W. H. Burge

Stream Processing Functions

Abstract: One principle of structured programming is that a program should be separated into meaningful independent subprograms, which are then combined so that the relation of the parts to the whole can be clearly established. This paper describes several alternative ways to compose programs. The main method used is to permit the programmer to denote by an expression the sequence of values taken on by a variable. The sequence is represented by a function called a *stream*, which is a functional analog of a coroutine. The conventional **while** and **for** loops of structured programming may be composed by a technique of stream processing (analogous to list processing), which results in more structured programs than the originals. This technique makes it possible to structure a program in a natural way into its logically separate parts, which can then be considered independently.

Introduction

One of the underlying principles of structured programming [1] is that the separation of the parts of a program, and the relation of the parts to the whole, should both be clearly apparent from its written form. A second principle is that the meaning of each part should depend in a simple way only on the meaning of its subparts, and not on any other properties. Programs written in this way are easy to understand and write, and the details of their operation are transparently clear. This principle of structured programming is epitomized in the expression format of programming languages. The value of an expression depends only on the value of its subexpressions, and it depends on them by the simple notion of the application of a function to its argument. A more general type of expression than that commonly provided in programming languages is found in the notation of the lambda the arguments and results of a function can be a function. The programming language used in this paper is ISWIM, a modified version of the notation of the lambda calculus due to Landin [2] that is more palatable for programming purposes.

Another theme that runs through the writing on structured programming is the notion of *abstraction*. In its technical sense *abstraction* is an operation for producing an expression describing a function from an expression by indicating which identifiers are its variables. Thus $\lambda x.x^2 + x$ denotes a function that when presented with a number adds it to its square.

Another meaning given to *abstraction* is related to the levels of detail of a program. The program is first conceived in an abstract way and then elaborated by stepwise refinement to produce the final version. This idea of abstraction may also be applied to the data structures used in a program. An attempt should be made to separate the decisions about the logical structure of the information from decisions about how the structures are to be physically represented. It is valuable to attempt this separation because the shape of the structures needed depends on the problem being solved, whereas the choice of physical format depends more on the operations and data structures provided by the computer or programming system being used. In practice the distinction between logical and physical structure is difficult to make, and decisions about logical structure are usually made by default. The physical representation is often chosen at too early a stage, and wholesale changes of representation become impossible without rewriting large parts of the program. The effort of retaining the distinction seems worthwhile, however, because a clear picture of a set of information structures implies to some extent the shape of the programs that operate on, or create members of, the set. In some cases the underlying data structure may be hidden and take the form of arithmetical calculations performed by the instructions in the program. It seems preferable, especially in the early stages, to make this structure explicit in order to make the resulting programs easier to understand.

This paper describes a technique for constructing programs in which a general-purpose program that is a mere skeleton is first written and is fleshed out later by supplying arguments that specify the actions to be taken within the skeleton. The skeletal program can be considered to specify the whole family of programs obtained in this way. The general-purpose skeleton embodies the method of scanning the structure, and the arguments supplied specify the actions to be taken during the scanning. The

general-purpose programs can be produced in a mechanical fashion from the description of the set of data structures (sometimes called the abstract syntax of the set) being scanned.

Some examples of this method of constructing programs that operate on lists are given first. The technique is then applied to streams. A *stream* is a functional analog of a coroutine [3, 4] and may be considered to be a particular method of representing a list in which the creation of each list element is delayed until it is actually needed. Many examples of stream-processing programs are given in which list processing techniques are seen to correspond to the more conventional programming techniques using **while** and **for** loops.

It is often easier to understand and program a sequence of passes that happen one after the other than to consider an involved process in which different program elements are mixed together. This paper presents a method for getting the best of both worlds in the sense that the program is written as if it were a multipass program, but when it is executed the parts of the separate passes are interleaved. This principle seems to apply to programs that would not usually be considered natural for coroutine treatment. It is often easier to consider the sequence of values taken on by a variable in a program as an object that can be manipulated, rather than considering the mechanisms that use, test, and change the variable. The coroutine technique has the advantage that the pieces of program that contain the variable are gathered together in one place, rather than being scattered throughout the program. The stream technique makes it possible to structure a program in a natural way into its logical separate parts, which can be considered independently.

Describing sets of data structures

A set of structures can be introduced by both naming its components and by specifying the type of each component. If there are alternative formats for the same set, then a predicate is provided for each set and given a name. For example, the definition of an expression having infixed operators can be written:

An expression is
    either *atomic* and is an identifier
    or is *compound*
        and has an *operator*, which is an infixed operator,
        and a *right* and *left*, both expressions.

An expression defined in this way depends on the set's identifier and infixed operator. It is also assumed that functions for constructing tree-like data structures are introduced. In this case an operator is introduced to construct a compound expression from two expressions and an infixed operator. Given this description of a set, a general-purpose function can be constructed for processing its members by introducing a conditional expression to distinguish and process the alternative formats, by introducing a function applicable to each component of a Cartesian product, and another function to combine the results. The general shape of a program for operating on the expression defined above is:

$fx =$

    if *atomic* x

    then $f_1 x$

    else $f_2(f_3(operator\ x))(f_4(left\ x))(f_5(right\ x))$

in which $f_1, f_2, \cdots f_5$ are parameterized functions that specify the action to be carried out. If the subexpressions are to be treated in the same way as the whole expression, then the following function results.

$exp\ f\ g\ h\ x =$

        if *atomic* x

        then $f\ x$

    else $g(h(operator\ x))(exp\ f\ g\ h(left\ x))$

        $(exp\ f\ g\ h(right\ x))$.

The value of an expression, for example, could be obtained by applying the function

$(exp\ v\ g\ v)$ **where** $g\ x\ y\ z = x(y, z)$

in which v is a function for finding the value of an identifier or operator.

Lists

Lists, in one guise or another, are the most commonly used data structure in programming. The definition of a list of elements of type A is:

An *A-list* is

    either *null*

    or has a *head*(h), which is an A, and

        a *tail*(t), which is an *A-list*.

Associated with an object x there is a function for prefixing it to a list, which is called *prefix x*. The expression *x:y* is used as an abbreviation for *prefix x y*, () denotes the null list, *a, b, c, d, e* is used instead of *a:(b:(c:(d:(e:*()))))), and *u x* is used to denote a list with one element. The types of the list functions introduced are

null ε *A-list* → *truth value*

h ε *nonnull A-list* → *A*

t ε *nonnull A-list* → *A-list*

$prefix \in (A \rightarrow (A\text{-}list \rightarrow A\text{-}list))$

$(\ ) \in A\text{-}list$

These functions are closely interrelated as follows:

$$null(\ ) = true$$

$$null(x{:}y) = false$$

$$h(x{:}y) = x$$

$$t(x{:}y) = y$$

$$(h\ z){:}(t\ z) = z$$

in which x is an A, y is an $A\text{-}list$, and z is a nonnull $A\text{-}list$. Each structure description is assumed to create functions that conform to a number of axioms of this type. Many functions that operate on lists have the same basic structure. For example

$sum\ x =$

if $null\ x$

then 0

else $(h\ x) + sum(t\ x)$

e.g., $sum(1, 2, 3, 4) = 10$

$product\ x =$

if $null\ x$

then 1

else $(h\ x) \times product(t\ x)$

e.g., $product(1, 2, 3, 4) = 24$

$append\ x\ y =$

if $null\ x$

then y

else $(h\ x){:}(append(t\ x)y)$

e.g., $append(1, 2)(3, 4, 5) = 1, 2, 3, 4, 5$

$concat\ x =$

if $null\ x$

then $(\ )$

else $append(h\ x)(concat(t\ x))$

e.g., $concat((1, 2), (3, 4), ()) = 1, 2, 3, 4$

$map\ f\ x =$

if $null\ x$

then $(\ )$

else $(f(h\ x)){:}(map\ f(t\ x))$

e.g., $map\ square\ (1, 2, 3, 4) = 1, 4, 9, 16$.

It can be seen that these functions only differ in the first arm of the conditional expression and in the function that combines whatever is produced from the head of the list (usually the head itself) with whatever is produced by applying the same function to the tail of the list. The common parts of these functions may be expressed as a function called *list1*, defined below, in which the parts that are not common have been made variables.

$list1\ a\ g\ f\ x =$

if $null\ x$

then a

else $g(f(h\ x))(list1\ a\ g\ f\ (t\ x))$

The result of applying the function $(list1\ a\ g\ f)$ to a list is a if the list is empty; otherwise it is the result of applying g to two arguments, 1) the result of applying f to the head of the list and 2) the result of applying the same function to the tail of the list. It is now possible to redefine the five functions in terms of *list1*. It can be seen that this technique both saves writing and is more likely to produce a correct program because the complex programming (i.e., the conditional expression and looping) has been written once and for all in the function *list1*. In the following definitions I is the identity function, $K\ x\ y = x$, and *postfix* adds a new item to the end of a list, so that

$$postfix\ x\ y = append\ y\ (u\ x)$$

The new definitions are:

$$sum = list1\ 0\ plus\ I$$

$$product = list1\ 1\ mult\ I$$

$$append\ x\ y = list1\ y\ prefix\ I\ x$$

$$concat = list\ 1\ (\ )\ append\ I$$

$$map\ f = list1\ (\ )\ prefix\ f$$

Some other examples are:

$$length = list1\ 0\ plus\ (K\ 1)$$

$$sumsquares = list1\ 0\ plus\ square$$

$$reverse = list1\ (\ )\ postfix\ I$$

$$identity = list1\ (\ )\ prefix\ I$$

The functions defined in terms of *list1* all scan the list (i.e., accumulate the result) from right to left. There is a second family of functions that scans lists from left to right, which can be defined by using a function called *list2*.

$list2\ a\ g\ f\ x =$

if $null\ x$

then a

else $list2(g(f(h\ x))a)g\ f\ (t\ x)$

The function *list2* may be implemented by the "iterative" program below; thus it may be more efficient to use *list2* than *list1*.

L: **if** *null x*

then *a*

else

$a := g(f(h\ x))a$

$x := t\ x$

go to *L*

A function that operates on two equal length lists and combines corresponding members by using a function f and then combines the results by using successive applications of a function g is defined below.

zip1 a g f x y =

if *null x*

then *a*

else $g(f(h\ x)(h\ y))(zip1\ a\ g\ f(t\ x)(t\ y))$

Examples of its use are to form the scalar product (*zip 1 0 plus mult*) or to produce a list of pairs from a pair of lists by

zipm = *zip1* ( ) *prefix pair*

where *pair x y* = *x*, *y*

Several other examples of this technique of matching the program structure to the data structure may be found in Burge [5], in which an analogy is shown between constructing functions in this way and constructing the enumerating generating functions of combinatorial theory.

Sequences, coroutines, and streams

When one function produces a list in its natural order and another processes the list items in the same order, it is often unnecessary to produce the whole list before applying the second function to it. The two functions can be combined so that at any stage the second function can issue a demand for the next list item, which is then provided by the first function. The creation of the next list item is thereby delayed until it is actually needed. It is often easier to write programs in two stages in which the list is an intermediate result of the computation. However, it is more economical of storage to use the combination of the two functions, in which only one member of the list appears as an intermediate result. This section contains an examination of methods of combining functions in this way. It is possible to have the best of both worlds by writing the program as if the whole list appeared as an intermediate result but having the actual implementation only create one member at a time. The function called upon to produce the next item must both produce it and reset itself to be prepared to deliver the remainder, or tail, of the list the next time it is called.

The data structure that is relevant is an *A-sequence*, defined as follows:

An *A-sequence* has a *hs*, which is an *A*,

and a *ts* which is an *A-sequence*.

Streams A sequence is therefore an infinite list, and the problem of conserving storage for its representation inside a computer becomes even more pressing. A sequence can be represented by a particular type of function, which is called a *stream function* or a *stream*. A *stream* is applicable to an empty list of arguments, and it produces a pair whose first is the next item in the sequence and whose second is a stream for the tail of the sequence. Thus

A-stream $\subseteq$ (*null list* $\rightarrow$ *A* $\times$ *A-stream*)

The head, tail, and prefix functions for a stream are defined as follows:

def *hs s* = *first*(*s*( ))

def *ts s* = *second*(*s*( ))

def *prefixs x s* = $\lambda(\ ).x, s$

The *hs* of a stream is the first member of the pair that results from applying the stream to the null list, and the *ts* is the second member. It follows that *s* is applied each time that either *hs* or *ts* is applied. It is often more economical to make sure that the stream is only applied once by using a construction such as

let $x, y = s(\ ); \cdots x \cdots y \cdots x \cdots x \cdots$

A stream can be constructed from its head x and its tail s by the function *prefixs x s* = λ ( ).*x*, *s*. When applied to the null list, this function produces the pair (x, s). The axioms that relate streams and their components are:

$hs(\lambda\ (\ ).(x, y)) = x$

$ts(\lambda\ (\ ).(x, y)) = y$

$prefixs(hs\ z)(ts\ z) = z$

Stream processing functions A number of examples of stream processing functions that are analogous to list processing functions are defined below.

Example 1 Given a transformer f and an initial value x, a stream function for the sequence $x, f\,x, f^2\,x, f^3\,x \cdots$ may be obtained by using

def rec *generate f x* ( ) = *x*, *generate f* (*f x*)

The first member of the sequence of the stream (*generate* $f\,x$) is x, and the remainder of the sequence is represented by the stream (*generate* f ($f\,x$)). Given zero and a successor function, the sequence of nonnegative integers can be represented by the stream *integer* = (*generate successor* 0) = 0, 1, 2, 3, $\cdots$.

Example 2 The stream representations of sequences can be treated as if they were lists. It is possible to transform streams to other streams, for instance, by using the function *maps* defined below

def rec *maps* $f\,s\,(\,) = f\,x$, *maps* $f\,y$ **where** $x, y = s(\,)$.

The function *maps* transforms a sequence

$x_1, x_2, x_3, \cdots$

into the sequence

$f\,x_1, f\,x_2, f\,x_3, \cdots$.

The function *maps* delays the production of the next member of s until the next member of (*maps* $f\,s$) is required; it then applies the function f to the first member of s to produce the first member of (*maps* $f\,s$). The stream for the sequence of squares of nonnegative integers, for instance, is (*maps square integer*) = 0, 1, 4, 9, $\cdots$.

Example 3 The function *thefirst*, which finds the first member of a sequence having the property p and produces it as a result, together with the remaining stream, is defined below

def rec *thefirst* $p\,s =$

let $x, y = s(\,)$

if $p\,x$

then x, y

else *thefirst* $p\,y$

Assuming that the predicate *nonspace* tests whether a character is a nonspace character, then the next nonspace character can be obtained from a character stream by applying (*thefirst nonspace*) to it and then selecting the first of the pair produced. As another example, the first integer whose square is greater than *1000* is the first member of the pair

thefirst p integer **where** $p\,x = x^2 > 1000$

Example 4 The function *filter*, defined below, operates on a stream and a predicate p and produces a stream for those members having the property p.

def rec *filter* $p\,s\,(\,) =$

let $x, y = s(\,)$

if $p\,x$

then x, *filter* $p\,y$

else *filter* $p\,y$

A stream of nonspace characters can then be obtained from a character stream by applying (*filter nonspace*) to it. As another example (*filter prime integer*) is the stream of prime numbers.

Example 5 Two streams can be processed to produce a third by a function that is analogous to *zip1*.

def rec *zips* $f\,x\,y = \lambda\,(\,).(f(hs\,x)(hs\,y))$,

$(zips\,f\,(ts\,x)(ts\,y))$

The stream of pairs is produced from two streams by (*zips pair*).

Example 6 Streams are most useful for implementing functions that process character streams from input. The function *while*, defined below, produces a list from the initial segment of a stream just as long as its members all have the property p.

def rec *while* $p\,s =$

let $x, y = s(\,)$

if $p\,x$

then let $u, v = while\ p\ y$

$x{:}u, v$

else $(\,), s$

A related function is *until* p = *while* (*not* · p). If the predicate *sameline* is *not* · (*equal newline*), where *newline* is the carriage return line feed character, then the function (*while sameline*) operates on a character stream and produces a pair whose *first* is the next line of input and whose *second* is the remaining stream. To be able to reapply the same function the newline character must be removed by using *remove* $(x, y) = x, ts\ y$.

Example 7 Any function that produces a pair whose second member is the same type as its argument can be made into a stream-producing function by applying a function called *next* to it, defined as follows:

def rec *next* $r\,s\,(\,) =$

let $x, y = r\,s$

x, *next* $r\,y$

The function *next* is applicable to any function of the type

$r \in A \rightarrow B \times A$,

and produces a function of the type

$A \rightarrow B$-*stream*.

It follows that

$next \varepsilon ((A \to B \times A) \to (A \to B\text{-}stream))$.

The function *filter* can be redefined in terms of *next* as follows:

def *filter p s* = *next*(*thefirst p*)*s*

The function

(*next*(*remove* · (*while sameline*))) **where** *remove*(*x*, *y*)
= *x*, *ts y*

converts a character stream containing *newline* characters into a line stream in which the lines are the character lists between adjacent *newline* characters.

Example 8 The inverse operation converts a list stream into a character stream. Suppose that *concats* is a function for "flattening" a line stream into a character stream or, more generally, transforming an (A-list)-stream into an A-stream.

```
def rec concats s =
        let x, y = s( )
        if null x
        then concats y
        else λ( ).(h x, concats λ( ).t x, y)
```

The inverse operation of putting back the *newline* characters and flattening the line stream into a character stream is then (*concats* · *maps* (*postfix newline*)). The operation of *concats* is similar to an input buffering process in which blocks of records are read from outside, but the program requires the individual records one at a time. The function *concats* therefore converts a block-reading routine into a next-record routine.

Representing lists by streams In order to be able to represent a list by a stream it is necessary to choose some object that cannot be a list item to serve as an indication in the stream for the end of the list. This is called *end*. In fact, since a stream is a function, it can always be applied, the stream corresponding to an infinite list of *end*'s will serve as the indication. The null stream can be defined by *nullists* = (*generate I end*). The predicate for the null stream is defined as *nulls x* = ((*hs x*) = *end*). These lists, which are represented as streams, can be treated as if they were lists. The correspondence between the list functions and stream functions is given in Table 1.

It follows that any function that operates on, or produces, lists can be immediately transformed into a function that operates on, or produces, streams. General-purpose stream functions can be defined by analogy with the *list1* or *list2* functions.

Table 1 List functions and corresponding stream functions.

lists	*streams*
null x	*nulls x* = ((*hs x*) = *end*)
h	*hs*
t	*ts*
( )	*nullists*
x:*y*	λ( ).*x*, *y*
prefix	*prefixs*
u	*us x* = *prefixs x nullists*

```
def rec stream1 a g f s =
        if nulls s
        then a
        else g(f(hs s))(stream1 a g
                                f(ts s))
def rec stream2 a g f s =
        if nulls s
        then a
        else stream2 (g(f(hs s))a) g
                                f(ts s)
```

The *stream1* function produces the whole list before operating on it; the *stream2* function produces items one at a time and operates on them as they are produced. The functions on lists can be carried over to streams. The stream versions of the functions *map*, *append*, and *concat* follow:

```
def rec mapl f s =
        if nulls s
        then nullists
        else λ( ).f(hs s), mapl f(ts s)
def rec appendl x y =
        if nulls x
        then y
        else λ( ).hs x, appendl(ts x)y
def rec concatl s =
        if nulls s
        then nullists
        else let x, y = s( )
```

if *null x*

then *concatl y*

else λ ( ).*h x*,

concatl(λ ( ).*t x*, *y*)

Note that in this representation the *appendl* function is more efficient than the *append* function, which has to scan through the first argument to create its result. Another way to represent a stream is by a list x for its initial segment and a stream y for the remainder. The list is similar to an input buffer. Such a stream can be constructed by another variation of *append*.

def rec *appendls x y* ( ) =

if *null x*

then *y*( )

else *h x*, *appendls*(*t x*)*y*

The *concatl* function operates on a stream of lists and produces a stream. There are eight variations of the concatenating function produced by changing the top level list, the second level list, or the resulting list to streams. The version in which all three are streams is called *concatss* and is defined below:

def rec *concatss s* =

if *nulls s*

then *nullists*

else let *x*, *y* = *s*( )

if *nulls x*

then *concatss y*

else λ ( ).*hs x*,

concatss(λ ( ).*ts x*, *y*)

Some care has to be taken when a stream is produced to make sure that its elements are not really a list in disguise, in other words, to make sure that the stream elements are not materialized too soon. The method of evaluation that has been assumed is that the operator and operand parts of an expression are evaluated, and then the value of the operator is applied to the value of the operand. It is also assumed that the body of a lambda expression, i.e., the M part of an expression $\lambda x.M$, is only evaluated when the function is applied. If the *appendl* function had been put in the lambda convertible form

appendl x y =

if *null x*

then *y*

else *prefixs*(*hs x*)(*appendl*(*ts x*)*y*)

then the inner expression *appendl*(*ts x*)*y* would be applied when the function *appendl* is applied to x and y. This would cause the elements of the stream x to be materialized, and prefixed, using *prefixs*, to the stream y. In the first version of *appendl*, on the other hand, the expression *appendl* (*ts x*)*y* is only evaluated when the stream *appendl x y* is applied to the null list. The two definitions of *appendl* are lambda convertible and therefore are equivalent. The assumed method of evaluation, however, causes the two functions to behave in different ways. In order to produce the most delayed version of a stream the construction λ ( ).*x*,*y* should be used instead of *prefixs x y*, and expressions containing *prefixs* should not be used.

Loop control Streams can be used to implement the sequence of values taken on by a variable in a **do, for,** or **while** loop in a programming language. The loop control can be separated from the loop by using streams. This means that the same loop control can be used with two different loops or that the same loop can be used with two different loop controls. The list of numbers from *0* to n, for example, have the stream *whiles*(≤ *n*)*integer*, where:

def rec *whiles p s* ( ) =

let *x*, *y* = *s*( )

if *p x*

then *x*, *whiles p y*

else *nullists*( )

Again there is a companion function *untils p* = *whiles*(*not* · *p*). The stream corresponding to the Algol 60 phrase *a* **step** *b* **until** *c* is

untils(λ*x*(*x* − *c*) × *sign*(*b*) > 0)(*generate*(*plus b*)*a*).

The stream for *1* **step** *1* **until** *n* is

def *til n* = *untils*(> *n*)(*generate*(+ *1*)*1*).

Yet another definition of the factorial function is given below:

factorial n = *stream2 1 mult 1* (*til n*).

This separation of the loop control from the loop permits nonnumerical streams to control the looping. The double loop introduced by a piece of program of the form

for *i*: = *1* **step** *1* **until** *n* **do**

for *j*: = *1* **step** *1* **until** *m* **do**

could be regarded as a loop controlled by a stream of pairs. The function *map*, which replaces each item in a list by its transform under a function, can be extended to apply to two lists. The function *map2*, defined below, pro-

duces a list of the results of applying f to every pair of items, one from one list and the second from the other.

```
def map2 f x y = let g z = map(f z) y
                         concat(map g x)
```

The result of applying (*map2 pair*) to the two lists (*1*, *2*, *3*) and (*4*, *5*) is the cross product

(*1*, *4*), (*1*, *5*), (*2*, *4*), (*2*, *5*), (*3*, *4*), (*3*, *5*).

There is a stream version of this function, produced by replacing *map* by *mapl* and *concat* by *concatss*, i.e.,

```
def map2l f x y = let g z = mapl(f z) y
                          concatss(mapl g x)
```

The double **for** loop control above can be regarded as the stream of pairs (*map2l pair* (*til n*) (*til m*)).

"Walking" through trees It is often useful to be able to scan a data structure by using a stream instead of first listing its elements and then scanning the list. A general technique for producing a list from a structure is to replace all atomic elements by 1-lists, to produce a list of lists from each nonatomic component, and then concatenate these lists. These functions for producing lists can be systematically changed so that each atomic element is converted into a 1-stream, each nonatomic component is converted into a stream of streams, and then this stream of streams is concatenated to produce a stream by *concatss*.

A binary tree, for example, can be defined as being either empty or having a root and a right and a left, which are both binary trees. One function for flattening the binary tree to a list is:

```
def rec flatn x =
        if empty x
        then ( )
        else concat(flatn(left x), u(root x),
                                   flatn(right x))
```

This produces a list of nodes of a tree. In order to "walk" through the nodes of the tree one step at a time, one needs to produce a stream rather than a list. The stream for the binary tree is produced by applying

```
def rec flatn x =
        if empty x
        then nullists
        else concatss(flatn(left x), us(root x),
                                     flatn(right x))
```

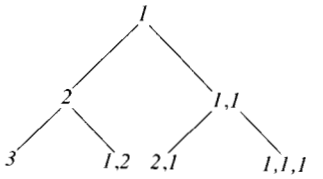

Figure 1 Top levels of infinite binary tree.

to the binary tree. Alternative scanning methods can be obtained by permuting the arguments of *concatss*.

Confluent streams It is clear that any tree-like data structure can be represented by a function such as a stream. An infinite binary tree, for example, can be represented by a function that, when applied to the null list, produces the root and two functions representing the left and right subtrees. A function for generating the trees in which the subtrees depend on the root can be defined by using:

```
def rec genbtree f g x ( ) = x, genbtree f g(f x),
                                genbtree f g (g x)
```

The binary tree that contains the compositions of n (i.e., all lists of positive integers whose sum is n) on level n is denoted by

```
genbtree f g(u 1) where f x = ((h x) + 1):(t x)
                      and g x = 1:x
```

This generates the infinite binary tree whose top is given in Fig. 1.

The empty binary tree can be defined as *emptys* = (*genbtree 1 1 end*) and the *empty* predicate defined as *empty f* = (*first*(*f*( ))) = *end*. The functions *roots*, *lefts*, and *rights* can be defined as the *first*, *second*, and *third* of the result of applying the binary tree to the null list. Functions analogous to those on streams can be constructed for trees. For example, the function *prune* defined below converts an infinite tree into a finite one and is analogous to *untils* for streams.

```
def rec prune p s ( ) =
        let x, y, z = s( )
        if p x
        then emptys( )
        else x, prune p y, prune p z
```

The binary tree for the compositions down to level n is obtained by applying (*prune*((*greater n*) · *sum*)).

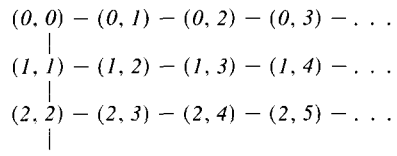

Figure 2 Top levels of a tree of pairs.

These infinite tree processing techniques can be used to solve a problem mentioned by Dijkstra [1] and attributed to Weizenbaum. Given an integer n, the problem is to write a program to find the smallest number that can be decomposed into the sum of two nth powers in at least two different nontrivial ways. The relevant data structure is an infinite tree of pairs, which starts as shown in Fig. 2. The structure is an (integer-pair) tree, where a tree is defined as follows:

An A-tree

has a *root*, which is an A,

and a *left*, which is an A-tree,

and a *right*, which is an A-sequence.

The required tree is

gen F G(0, 0)

where $F(x, y) = x + 1, y + 1$

and $G(x, y) = x, y + 1$

and rec *gen f g x* ( ) = *x*, *gen f g*(*f x*), *generate*

g(*g x*).

The infinite tree formed by mapping with $H(x, y) = x^n + y^n$ has the property that its root is smaller than any root in its subtrees. The resulting tree can therefore be sorted by taking the root as the first in the sorted sequence and then merging the two subtrees. The mapping function is

def rec *mapt f x* ( ) = **let** *root, left, right* = *x*( )

f root, mapt f left, maps f right

The tree can now be sorted using the function *sort*, which produces a stream of sorted numbers from the tree.

def rec *sort x* = **let** *root, left, right* = *x*( )

λ ( ). *root*, *sort*(*merge left right*)

where rec

merge x y =

let *a*, *b*, *c* = *x*( )

let *d*, *e* = *y*( )

if $a < d$

then λ ( ).*a*, (*merge b c*), *y*

else λ( ).*d*, *x*, *e*

The first repeated member of this stream has to be found next, using

def rec *repeat s* = **let** *x*, *y* = *s*( )

let *u*, *v* = *y*( )

if $x = u$

then *x*

else *repeat* λ ( ).*u*, *v*

The whole function is

def *nd n* = **let** *tree* = *gen F G* (0, 0)

where $F(x, y) = x + 1, y + 1$

and $G(x, y) = x, y + 1$

repeat(*sort*(*mapt H tree*)))

where $H(x, y) = x^n + y^n$

Destructive streams The stream functions defined above contain no assignment statements, so the stream *s* still exists after it has been applied to the null list. It follows that being able to use a stream more than once, or "backtracking," is the normal mode of operation. Often a stream is no longer needed after it has been applied and can be implemented by a "destructive" routine, which contains private storage within itself to record the *tail* of the sequence. Successive items of the stream occupy the same storage. When the stream is applied to the null list, the private storage is reset by an assignment statement. Such a routine therefore destroys the original stream when it is applied to the null list. An *own* variable feature was introduced in Algol 60 to supply this kind of private storage, although there are some questions about how it should be implemented. The intention was to have a variable that was local to a block but that unlike the regular locals had a value that survived the activation of the block.

The version of the *own* variable introduced here is attached to a procedure rather than a block, and like a stream it depends on using a particular expression construction that can be used in any context. It does not rely on the addition of any special mechanical devices for its specification or implementation. The only rule used is that the body of a lambda expression is evaluated only

when the function is applied. The expression construction that introduces an *own* variable is a function that is written as an expression whose operator part is a lambda expression and whose operator lambda expression has a body that is a lambda expression. In other words, it is an expression of the form

$(\lambda x.\lambda y.\, M)\, N$

The *own* variable in the expression above is x, and when the expression is evaluated, x takes as initial value the value of N. In order to guard against assignments to this variable from outside, the initial value should be copied, so $(\lambda x.\lambda y.\, M)\,(copy\ N)$ is better. This ensures that the only way of assigning to x is by an assignment statement of the form $x := E$ within M.

Cascading streams Streams, like coroutines, are most useful in specifying a cascade of editing processes. An example of the method of constructing a program using streams is given next. The problem is taken from Dijkstra [1]. The input is made up of a sequence of words composed of letters, separated by any number of spaces, and terminated by spaces and a point. The input is assumed to be a character stream called *rnc*. The required sequence of characters replaces the separating spaces by just one space and reverses every other word, and it is then terminated with a point.

The function (*while letter*) takes a word from the head of the input, and (*while space*) takes spaces until it finds a nonspace. The function *absorbword* defined below has to be repeatedly applied to produce a word sequence.

```
def absorbword s = let w, s1 = (while letter s)
                   let sps, s2 = (while space s1)
                   w, s2
```

To produce a word stream this function has to be applied repeatedly by applying (*next absorbword*) to the character stream. The point at the end (in fact any nonletter) gives rise to a tail made up of an infinite sequence of empty lists. Now every other word has to be reversed. The sequence (*generate not false*) can be generated to record whether to reverse a word or not. The two streams can then be merged to form a resulting stream using the function

(*zips g*) **where** *g x y* = **if** *y* **then** *reverse x* **else** *x*,

which when applied to the word stream (*next absorbword rnc*) and (*generate not false*) produces a word stream with alternate words reversed. The next step is to produce a character stream from a word stream. The spaces and point have also to be inserted. The spaces can be inserted by prefixing a space to every word except the first. This can be done by applying (*maps* (*prefix space*)).

Finally the whole character stream is obtained by applying *concatl* and postfixing a point. The whole program becomes:

```
let s1 = zips g (next absorbword rnc) (generate not false)
    where g x y = if y then reverse x else x
let s2 = concatl(untils(null · t) (maps(prefix space) s1))
    postfixs point (ts s2)
    where postfixs x y = appendl y (us x)
```

As in all programming systems, there is some choice in the strategy that can be adopted. One of the important decisions seems to be the stage at which one changes from dealing with infinite streams to dealing with streams that represent lists. If a stream is denoted by an expression and is not named, or if it is named and the name is only used once, then it is valid to use a destructive representation of the stream. All the streams in the example above can be replaced by destructive streams.

It should be clear from these examples that any function that operates on, or produces, tree-like data structures can be adapted to operate on, or produce, stream functions.

Parsing relations It is possible to represent sets by lists, in a nonunique way, and to represent lists by streams. It is also possible to regard a relation as a function from an object to the set (or stream) of objects related to it. Using this notion it is possible to re-express the familiar top down syntactic analysis program in terms of "parsing relations." Suppose that for each symbol, whether terminal or nonterminal, there is a relation from a string to a set of strings. The relation that corresponds to a particular phrase is between a string and a string from which an initial segment that is an instance of a phrase has been removed. The relation can be considered a function from a string to a set of strings. If the function does not find its phrase at the head of the string, the result is the empty set. The relation that corresponds to each terminal symbol tests whether that symbol is at the head of the string. If so, then the result is a set containing one member, the tail of the string. If not, then the result is the empty set. If sets are represented by stream functions, then the function Q, defined below, operates on a terminal symbol to produce its corresponding relation.

```
def Q x s =
        if null s
        then nullists
        else if x = h s
            then us(t s)
            else nullists
```

If the string starts with the character x, then the result is the stream $us(t\ s)$; otherwise it is *nullists*, the stream that represents the empty set. There are two special relations that correspond to 1) the empty set whose value is always the empty set and 2) the relation corresponding to the nullist set whose value is the set containing one member, the original argument string. These parsers are defined below.

```
def nullstring s = us s

def empty s = nullists
```

From these relations new relations can be constructed by replacing each union by the union of two relations:

```
def union f g s = appendl(f s) (g s)
```

and by replacing each Cartesian concatenation operator by the concatenation of two relations.

```
def followedby f g s = concatss(mapl f(g s))
```

The relation $(followedby\ f\ g)$ therefore finds the set of strings related to s by f and for each member finds the set related by g and forms the union of this set of sets. The context-free productions of a language without left-recursive symbols can now be reinterpreted as a set of mutually recursive functions defining the parsing relation for the language. A string belongs to the language provided that the null list is a member of the set produced by applying the parsing relation for the language to the string. In other words if L is the relation for the language, then the string is recognized if $(exists\ null\ (L\ s))$ is *true*, where *exists* is defined below.

```
def rec exists p s =
        if nulls s
        then false
        else let x, y = s( )
                if p x
                then true
                else exists p y
```

In order to produce a parser from this recognizer it is necessary to elaborate a relation so that it produces a set of pairs. The first of the pair is the object produced from the phrase recognized at the head of the string. The second of the pair is the remaining string. The *followedby* function has now to be elaborated by providing another function as argument. The function $(cc\ h\ f\ g)$ defined below produces the set of pairs found by applying f and g and combining the results of f and g by applying the function h.

```
def cc h f g s =
        concatss (mapl q(f s))
            where q(u, v) = mapl r(g v)
                where r( y, z)
                = (h u y, z)
```

The *union* operator is unchanged.

Merging Occasionally two programs are found to be very similar in operation, and sometimes the difference can be accounted for by considering that one program is operating on lists and the other is carrying out the same operation on streams. One example of such a pair is a pair of programs that both merge a list of sorted strings. To merge two lists the smallest head of the two lists is selected and removed, leaving two sorted lists that then have to be merged in the same way.

```
def rec merge x y =
        if null x
        then y
        else if null y
            then x
            else if h x < h y
                then h x:merge(t x) y
                else h y:merge x(t y)
```

A nonempty list of items can be sorted by merging if one changes each item into a 1-list and repeatedly merges until one sorted list remains

```
def rec mergepairs y =
        if null y
        then ( )
        else if null (t y)
            then y
            else (merge(h y) (h(t y)):
                mergepairs(t(t y)))

def rec mmerge x =
        if null (t x)
        then h x
        else mmerge(mergepairs x)
```

The *merge* function can now be changed to operate on streams and produce a stream as follows:

```
def rec merge x y =
        if nulls x
        then y
        else if nulls y
            then x
            else if hs x < hs y
                then prefixs(hs x)
                     (merge(ts x)y)
                else prefixs(hs y)
                     (merge x (ts y))
```

It is therefore possible to combine a list of streams by merging to produce a stream for the sorted list. It is more efficient to represent the streams using a buffer of size *1* because the head of the stream is to be referred to more than once. In this new representation of a stream, each stream is represented by a pair whose first is the head of the sequence and whose second is the old stream representation of the tail. The new way of representing streams can be summarized as follows

nulls s	*(h s) = end*
nullists	*(end, generate 1 end)*
hs	*h*
ts(x, s)	*s( )*
prefixs x s	*x, λ( ).s*

With this change of representation the *merge* function becomes

```
def rec merge x y =
        let u, f = x
        let v, g = y
        if u = end
        then y
        else if v = end
            then x
            else if u < v
                then u, λ( ).merge(f( ))y
                else v, λ( ).merge x(g())
```

The function *mmerge* produces a stream pair whose first is the smallest item and whose second is a stream that takes the form of a tree or tournament, called a "loser" tree. The application of *mmerge* requires $n-1$ comparisons. The production of the remaining stream elements require at most $\log_2 n$ comparisons each. The same pairs of elements are compared in both programs, but the comparisons are carried out in a different order.

Efficiency One of the outstanding problems is to devise general rules for creating efficient programs from the notation used. Some of the possible program manipulations that might be carried out to produce a factorial program from

```
stream2 1 mult 1 (untils (>n)(generate(+1)1))
```

will be given next.

Two lambda convertible expressions that denote a function are equivalent, although they may operate in different ways under the assumed method of evaluation. It should be possible to choose the most efficient convertible form. It is possible to extract common subexpressions in the definition of *stream2* that follows

```
def rec stream2 a g f s =
        if nulls s
        then a
        else stream2(g(f(hs s))a)g f
             (ts s)
```

in order to make sure that the stream is only applied once when *(stream2 a g f)* is applied. This lambda conversion produces the definition

```
def rec stream2 a g f s =
        let x, y = s( )
        if x = end
        then a
        else stream2(g(f x)a)g f y
```

A second main method of producing efficient programs is to detect when a piece of storage is no longer needed and reuse it. The recursive call of *stream2* can be replaced by a **go to** instruction because the subroutine linkage information is unchanged by the inner application. The piece of storage that holds the arguments can also be reused if the arguments of one application of *stream2* are not required after the next. This recursion removal produces the program:

```
stream2 a g f s =
        let b = a
        let r = s
    L:let x, y = r( )
              if x = end
              then b
              else b := g(f x)b
                   r := y
                   g to L
```

Now the arguments can be substituted, producing

```
stream2 1 mult I s =
        let b = 1
        let r = s
    L:let x, y = r( )
        if x = end
        then b
        else b := x × b
             r := y
             go to L
```

The next stage is to notice that the stream is only used once within the program, so there is no question of backtracking in the stream. The stream can therefore be represented as a destructive stream that uses the same piece of storage for its elements. A destructive stream for (*generate f x*) can use just one variable *r* to represent both the head of the stream and the whole stream. The *ts* of the stream is (*f r*). The application of (*until s p*) to a destructive stream causes the expression (*x = end*) to be replaced by (*p x*). With these changes the program becomes:

```
    let b = 1
    let r = 1
L:let x = r
    and y = r + 1
    if x > n
    then b
    else b := x × b
         r := y
         go to L
```

Since **let** $x = r$ is merely a renaming and y only occurs once in the program qualified by its definition, they can both be substituted to give:

```
factorial n =
        let b = 1
        let r = 1
    L:if r > n
        then b
        else b := r × b
             r := r + 1
             go to L
```

Summary

The examples of structured programs in this paper are, for the most part, familiar programs that have been recast in an unfamiliar way. We hope they demonstrate that using expressions that denote objects of computation as opposed to using instructions that denote machine behavior can often result in a clearer structuring of programs. The use of stream processing methods often leads to the rapid development of a program having complexities that might be troublesome to master by using other methods.

The main problem is to be able to produce efficient but perhaps less structured programs automatically from the notation used. The compiler has to be able to detect those constructions that correspond to more conventional programming techniques, and to rearrange the program accordingly. The tentative suggestions made in this paper only touch the surface of this problem of producing efficient programs.

This paper contains some of the material from a chapter of a forthcoming book entitled "Recursive Programming Techniques" to be published by the Addison-Wesley Publishing Co. as one volume in the *IBM Systems Programming Series* of books.

Cited and general references

The notion of a stream is due to Landin (private communication 1962) and is briefly described in [6]. The construction is possible in several programming languages that are either based on the lambda calculus or permit coroutine constructions [7,8,9,10,11].

1. O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, London and New York, 1972.
2. P. J. Landin, "The Next 700 Programming Languages," *Commun. ACM* **9**, 157 (1966).
3. M. E. Conway, "Design of a Separable Transition-diagram Compiler," *Commun. ACM* **6**, 396 (1963).

4. A. Evans, "PAL – A Language Designed for Teaching Programming Linguistics," *Proc. 23rd ACM Conf.*, 395 (1968).
5. W. H. Burge, "Combinatory Programming and Combinatorial Analysis," *IBM J. Res. Develop.* **16,** 450 (1972).
6. P. J. Landin, "A Correspondence Between Algol 60 and Church's Lambda-notation," *Commun. ACM* **8,** Part 1, 89, Part 2, 158 (1965).
7. R. M. Burstall, J. S. Collins, and R. J. Popplestone, *Programming in POP-2*, Edinburgh University Press, Edinburgh, Scotland, 1971.
8. O. J. Dahl and K. Nygaars, "Simula – An Algol-based Simulation Language," *Commun. ACM* **9,** 671 (1966).
9. T. I. Fenner, M. A. Jenkins, and R. D. Tennent, "QUEST: The Design of a Very High Level, Pedagogic Programming Language," *SIGPLAN Notices* (ACM) **8,** 3 (1973).
10. J. C. Reynolds, "GEDANKEN – A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept," *Commun. ACM* **13,** 308 (1970).
11. J. E. Stoy and C. Strachey, "OS6 – An Experimental Operating System for a Small Computer," *Computer J.* **15,** No. 2, 117 and No. 3, 195 (1972).
12. S. W. Golomb and L. D. Baumert, "Backtrack Programming," *J. Assoc. Comput. Mach.* **12,** 516 (1965).
13. D. E. Knuth, "Structured Programming With Go to Statements," to be published.
14. P. Naur, "Programming By Action Clusters," *BIT* **9,** 250 (1969).

Received April 19, 1974; revised June 25, 1974

The author is located at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598.