AUTOMATIC PROGRAMMING FOR STREAMS

David Barstow
SchlumbergerDoll Research
Old Quarry Road
Ridgefield, Conn 06877-4108

Abstract

Most automatic programming research has focused on programs which terminate and which produce output values upon termination. By contrast, programs which operate on streams of data usually do not terminate and usually produce streams of output data during execution. Such stream programs may be specified with a technique which is a generalization of specification techniques for conventional programs The use of transformations also seems to be an appropriate technique for automatically synthesizing stream programs.

I. Introduction

Previous work on automatic programming (e.g., [Biermann et al 84]) has been concerned with sequential programs whose inputs are available before the program executes and whose outputs need not be produced until the program terminates However, many computations must deal with data which is not available until after the program has started and must produce data before it terminates. In fact, in many cases the program must not terminate; rather it must continually execute, responding to input data as they become available and producing output data whenever appropriate. Such computations are usually modeled as communicating concurrent processes. One form of communication among such processes involves streams of data. In this paper, the automatic programming problem for stream programs will be defined and an approach to solving the problem will be described.

II. The Problem

A. Streams

There are several ways to model computation with streams, ranging from pure dataflow [Ackerman 82] to coarse-grained dataflow [Kahn and McQueen 77] to pipes [Ritchie and Thompson 74] In our work, we use a model called the Stream Machine, which is essentially a coarse-grained dataflow model with extensions for real-time computations [Barth, Guthery, and Barstow 85].

For the sake of clarity, a somewhat simplified formulation will be used in this paper:

A *stream* is a sequence of data values:

$$s = s[1], s[2], \ldots, s[k_s]$$

The length of s. k_s, is initially 0 (i.e.. there are no data values) and increases as new data values are added to the stream.

Produce(s,x) adds x to s as a new data value; that is. k_s is incremented by one and s[K_s] is set to x Only one process may add data values to any given stream.

Consume(s) reads a data value from stream s Specifically, it returns the data value whose index is the lowest of those values not yet read by the consuming process. If there is no such value (i e , not enough data values have been produced yet), the process suspends execution and resumes after another data value has been produced for the stream Different consuming processes may consume the elements at different rates without interfering with each other Conceptually, one may think of each consumer operating on a different copy of the stream.

Note that no process may alter a stream other than by adding at one end and removing at the other, nor may a process determine the length of the stream. Because of these characteristics, computations modeled as processes communicating through streams are equivalent to conventional dataflow, and thus are deterministic.

B. Stream Programs

A stream program consists of a set of concurrently executing processes communicating via streams. Each of the processes is defined by a program written in a traditional sequential language, extended by the two stream operations defined earlier *Produce(s,x)* and *Consume(s).* Perhaps the most interesting aspect of stream programs is that, in general, they may not terminate. In fact, a typical stream process is a non-terminating loop which consumes from some streams and produces on others during each execution of the loop body.

C. Specifying Stream Programs

A specification of a stream programming problem consists of terms and predicates. The terms are either static, refenng to single data values, or they are streams, refering to streams of data values*. All terms are typed; however, in the following discussion, the types will be left implicit and should be clear from context. The

"In the following discussion, stream terms will be denoted by SMALL CAPITALS, individual elements of a stream will be denoted by the index in brackets terms are partitioned into three sets, input terms, output terms, and intermediate terms. There are two sets of predicates-preconditions are predicates whose arguments are input terms; postconditions are predicates whose arguments are input, output, or intermediate terms.

Such a specification is similar to specification techniques for conventional sequential programs [Biermann et al 84] In fact, if there are no stream terms, we have a conventional style of specification. For such conventional specifications, a program would be said to satisfy the specifications if, for all values of the input terms which satisfy the preconditions, the program terminates with values for the output terms which satisfy all of the postconditions. However, since there may be streams in the specification and since stream programs may not terminate, we must define the requirements for a target program somewhat differently. Informally, we would like the program to guarantee that all postconditions are satisfied by all stream elements which have been consumed or produced so far. Somewhat more formally, a program will be said to satisfy a specification if, for all sequences of initial values on input streams which satisfy the preconditions, the program eventually produces output streams whose initial values satisfy the postconditions. For example, consider the specification:

$$\mathrm{OUTPUT}[k] = \max\{\mathrm{INPUT}[i] \mid 1 \leq i \leq k\}$$

where INPUT and OUTPUT are input and output streams respectively. A program satisfies this specification if, for all sequences of initial values, INPUT[l], INPUT[2]. . . . .INPUT[H], the program eventually produces at least *n* elements of OUTPUT such that

$$\mathrm{OUTPUT}[k] = \max\{\mathrm{INPUT}[i] \mid 1 \leq i \leq k\} \text{ for } 1 \leq k \leq n$$

Thus, the automatic programming problem for streams is to transform a specification of the form given above into a set of programs, each described in a sequential language extended with stream constructs, that satisfies the specification

III. An Approach

A. Relations on Streams

The difficult part in specifying a stream program is to describe the pre- and postconditions. In our work, we initially tried to use a relatively general technique in which relations referenced stream values directly by their indices. We found, however, that subsequent reasoning about the specifications was quite difficult for many common cases (e.g merging two streams together to create a third) because the manipulations of stream indices was fairly complex. To simplify both specifications and subsequent reasoning we are using a technique based on operators and relations oriented toward streams, rather than toward stream elements. The particular operators which we have found useful are:

Generated)

F is a function of one integer argument. The resulting stream consists of successive values of *F* for the natural numbers. That is:

$$S = Generate(I) \text{ iff } S[i] = F(i)$$

StreamMap$(F, S_1, S_2, \ldots, S_k)$

The s, are all streams; F is a function of *k* arguments defined on the types of the elements of the s streams. The resulting stream contains the value of *F* applied to successive elements of the s streams

$$S = StreamMap(F, S_1, S_2, \ldots, S_k) \text{ iff}$$
$$S[i] = F(S_1[i], S_2[i], \ldots, S_k[i])$$

hilter{s,P)

s and P are streams. The elements of s are of any type; the elements of P are Booleans. The elements of the resulting filtered stream are those elements of s for which the corresponding element of P IS true. More formally:

$$F = Filter(S,P) \text{ iff } F[i] = S[j]$$

where is the smallest integer such that *True* occurs / times in $\{P[1], P[2], \ldots P[j]\}$. Note that this implies that $P[j] =$ *True.*

$Merge(C, S_1, S_2, \ldots, S_k)$

The elements of the s streams are all of the same type; the elements of c are integers in the range *[1, .k].* The resulting stream contains all of the elements of the s, streams, merged according to the elements of c:

$$M = Merge(C, S_1, S_2, \ldots, S_k) \text{ iff } M[i] = S_{C[i]}[j])$$

where / is the number of occurrences of c[/] in $\{C[1], C[2], \ldots, C[i]\}$.

Shift(s,k)

S is a stream; *k* is an integer. The elements of the resulting stream are the same as the elements of s, shifted by *k* indices. More formally.

$$S' = Shift(S,k) \text{ iff } S'[i] = S[i+k]$$

Pack/ng(S,k)

S is a stream; *k* is an integer. The elements of the resulting stream are vectors' of length *k* whose elements are of the same type as the elements of s. The vectors in the resulting stream correspond to continuous subsequences of the elements of s

$$P = Packing(S,k) \text{ iff}$$
$$P[i] = \langle S[k*(i-1)+1], S[k*(i-1)+2], \ldots, S[k*i] \rangle$$

Window[sM)

s is a stream; *k* is an integer. The elements of the resulting stream are vectors of length *k* whose elements are of the same type as the elements of s The vectors in the resulting stream correspond to a moving window over s:

$$W = Window(S,k) \text{ iff}$$
$$W[i] = \langle S[i], S[i+1], \ldots, S[i+k-1] \rangle$$

In addition, we require one type of stream predicate which cannot be expressed as a simple stream operator:

$StreamRelation(R, S_1, S_2, \ldots, S_k)$

The s are all streams; R is a relation of *k* arguments

'Angle brackets will be used to denote the construction of a vector from a sequence of values, individual vector elements will be denoted by enclosing the index in angle brackets defined on the types of the elements of the s streams. The s streams satisfy the *StreamRelation* expression if successive elements of the s_i streams satisfy *Fi-*

StreamRelation(R,s_1,s_2,...,s_k) iff
R(s_1[i],s_2[i],...,s_k[i])

bs a simple example of a specification, consider a switching problem: the elements of INPUT are to be split off into one of two other streams, OUTPUT_1 or OUTPUT_2, depending on the value of CONTROL, which is a stream whose elements are either 1 or *2.* Informally, INPUT is a merge of OUTPUT_1 and OUTPUT_2 More formally:

We do not claim that this particular set of operators is complete or in any sense primitive. Rather they seem to cover well the software tasks which we have been studying. As our work continues, we expect the set of operators to grow and evolve

Input Terms, INPUT, CONTROL

Output Terms, OUTPUT_1 OUTPUT_2

Postconditions

INPUT = Merge(CONTROL,OUTPUT_1 ,OUTPUT_2)

As a second example, consider the specification of a simple feedback loop which adjusts the gain on an amplifier to keep the amplitude of a signal close to *1*:

Input Terms: AMPLITUDE

Output Terms' GAIN

Postconditions.

```
GAIN[1] = 0
Shift(GAIN,1) = StreamMap(NewGain,GAIN,AMPLITUDE)
  NewGain(x,y) <=> x       if .9 ≤ y ≤ 1 1
                   x + 1   if y < .9
                   x + 1   if 1.1 < y
```

B. Target Language

As stated earlier, a stream program consists of a set of concurrently executing sequential programs. The sequential programs themselves are written in a traditional sequential programming language, extended to include stream operations. The details of the sequential language are not particularly important. In this paper, we will consider the basic primitives to be the two stream operations, Produce(s,x) and *Consume(s),* and assignment to a local variable. In addition, we will consider as primitive any problem specification which does not involve streams. We will assume that such programming problems can be handled adequately by some other technique (e.g., algebraic manipulation [Barstow et al 82]), leaving us free to focus in this paper on automatic programming techniques for streams

In addition to these primitive operations, we will include in our target language the following three control structures:

```
sequence      {program; ... program}
conditional   case expression of
                {v1: program;
                 v2: program;
                 ...
                 [illegible]
repetition    while expression do program
```

C. Transformation Rules for Stream Programs

Our approach to automatic programming for stream programs involves the use of transformations: we represent knowledge about programming with streams as transformations which replace one part of a partially developed program by another To date, we have identified three general types of transformations: *algorithm instantiation* transformations produce sequential algorithms for stream problems; *problem reduction* transformations split a single stream problem into several, presumably simpler, stream problems; *stream elimination* transformations remove unnecessary streams by collapsing several sequential processes into a single one.

C.1. Algorithm Instantiation Transformations

The algorithm instantiation transformations have two parts:

patterns consisting of particular types of relations on streams

replacements consisting of particular sequential algorithms

In general, the patterns may involve several relations. To date, however, we have been working only with transformations whose patterns involve a single relation, such as the following:

```
S = Generate(F)
      Output: S
->    {i := 1;
       while True do
             {Produce(S,F(i));
              i := i+1}}

F = Filter(S,P)
      Input: P, S
      Output: F
->    while True do
            {x := Consume(S);
             case Consume(P) of
                  {True: Produce(F,x)}}

M = Merge(C,S1,...,Sk)
      Input: C, S1, ..., Sk
      Output: M
->    while True do
            {case Consume(C) of
                  {1: m := Consume(S1);
                   ...
                   k: m := Consume(Sk)};
             Produce(M,m)}

M = Merge(C,S1,...,Sk)
      Input: C, M
      Output: S1, ..., Sk
->    while True do
            {m := Consume(M);
             case Consume(C) of
                  {1: Produce(S1,m);
                   ...
                   k: Produce(Sk,m)}}
```

```
S'[1] = $x_1$
...
S'[k] = $x_k$
S' = Shift(S,k)
        Input: S
        Output: S'
->      {Produce(S',$x_1$);
         ...
         Produce(S',$x_k$);
         while True do
                Produce(S',Consume(S))}

P = Packing(S,k)
        Input: P,k
        Output: S
->      while True do
                {p := Consume(P);
                 i := 1;
                 while i<=k do
                        {Produce(S,p<i>);
                         i := i+1}}

P = Packing(S,k)
        Input: S,k
        Output: P
->      while True do
                {i := 1;
                 while i<=k do
                        {p<i> := Consume(S);
                         i := i+1};
                 Produce(P,p)}

W = Window(S,k)
        Input: S,k
        Output: W
->      {i := 1;
         while i<k do
                {w<i> := Consume(S);
                 i := i+1};
         while True do
                {w<k> := Consume(S);
                 Produce(W,w);
                 i := 1;
                 while i<k do
                        {w<i> := w<i+1>;
                         i := i+1}}}

S = StreamMap(F,$S_1$,...,$S_k$)
        Input: $S_1$, ... $S_k$
        Output: S
->      while True do
                {$x_1$ := Consume($S_1$);
                 ...
                 $x_k$ := Consume($S_k$);
                 Produce(S,F($x_1$,...,$x_k$))}

StreamRelation(R,$S_1$,...,$S_k$)
        Input: $S_{i_1}$, ... $S_{i_m}$
        Output: $S_{j_1}$, ... $S_{j_n}$
->      while True do
                {$x_{i_1}$ := Consume($S_{i_1}$);
                 ...
                 $x_{i_m}$ := Consume($S_{i_m}$);
                 R($x_1$, ..,$x_k$)*
                 Produce($S_{j_1}$,$x_{j_1}$);
                 ...
                 Produce($S_{j_n}$,$x_{j_n}$)}
```

C.2. Problem Reduction Transformations

Problem reduction transformations are intended to reduce complex problems to problems which are simple enough to be handled by the algorithm instantiation transformations. Since subproblems handled by the algorithm instantiation transformations correspond to separate processes in the target program, the effect of a problem reduction transformation is to introduce additional processes into the final program. Two examples, stated informally, are

> A postcondition relation may be separated into a separate problem specification if it involves only input streams and at most one intermediate or output stream
>
> A postcondition relation may be separated into a separate problem specification if none of the remaining postcondition relations involve any of the first relation's intermediate or output streams.

C.3. Stream Elimination Transformations

While streams are convenient conceptual tools which can contribute to simplicity and modularity in programming, there may be a computational cost associated with their use. For this reason, we are developing stream elimination rules.** Two such rules, stated informally, are the following:

> If two processes consume a stream exactly once during each iteration of the loop in the process body, the loop bodies of the two processes may be combined.
>
> If a process produces a single output stream which is consumed by only one other process, and the second process consumes the stream exactly once in the body of the loop, then the body of the loop of the second process may be merged with the body of the loop of the first process at the point at which the stream is produced by the first process.

IV. Example

In this section, we will consider an example drawn from a program

•Note that *R*(x_1 ..x_k.) is simply another program specification whose input terms are x_{i_1}, . .x_{i_m} and whose output terms are x_{j_1}.. .x_{j_n}

"Note, however, that the use of such rules does not necessarily produce more efficient code, since the efficiency of streams depends on the architecture of the target machine For example, reducing the number of streams and processes may prevent taking advantage of parallelism on a multiprocessor architecture designed to control a remote physical device communicating with a computer through streams of signals and commands. The primary job of the computer is to command the device to alternate between two measurements. Associated with each measurement is an A/D converter whose gain must be set with each command. The gain commands for the measurements are received on separate streams but must be sent to the device on a single output stream.

A. Specification

The specification of the stream program for this example is as follows:

Initial Problem

Input Terms: $GAIN_1$,$GAIN_2$

Output Terms: COMMAND,GAIN

Intermediates: CONTROL

Preconditions:

Postconditions:

CONTROL = *Generated)*

F(i) = (i+ 1)mod 2)+ 1

GAIN = Merge(CONTROL,$GAIN_1$,$GAIN_2$)

StreamRe/ation(MeaSL/remenr,CONTROL,COMMAND)

Measurement(x,y) < = > x = 1 and y = "$Measurement_1$"
or x = 2 and y = "$Measurement_2$"

B. Synthesis

B.1. Problem Reduction

This specification can be reduced to three simple subproblems by applying each of the two problem reduction transformations shown earlier. The specifications of the resulting subproblems are:

Subproblem 1

Input Terms:

Output Terms: CONTROL

Preconditions:

Postconditions:

CONTROL = *Generate(F)*

F(i) = ((i + 1) mod 2) + 1

Subproblem 2

Input Terms: $GAIN_1$.$GAIN_2$CONTROL

Output Terms: GAIN

Preconditions:

Postconditions:

GAIN = *Merge*(CONTROL,$GAIN_1$,$GAIN_2$)

Subproblem 3

Input Terms: CONTROL

Output Terms: COMMAND

Preconditions:

Postconditions:

StreamRe/ation(Measurementf,cONTROL,cOMMANO)

Measurement(x,y) < = > x = 1 and y = "$Measurement_1$"
or x = 2 and y = "$Measurement_2$"

B.2. Algorithm Instantiation

The first two subproblems may each be transformed into algorithms by applying one of the transformations shown earlier. The resulting algorithms are:

Subprogram 1

```
{i := 1;
 while True do
       {Produce(CONTROL,((i+1) mod 2)+1);
        i := i+1}}
```

Subprogram 2

```
while True do
      {case Consume(CONTROL) of
            {1: m := Consume(GAIN_1);
             2: m := Consume(GAIN_2)};
       Produce(GAIN,m)}
```

The third subproblem may also be transformed by an algorithm instantiation transformation, but the relation *Measurement* remains as a subproblem.

Subprogram 3

```
while True do
      {x := Consume(CONTROL);
       Measurement(x,y);
       Produce(COMMAND,y)}
```

The relation *Measurement* does not involve any streams and may be replaced by its definition, solved for *y:*

Subprogram 3

```
while True do
      {x := Consume(CONTROL);
       case x of
            {1: y := "Measurement_1";
             2: y := "Measurement_2"};
       Produce(COMMAND,y)}
```

B.3. Stream Elimination

The second and third subprograms may be combined by applying the first stream elimination transformation given earlier. The result is:

Combined Subprograms: 2, 3

```
while True do
      {x := Consume(CONTROL);
       case x of
            {1: m := Consume(GAIN_1);
             2: m := Consume(GAIN_2)};
       Produce(GAIN,m);
       case x of
            {1: y := "Measurement_1";
             2: y := "Measurement_2"};
       Produce(COMMAND,y)}
```

Note that, in this case, the *StreamRelation* postcondition could have been expressed as a *StreamMap:*

COMMAND = *StreamMap(MeasurementFunction,*CONTROL)
MeasurementFunction(x) = "$Measurement_1$" if x = 1
= "$Measurement_2$" if x = 2

The relation form was chosen for the sake of illustration.

This may be combined with the first process by applying the second stream elimination transformation given earlier, yielding:

Combined Subprograms: 1,2,3

```
{i := 1;
 while True do
       {x := ((i+1) mod 2)+1,
        case x of
             {1: m := Consume(GAIN$_1$);
              2: m := Consume(GAIN$_2$)};
        Produce(GAIN,m);
        case x of
             {1: y := "Measurement$_1$";
              2: y := "Measurement$_2$"};
        Produce(COMMAND,y);
        i := i+1}
```

B.4. Simplification and Optimization

Several simplifications and optimizations not related to streams may now be applied, resulting in the final code for the original specification:

Final Program

```
{i := 1;
 while True do
       {case i of
             {1: {m := Consume(GAIN$_1$);
                  y := "Measurement$_1$"};
              0: {m := Consume(GAIN$_2$);
                  y := "Measurement$_2$"}};
        Produce(COMMAND,y);
        Produce(GAIN,m);
        i := (i+1) mod 2}
```

V. Discussion

To date we have tested these techniques by hand simulation on a variety of specifications for software to control and record data from Schlumberger's well logging tools. The tasks to be performed by the software include encoding and decoding of signals, multiplexing and demultiplexing of communication channels, feedback loops, and simple calculations. We have also done partial simulations for a few log interpretation programs. In both situations, the specification and implementation techniques seem to work well. As we implement the techniques, we expect to make changes and extensions at a detailed level, but we also expect the overall approach to remain essentially the same.

It should be noted that these techniques address only one aspect of the problem of programming with streams. In a previous study, we characterized the programming process for log interpretation software in terms of four activities: informal problem solving, formal manipulation, implementation selection, and target language translation [Barstow 84]. The techniques described in this paper only address issues which arise during formal manipulation. Work is also underway on the other activities, such as informal problem solving in which streams are used to approximate continuous functions.

Acknowledgements

I am grateful to many colleagues for their criticisms and insights, including Paul Barth, Paul Dietz, Rick Dinitz, Sol Greenspan, Elaine Kant, David Steier, Stephen Smoliar, and Peter Will.

References

[Ackerman 82] W. Ackerman.
Dataflow Languages.
Computer 15(2): 15-23, February, 1982.

[Barstow 84] D. Barstow.
A perspective on automatic programming
A I Magazine 5(1):5-27. Spring, 1984.

[Barstow et al 82] D. Barstow, R. Duffey. S Smoliar, S. Vestal.
An automatic programming system to support an experimental science.
In *Sixth International Conference on Software Engineering.* Tokyo, Japan, September, 1982.

[Barth, Guthery, and Barstow 85]
P. Barth, S. Guthery, D. Barstow.
The Stream Machine.
In *Eighth International Conference on Software Engineering.* London, England, August, 1985.

[Biermann et al 84]
A. Biermann, G. Guiho, Y. Kodratoff (editors).
Automatic Program Construction Techniques.
Macmillan, 1984.

[Kahn and McQueen 77]
G. Kahn, D. MacQueen.
Coroutines and networks of parallel processes.
In *Information Processing 77.* International Federation of Information Processing Societies, 1977.

[Ritchie and Thompson 74]
D. Ritchie, K. Thompson.
The UNIX Time-Sharing System.
Communications of the ACM 17(7):365-275, July, 1974.