implement the same function mapping input data into result data.

We assume that the Compute and Auxiliary boxes are functions that

```
type Signal = stream [integer];
type ImageStream = array [ stream [integer] ];
type MarkStream = array [ stream [boolean] ];

function Process (
   D: ImageStream, w: integer
   returns MarkStream )

   let
      R := for i in 1, w
      return array of
         FourForThree ( AveragePairs ( D[i] ) )
      end for
   in
      PeakDetect ( TwoDimFilter ( R, w ) )
   end let

end function
```

Figure 11: Composition of stream functions in Sisal.

map vectors of scalars into vectors of scalars (Let the vector S have n elements.):

$$\text{Compute} : A^{m+n} \rightarrow A^{q}$$
$$\text{Auxiliary} : A^{m+n} \rightarrow A^{n}$$

The input and output streams are denoted by the (possibly infinite) sequences:

$$\begin{aligned} D &= d_1, d_2, \ldots \\ R &= r_1, r_2, \ldots \end{aligned}$$

First we present the relationships among values imposed by each of the two schemes; a superscript r refers to the recursion scheme and a superscript d refers to the dataflow scheme. For the tail-recursion scheme, Figure 6, let the index $j \leq 0$ be the depth of recursion.

$$D_0^r = D$$

$$
\begin{aligned}
D^r_{j+1} &= \text{Tail}\,(p, D^r_j) \\[1ex]
X^r_j &= \text{Extract}\,(m, D^r_j) \\[1ex]
S^r_0 &= S \\
S^r_{j+1} &= \text{Auxiliary}\,(X^r_j, S^r_j) \\[1ex]
E^r_j &= \text{Compute}\,(X^r_j, S^r_j) \\[1ex]
R^r_j &= \text{Affix}\,(q, E^r_j, R^r_{j+1})
\end{aligned}
$$

For the dataflow scheme, Figure 7, j indexes the successive values (tuples or stream elements) passed over links of the graph.

$$
\begin{aligned}
D^d_0 &= D \\
D^d_{j+1} &= \text{Tail}\,(p, D^d_j) \\[1ex]
X^d_j &= \text{Extract}\,(m, D^d_j) \\[1ex]
S^d_j &= \textbf{if } j = 0 \textbf{ then } S \textbf{ else } \text{Auxiliary}\,(X^d_{j-1}, S^d_{j-1}) \\[1ex]
E^d_j &= \text{Compute}\,(X^d_j, S^d_j) \\[1ex]
R^d_j &= E^d_0 \,||\, E^d_1 \,||\, E^d_2 \,||\, E^d_3 \,||\, \ldots
\end{aligned}
$$

We show by induction that corresponding variables are equal for all $j \geq 0$.

Basis ($j = 0$):

$$
\begin{aligned}
X^r_0 &= \text{Extract}\,(m, D^r_0) \\
&= \text{Extract}\,(m, D) \\
&= \text{Extract}\,(m, D^d_0) \\
&= X^d_0
\end{aligned}
$$

$$
\begin{aligned}
S_0^r &= S = S_0^d \\
\\
E_0^r &= \text{Compute}\,(X_0^r, S_0^r) \\
&= \text{Compute}\,(X_0^d, S_0^d) \\
&= E_0^d
\end{aligned}
$$

Induction ($j > 0$):

$$
\begin{aligned}
D_j^r &= \text{Tail}\,(p, D_{j-1}^r) \\
&= \text{Tail}\,(p, D_{j-1}^d) \\
&= D_j^d \\
\\
X_j^r &= \text{Extract}\,(m, D_j^r) \\
&= \text{Extract}\,(m, D_j^d) \\
&= X_j^d \\
\\
S_j^r &= \text{Auxiliary}\,(r, X_{j-1}^r, S_{j-1}^r) \\
&= \text{Auxiliary}\,(r, X_{j-1}^d, S_{j-1}^d) \\
&= S_j^d \\
\\
E_j^r &= \text{Compute}\,(X_j^r, S_j^r) \\
&= \text{Compute}\,(X_j^d, S_j^d) \\
&= E_j^d
\end{aligned}
$$

It follows that

$$
\begin{aligned}
R^r &= \text{Affix}\,(q, E_0^r,\ \text{Affix}\,(q, E_1^r, \ldots)) \\
&= E_0^r \mid\mid E_1^r \mid\mid \ldots \\
&= E_0^d \mid\mid E_1^d \mid\mid \ldots \\
&= R^d
\end{aligned}
$$

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, MA, 1985.

[2] W. B. Ackerman and J. B. Dennis. VAL—A Value-Oriented Algorithmic Language. Technical Report 218, Laboratory for Computer Science, MIT, 1979.

[3] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall International Series in Computer Science, 1988.

[4] Dataflow Computer Corporation. Time Dependent Signal Processing Algorithms for Optical Surveillance. Final Report for Contract HA4176 to Boeing Aerospace, Dataflow Computer Corporation, Belmont, MA 02178, November 1988.

[5] J. B. Dennis. An operational semantics for a language with early completion data structures. In *Formalization of Programming Concepts*, volume 107 of *Lecture Notes in Computer Science*, pages 260–267. Springler-Verlag, 1981.

[6] J. B. Dennis and K. S. Weng. An abstract implementation for concurrent computations with streams. In *Proceedings of the 1979 International Conference on Parallel Processing*, pages 35–45, 1979.

[7] Jack B. Dennis. First version of a data-flow procedure language. In *Proceedings of the Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 362–376. Springler-Verlag, 1975.

[8] Jack B. Dennis. The evolution of 'static' data-flow architecture. In J.-L. Gaudiot and L. Bic, editors, *Advanced Topics in Data-Flow Computing*, chapter 2. Prentice-Hall, 1991.

[9] Jack B. Dennis and Guang R. Gao. Multithreaded architectures: principles, projects, and issues. In Robert A. Ianucci, editor, *Advances in Multithreaded Computer Archtecture.* Kluwer, 1994.

[10] David. P. Friedman and Daniel S. Wise. CONS should not evaluate its arguments. In S. Michaelson and R. Milner, editors, *Automata, Languages and Programming: Third International Colloquium*, pages 257–284. Edinburgh University Press, 1976.

[11] G. R. Gao, R. Govindarajan, and P. Panangaden. Well-behaved programs for DSP computation. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, San Francisco*, pages V–561–564, March 1992.

[12] W. H. Ho, Edward A. Lee, and D. G. Messerschmitt. High level data flow programming for signal processing. In R. W. Broderson and H. S. Muscovitz, editors, *VLSI Signal Processing, III*, pages 385–395, New York, 1988. IEEE Press.

[13] John Kelly, C. Lochbaum, and Victor Vyssotsky. A block diagram compiler. *Bell System Technical Journal*, 40(3), May 1961.

[14] Peter J. Landin. A correspondence between Algol 60 and Church's lambda notation: Part I. *Communications of the ACM*, 8(2):89–101, February 1965.

[15] Edward A. Lee. Consistency in dataflow graphs. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):223–235, April 1991.

[16] J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, and R. Thomas. SISAL: Streams and Iteration in a Single Assignment Language: Reference Manual Version 1.2. Technical Report M-146, Rev. 1, Lawrence Livermore National Laboratory, 1985.

[17] R. S. Nikhil and Arvind. Id: A Language with Implicit Parallelism. Computation Structures Group Memo 305, Laboratory for Computer Science, MIT, 1990.

[18] R. R. Oldehoeft, A. P. W. Bohm, D. C. Cann, and John T. Feo. SISAL Reference Manual: Language Version 2.0. Technical report, Lawrence Livermore National Laboratory and Colorado State University, 1992.

[19] K. S. Weng. Stream-Oriented Computation in Recursive Data Flow Schemes. Technical Report MAC/TM–68, MIT Laboratory for Computer Science, October 1975.