Seque: A Language for Programming with Streams*

Ralph E. Griswold and Janalee O'Bagy

TR 85-2

January 26, 1985

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grants DCR-840131 and DCR-8320138.

Seque: A Language for Programming with Streams

1. Introduction

Many programming problems deal with values that are produced in sequence by some computational process. Such sequences often are potentially infinite or are most easily characterized by an unlimited computation. Seque is an experimental language that provides facilities for characterizing such sequences in terms of *streams*.

A stream is a data object that is capable of producing values. The values are produced on demand and, as such, constitute a sequence of values produced in time. The values that a stream can produce are specified by a sequence of computations.

Streams retain the values that they have produced. These values can then be accessed by position, according to the order in which they were produced. Thus, there is the notion of the first, second, ... *n*th values of a stream. Positional access does not require the recomputation of values that have already been computed. However, if a value that has not been computed is referenced by position, the necessary computations are performed to produce this value.

Another way to view a stream is as a composite of values that already have been produced together with values that potentially may be produced. Since not all the values that a stream can produce ever need to exist physically, a stream can produce an unlimited number of values. Such "infinite" streams are particularly useful as finite characterizations of infinite sequences.

Seque is embedded in the Icon programming language [1]. Embedding Seque in Icon facilitates its experimental nature, allowing ideas to be easily changed as the result of experience. In addition to ease of implementation, Seque benefits from the availability of many features of the base language, such as the string and list repertoire of Icon. More importantly, Icon's expressions, which can produce zero, one, or many results, allow the computational component of streams to be characterized in a natural and concise way.

There are a number of other programming languages that deal with streams and sequences. See, for example, [2-5]. Seque differs from these languages in a number of ways, including its support for nonhomogeneous streams, streams of streams, and self-referential streams. Icon, as the base language of Seque, provides versatile and compact ways of specifying streams.

The description that follows assumes a knowledge of Version 5 of Icon.

2. Constructing and Operating on Streams

There are many operations in Seque that produce streams. One of the most elementary stream-valued operations is a sequence of expressions:

{ $expr_1$, $expr_2$, $expr_3$, ..., $expr_n$ }

These expressions are evaluated in order from left to right, as needed, to produce the values of the corresponding stream. For example,

```
Primaries := { "red", "yellow", "green" }
```

assigns to Primaries a stream that is capable of producing the strings red, yellow, and green. Similarly,

```
In3 := { read(), read(), read() }
```

assigns to In3 a stream that is capable of producing three lines from the standard input file.

The values assigned to Primaries and In3 have type stream and are data objects that can be transmitted and operated on throughout the program like any other values.

In order to describe the relationship between a stream and the values that it is capable of producing, the following notation is used:

$$X \rightarrow < x_1, x_2, x_3, \ldots, x_n >$$

The sequence of values enclosed in angular brackets is an abstraction that represents the values that the stream may produce. A simple example is:

```
{ 2, 4, 6, 8 } → < 2, 4, 6, 8 >
```

Expressions in a stream can be arbitrarily complex and each can produce an arbitrary number of values. For example,

```
{ 1 to 3, !"abcd" } → < 1, 2, 3, a, b, c, d >
```

An expression in a stream also may fail to produce a value. For example, if s1 is not a substring of s2, then

```
Indicies := { 0, find(s1, s2) > 10 } → < 0 >
```

Notice that streams may be non-homogeneous with respect to type. Streams may also have streams as values, as in

```
Palette := { Primaries, { "black", "white" } }
```

In Seque, the term *scalar* is used to refer to any value that is not a stream. For example, all the values of Primaries are scalars, whereas the values of Palette are non-scalars.

The length of stream X is the number of values that X is capable of producing and is denoted by |X|. For example, |Primaries| = 3, |Palette| = 2, and |Indices| = 1.

There are three predefined streams :

```
Phi     → < >
Izero   → < 0, 1, 2, 3, 4, ... >
Iplus   → < 1, 2, 3, 4, 5, ... >
```

Phi is the empty stream, containing no values, thus |Phi| = 0. The infinite streams Izero and Iplus are useful in the construction of more complex streams.

2.1 Referencing the Values in a Stream

The values produced by a stream may be referenced by position. X!i is the ith value produced by stream X. For example,

```
Primaries!2
```

produces the string yellow. Similarly, if

```
Alpha := { !&lcase, !&ucase }
```

then Alpha!2 produces the string b.

Referencing the ith value of a stream causes all the values prior to the ith value to be computed and saved. The values of a stream, once computed, are never recomputed.

2.2 Generating the Values of A Stream

The values of a stream X may be generated in sequence with the element generation operation @X. For example,

```
every @Primaries
```

generates the strings red, yellow, and green. Similarly,

```
every write(@Iplus) \ 10
```

writes the first ten positive integers.

2.3 Concatenation

The concatenation of two streams is specified with X -> Y and is a stream consisting of the stream for X followed by the stream for Y. For example,

```
{ 1, 2, 3 } -> { "a", "b", "c" } → < 1, 2, 3, a, b, c >
```

Concatenation may involve streams that are infinite. Thus

```
{ -3, -2, -1 } -> Izero → < -3, -2, -1, 0, 1, 2, 3, ... >
```

2.4 Sectioning

Streams consisting of portions of existing streams are created by sectioning operations. Sectioning is denoted by X{i:j} where $1 \leq i \leq j \leq |X|$. Sectioning produces a stream consisting of the i through j values of X, inclusive. For example,

```
Iplus{3:5} → < 3, 4, 5 >
```

A common form of sectioning involves eliminating either initial or final values of a stream. These forms of sectioning are called pre-truncation and post-truncation, and are denoted respectively by

X %% i $\rightarrow < x_{i+1}, x_{i+2}, x_{i+3}, \ldots, x_{|X|} >$
X \\ i $\rightarrow < x_1, x_2, x_3, \ldots, x_i >$

For example,

```
Izero %% 10 → < 11, 12, 13, 14, ... >
```

and

```
Iplus \\ 10 → < 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 >
```

2.5 Reduction

Reduction over a binary operation f is achieved by the function Red(X, f). Reduction applies f to the values of stream X and produces a stream of the accumulated results. The second argument f may be a procedure or a string representing a binary operation.

For example, let I be the stream { 1, 2, 3, 4 }. Reduction of I over addition produces a stream that consists of a running sum of the values in I. That is,

```
Red(I, "+") → < 1, 3, 6, 10 >
```

Similarly, if the operation is concatenation, the result is the concatenation of all the values of produced by the stream. For example, if

```
X := { "a", "b", "c", "d" }
```

then

```
Red(X, "||") → < a, ab, abc, abcd >
```

2.6 Operations over Streams

The operations of Icon with operator syntax, such as +, *, and !, may be applied to streams. The operations are *implicitly* distributed over the values of the stream. In the case of a binary operation, the operation is distributed pairwise over the values of the streams, producing the dot product. In general, if ⊕ is a binary operator, then

X ⊕ Y $\rightarrow < x_1 \oplus y_1, x_2 \oplus y_2, x_3 \oplus y_3, \ldots >$

For example,

```
{ 1, 2, 3 } + { 4, 5, 6 }  → < 5, 7, 9 >
```

If the streams have unequal lengths, the operation is performed for the first *i* values, where *i* is the minimum of the two lengths, and the resulting stream has length *i*. Thus,

```
Iplus + { 4, 5, 6 } → < 5, 7, 9 >
```

If one of the operands is a scalar, it is coerced to the corresponding unit stream. That is,

```
{ 1, 2, 3 } + 4 ≡ { 1, 2, 3 } + { 4 } → < 5 >
```

Operations are also applicable to streams that have streams as values. Consider the following example:

```
{ { 1, 2, 3 }, 6, 7, 8 } + { 1, 2, 3 } ≡ { { 1, 2, 3 } + 1, 6 + 2, 7 + 3 } ≡ { { 2 }, 8, 10 }
```

Unary operations defined for scalars are also applicable to streams, and in a similar manner, are applied successively to each value in the stream. For example, if

```
X := { 1, 2, 3, 4, 5 }
```

then

```
-X → < -1, -2, -3, -4, -5 >
```

Similarly, a stream consisting of the lengths of the lines from the file f is

```
*{ !f }
```

2.7 Functions on Streams

Seque provides several functions that operate on streams. The most commonly used functions are described here. Additional functions are described in the Seque reference manual [6].

The function Length(X) computes the length of the stream X. The length of a stream is the number of values it produces. For example, if

```
Lowers := { !&lcase }
```

then Length(Lowers) is 26. In order to determine the length of a stream, all the values a stream is capable of producing must be computed. Clearly, the computation of Length() on an infinite stream does not terminate.

The function Swrite(X) writes the values of a stream, one-per-line, to standard output. Swrite(X) expects the values produced by X to be scalars. Note that

```
Swrite(X) ≡ every write(@X)
```

The function Simage(X, i) produces a string that is the image of X, limited to i results. The default value for i is 5. Simage is recursive: if the values of X are themselves streams, Simage follows the structure of the stream. For example,

```
Simage(Palette)
```

produces the string

```
{ { "red", "yellow", "green" }, { "white", "black" } }
```

The function Compress(X) "flattens" a stream that contains streams as values. All streams are compressed to correspond to a sequence of scalar values. For example,

```
Simage(Compress(Palette))
```

produces the string

```
{ "red", "yellow", "green", "white", "black" }
```

3. Derived Streams

The values of a function defined over a subset of the integers are frequently useful. For example the sequence of the squares of the positive integers

1, 4, 9, 16, 25, 36, ...

can be expressed as

```
i ^ 2
```

where i takes on the values of Iplus. In Seque, the values of a stream may be derived from an expression over the positive integers by surrounding an expression in brackets. For example,

```
Squares := [ i ^ 2 ] → < 1, 4, 9, 16, 25, 36, ... >
```

Note that the variable i is distinguished in this context and is implicitly associated with Iplus. Streams created in this way are called *derived streams*.

Other examples of derived streams are

```
[ i - 2 ]  → < -1, 0, 1, 2, 3, 4, 5, 6,  ... >
[ i % 4 ] → < 1, 2, 3, 0, 1, 2, 3, ... >
[ "b" ] → < b, b, b, b, b, ... >
[ repl("a", i) ] → < a, aa, aaa, aaaa, ... >
```

The expression within brackets may be a stream, in which case the derived stream is the repeated concatenation of the bracketed stream. Thus, if

```
Three := { 1, 2, 3 }
```

then

```
[ Three ] ≡ { 1, 2, 3 } -> { 1, 2, 3 } -> { 1, 2, 3 } -> ... →  < 1, 2, 3, 1, 2, 3, 1, 2, 3, ... >
```

A derived stream may use a controlling stream other than Iplus. A derived stream is completely specified by

[: S : lambda(s) *expr*]

S is the *controlling stream* and may be any stream-valued expression. The variable s is the *bound variable* and takes on successive values from the controlling stream. All occurrences of s in *expr* are bound. As in the previous examples, when not specified, the controlling stream defaults to Iplus and the bound variable defaults to i.

As an example of specifying the controlling stream, assume that Squares is the stream of the squares of integers as shown above. Then the strings of a s whose lengths are perfect squares are derived with the expression

```
[ : Squares : repl("a", i) ] →  < a, aaaa, aaaaaaaaa, ... >
```

The controlling stream is completely general; it does not have to be integer-valued. For example, using the following derived stream whose values are the strings in the closure of a,

```
Astar := [ repl("a", i - 1) ]
```

the strings of the form a^j ba^j, where $j \leq 0$, are defined by

```
[ : Astar : lambda(s) s || "b" || s ]  → < b, aba, aabaa, aaabaaa, aaaabaaaa, ... >
```

If the expression within a derived stream contains variables other than the bound variable, the values of these variables are inherited from the context in which the derived stream is created. For example, if

```
j := 10
```

then

```
[ i + j ]  → < 11, 12, 13, 14, 15, 16, ... >
```

Derived streams may be nested. As an example, consider the following expressions, which produce the cartesian product of the values of two streams:

```
U := { "A", "B", "C" }
L := { "a", "b", "c" }
[ : U : lambda(s1) [ : L : lambda(s2) s1 || s2 ] ]
```

For each value of U, the concatenation is performed with each value of L, producing

```
< Aa, Ab, Ac, Ba, Bb, Bc, Ca, Cb, Cc >
```

4. Changing the Values in a Stream

The values of a stream are saved as they are computed. A previously computed value may be changed by assignment. For example, consider the following expressions:

```
X := {"r", "a", "d", "a", "r" }
   .
   .
   .
X!1 := X!5 := "m"
```

Initially the concatenation of the values of X forms the word radar. After the assignments, it forms the word madam.

Notice that self-referential streams may be created using assignment. Such a stream is recursively infinite, as in the example below:

```
A := { "a", "b" }
A!2 := A
```

After the assignment, A is a stream whose first value is a and whose second value references itself. The structure of A is illustrated by Simage(A), which produces

```
{ "a", { "a", { "a", { . . . } } } }
```

5. Examples

Filtering

Conditional expressions in a derived stream may be used to examine and selectively choose values from a stream. For example, the following expression filters the positive integers and selects those that are evenly divisible by three:

```
[ if (i % 3) == 0 then i ] → < 3, 6, 9, 12, ... >
```

Similarly, the expression below produces a stream of all the lines in standard input that contain the string the.

```
[ :Read(): lambda(s) if find("the", s) then s ]
```

Reduction

Reduction is used in a variety of ways. For example, the following program fragment determines the longest line of file f:

```
LL := Red( *{ !f }, "<")
write( LL!Length(LL) )
```

The stream argument to Red() consists of the lengths of the lines of standard input. LL is the stream of values produced by the successive comparisons made by the reduction, and its last element is the maximum line length.

Assignment to Stream Values

Often an inherently recursive problem can be solved iteratively in Seque by making use of assignment to stream values. In fact, using assignment in the expression of a derived stream leads to a concise, expressive linguistic idiom that is useful for a variety of problems.

As an example, consider the following program that writes, in lexical order, the strings in the closure of the set of strings a, b, and c over concatenation.

```
procedure main()
   X := { "a", "b", "c" }
   Closure :=  [ "" ]
   Closure := [ Closure!(i+1) := Kross(X, Closure!i)]
   every write(@Closure)
end

procedure Kross(X,Y)
   return [ :X: [ :Y: lambda(j) i || j ] ]
end
```

The procedure Kross returns a stream consisting of the cartesian product concatenation of its two argument streams. In the main program, X provides the a, b, and c and Closure begins as an infinite stream of empty strings. The effect of the expression in the derived stream is to successively define each ith value of Closure as the cartesian product of X with the previous value of Closure. In the expression, i takes on the values from the implicit controlling stream Iplus, giving

```
Closure(1+1) := Kross(X, Closure!1) →  < a, b, c >
Closure(2+1) := Kross(X, Closure!2) →  < aa, ab, ac, ba, bb, bc, ... >
Closure(3+1) := Kross(X, Closure!3) →  < aaa, aab, aac, aba, abb, abc, ... >
                              :
```

The values produced by the stream Closure are

```
a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, aab, ...
```

Sieves

Another example of this idiom appears in deriving the prime numbers by the sieve of Eratosthenes. This also provides an example of the ease and convenience of using infinite streams as data objects and illustrates the generality of the form of a controlling stream within a derived stream.

The sieve methhod for finding the prime numbers is a process of elimination: the composite numbers are "sieved out" by their property of divisibility. Beginning with the stream of positive integers { 2, 3, 4, 5, 6, ... }, the first value, 2, is the first prime. A new stream is constructed by eliminating the remaining values that are obviously not prime, that is, any value that 2 divides evenly. This gives the stream { 3, 5, 7, 9, 11, 13, ... }. The first value of this stream, 3, is the next prime. The process continues by eliminating the values in this stream that are divisible by the first value, 3. The resulting stream is { 5, 7, 11, 13, 17, ... }. By successively generating the streams described above, the primes are easily obtained by taking the first value from each stream. The following Seque program generates the primes in this manner:

```
procedure main()
   I := %Iplus
   Z := [ 0 ]
   Z!1 := I
   X := [ : [ : I : { Z!i := Sieve(Z!(i - 1)) } ] : lambda(Y) Y!1 ]
   every write(@X)
end

procedure Sieve(X)
   return [ if (X!i % X!1) > 0 then X!i ]
end
```

The stream Z begins as { { 2, 3, 4, 5, 6, ... } 0, 0, 0, ... }. Note that within the expression that defines X, the controlling stream is itself a derived stream. Within that derived stream is the expression that successively redefines the values of Z to be the streams described above. Beginning with the second value, the values of Z are defined by calling the procedure Sieve with the preceding value of Z. Sieve is called with a stream argument and returns a stream: the result of eliminating all the values from its argument stream that are divisible by its first value. As the calls to Sieve are made, Z becomes the stream

{ {2, 3, 4, 5, 6, ... }, { 3, 5, 7, 9, 11, ... }, { 5, 7, 11, 13, 17, ... }, ... }

This evolving stream is the controlling stream for the outer expression operating on Y. To generate the primes, the expression references the first value of each of the streams given to it.

6. Conclusions

Streams in Seque can be constructed by specifying their values explicitly or in terms of expressions that produce the values in a variety of ways. Derived streams, in particular, provide a concise way of representing many commonly encountered sequences. Streams can be infinite, they can contain any kinds of values, and they can be self-referential.

Icon, as the base language, contributes in a significant way both to the power and to the character of Seque. Icon's string-processing repertoire makes it easy to deal with streams of strings. More importantly, Icon's expression evaluation mechanism, in which an expression may produce a sequence of values, meshes naturally with the concept of streams.

This report describes only the essential aspects of Seque. In addition to the features described here, Seque has a number of other operations on streams and a facility for defining streams in terms of recurrences. There are also a number of technical issues that arise in programming with Seque that are not discussed here, including changes to the syntax of the Icon base language in order to accommodate Seque constructions. A complete description of Seque is contained in a reference manual [6].

Acknowledgements

Tom Hicks and Jorge Ochoa-lions provided helpful comments on drafts of this technical report.

References

1. R. E. Griswold and M. T. Griswold, *The Icon Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
2. I. Nataka and M. Sassa, *Programming with Streams*, Technical Report RJ 3751, IBM Research Laboratory, San Jose, California, Jan. 1983.
3. D. A. Turner, "Recursion Equations as a Programming Language", in *Functional Programming and its Applications; An Advanced Course*, J. Darlington, P. Henderson and D. A. Turner (ed.), Cambridge University Press, 1982, 1-28.

4. E. A. Ashcroft and W. W. Wadge, "Lucid, a Nonprocedural Language with Iteration", *Comm. ACM 20*, 7 (July 1977), 519-526.

5. W. H. Burge, *ISWIM Programming Manual*, Technical Report RA 129, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, Nov. 1981.

6. R. E. Griswold and J. O'Bagy, *Reference Manual for the Seque Programming Language*, The Univ. of Arizona Tech. Rep., in preparation.