Functional programming with streams

Tetsuo Ida* and Jiro Tanaka**

* Information Science Laboratory
Institute of Physical & Chemical Research
2-1, Hirosawa, Wako-shi, Saitama, 351, Japan
Telephone 0484-62-1111

** Department of Computer Science
University of Utah
Salt Lake City, Utah, 84112, U.S.A.

This research is supported in part by grant in aid of Ministry of Education No. 57780047.

Abstract

The advantage of programming paradigm to be called "stream programming" is discussed. The expressive power and conceptual clarity of stream programming derive from the fact that a stream is formulated as a natural extension of iterations as opposed to lazy evaluation approach. Streams are embedded in a functional programming system FFP. This permits clear and efficient treatment of streams. A set of primitive functions for stream programming are given and some examples are shown. The outline of the implementation of the primitives are also given.

§1 Introduction

The notion of a stream to encapsulate the (possibly) infinite flow of data has been used in several programming disciplines [1,2,3,4]. It is widely recognized that the use of a stream makes a certain class of programs simpler and clearer [1, 5]. A well-known method for processing a stream is to realize a stream as a recursive data structure and to evaluate it to the extent that only necessary part of the data structure is constructed. The operational semantics of the recursive treatment can be defined by a "lazy evaluation" model [6].

In this paper, we tackle streams from the other aspect of computing, i.e. iteration, in functional programming context. A stream is viewed as consisting of infinite number of elements that are generated one in each iterative loop. Iteration in its simplest form is expressed in recurrence relations

$$x_{i+1} = f:x_i,\ i = 1, 2, \cdots. \qquad (1)$$

where f is an arbitrary function, to be called a generator. The relationship between a stream and iteration was first observed by Landin [5] in 1965 in formalizing for-construct of ALGOL 60 by Lambda notation.

Our approach is based on the observation that a wide class of computations which lend themselves to recursive formulations are equally, or in many cases, more naturally, expressed in recurrence relations.

For example, we have a definition of Fibonacci number sequence stated as follows:

- the first two elements of the sequence are one,
- elements except the first two are the sum of the two preceding ones.

This definition is expressed in recurrence relations

$$x_1 = 1.$$
$$x_2 = 1.$$
$$x_i = x_{i-1} + x_{i-2}. \qquad (2)$$

which is more natural translation of the above statement than is defined in a recursive equation*1.

ソフトウェア基礎論 3～8 (1982. 12. 10)

fib = apndl:<1, f_2> (3)

where

f_2 = apndl:<1, add1s:<fib, f_2>>

and add1s is defined as

add1s = apndl·[plus·[1·1, 1·2], add1s·[tl·1, tl·2]]

or fib:1, fib2:, ---, using a recursive function

```
fib =   eq0 -> 1;                                   (4)
        eq1 -> 1:
        plus·[fib·sub2, fib·sub1].
```

---

foot note

*1 Throughout this paper, we use a functional programming language FP [7] for the description of the program. Refer to [7] for its syntax and semantics.

Taking <1, 1> as an initial input to an iterative loop (1) and using a generator [plus, 1], we can successively generate pairs of consecutive Fibonacci numbers. This idea is generalized in §2. In §3-6 we introduce basic functions of stream processing, and in §7 examples of stream programming are given. In §9 we discuss our approach with regard to studies on transformation of recursive-to-iterative program schemas.

We are motivated to the theme of the paper from the following considerations. Functional programming languages which rely on (essentially) applicative order of evaluations, notably FP, can exploit parallelism straightforwardly, and is more amenable to massive parallelism such as provided by Mago's machine [8]. Despite these advantages, this class of languages have a disadvantage that the evaluation of recursive equations such as (3) would not terminate. The scheme for stream processing to be presented in this paper effectively overcomes these difficulties.

§2 Definition of a stream

A stream is an infinite sequence $<x_1, x_2, \cdots, x_n, \cdots>$ whose element x_i is enumerated using two functions g and s, to be called a generator and a selector

respectively, and accumulator A_i by following relations:

$A_{i+1} = g:A_i \quad i=1, 2, \cdots$ (5)

$x_i = s:A_i$,

and A_1 is given.

We impose the condition that the enumeration of A_i requires the previous history of generation x_{i-1}, x_{i-2}, ... x_1, at most. In most cases, history dependency is weak, as we see in the following examples (a)-(d).

Hence, a stream is represented as a triple (A, g, s) where A is often called a seed of the stream rather than an accumulator.

examples of streams

(a) Integer sequence

(1, add1, id)

where add1 and id are defined by add1:n = n+1 and id:n = n.

(b) Power series $<a, ar, ar^2, \cdots, ar^n, \cdots>$

Similarly, a power series is defined as

(a, <bu, mult, r>, id)

(c) Fibonacci number sequence, fib defined in (2) or (3)

fib = (<1,1>, [plus, 1], 1)

This is a straightforward translation of (2) using accumulators in which two consecutive elements in the stream are saved.

(d) Finite sequence

A finite sequence $<x_1, x_2, \cdots, x_n>$ may be represented as a stream by adding infinite number of special element "eos" to the right of x_n, if we choose

$A = \langle x_1, x_2, \cdots, x_n \rangle$

g = null -> [$\overline{eos}$]; tl

$s = 1$

Although example (d) would lead us to generalize the concept of a stream by allowing finite sequence to be a stream, we rejected this generalization since it may complicate the laws of algebra of programs. For example reverse·reverse ≠ id, as indicated by Cohen and Meyers [9].

A stream has its own peculiarity such as:

- apndr is not allowed on infinite stream,
- appending an element J to the left of stream (A,g,s)
  is difficult, if not impossible, unless $J = s \cdot g^{-1}:A$
  or it is an instance of example (d).

Therefore, in this paper we consider a stream as a new data type (the reason for ( ) being used to denote a stream) and confine our attention to the stream and its primitive functions which operate on it.

§3 Operational definition of stream processing functions

We define two basic stream processing functions, "first" and "rest". These correspond to functions on finite sequences, "1" and "tl", respectively. "first" is a selector function which returns the first element of a stream. "rest" is a transformer which removes the first element of a stream and returns the rest of the stream.

Operationally, we can define "first" and "rest" as follows:

first:(A, g, s) = s:A (6)

rest:(A, g, s) = (g:A, g, s) (7)

Use of "first" and "rest" enables us to simply regard a stream as a pool of elements which successively supplies the first element, upon demand ("first"), and moves forward ("rest"). To view it differently, "A" is a state vector which determines the state of computation, "first" sees current value encoded in the state and "rest" carries out a computation one step further in the iterative loop of generating an element of a stream. However, in our treatment, the state information is contained in the triple and is invisible to the programmer.

As an illustration of the usage of "first" and "rest" we define function "prefix"*1 which gives a sequence of first n elements of stream $X = \langle x_1, x_2, \cdots x_n, \cdots \rangle$.

That is,

<prefix, n>:X = $\langle x_1, x_2, \cdots, x_n \rangle$

Using auxiliary function "prefix0", we obtain the following function definitions*2:

def prefix = prefix0·[2.1, $\overline{nil}$, 2]

def prefix0 = eq0·1 ->2; prefix0·[sub1·1, apndr·[2, first·3], rest·3]

Note that streams can be treated as if they had the same structures as finite sequences except that "first" and "rest" are used in place of "1" and "tl".

---

foot note

*1 Strictly speaking, distinction should be made between functions and functionals. However, the distinction is not essential in this paper, and we consistently use the word 'function' to denote either of the two.

*2 Meta composition $\langle g, f_1, \cdots, f_n \rangle:x = g:\langle\langle g, f_1, \cdots, f_n \rangle, x \rangle$ is used here [7].

---

§4 Programming with streams

Figure 1 shows a typical process of stream programming. A stream is created by a function "gen".

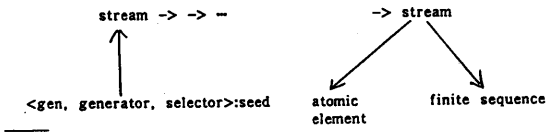
<gen, g, s>:A ⇒ (A, g, s),

where g, s and A are a generator, a selector and a seed, respectively.

Streams are transformed in several stages by stream-to-stream functions (cf. §5) and

are finally reduced to an element using reduction functions (cf. §6) or to a finite sequence using "prefix". Functions "first" and "rest" are not explicitly used. Moreover, recursion does not appear at this level of reasoning about programs.

Figure 1 Process in stream programming



§5 Stream-to-stream functions

We give basic stream handling functions which map streams to streams. Correspondence is noted between those to be given and those which operate on finite sequences.

- Map, βf applies f to all elements of the stream.

  βf:⟨A, g, s⟩ ⇨ ⟨A, g, f·s⟩

  β corresponds to α (apply to all) in the case of a finite sequence.

- stdistr:<stream, K> distributes K to the right of each element of a stream and forms a pair.

  stdistr:<⟨A,g,s⟩,K)> ⇨ ⟨A, g, [s,$\overline{K}$]⟩

- Similarly, stdistl:<K, stream> distributes K to the left of each element of a stream and forms a pair.

  stdistl:<K, ⟨A,g,s⟩> ⇨ ⟨A, g, [$\overline{K}$, s]⟩

- combine:<X_1, X_2> combines two streams by pairing each element.

combine:⟨X_1, X_2⟩

⇨ ⟨<A_1, A_2>, [g_1·1, g_2·2], [s_1·1, s_2·2]⟩

where X_1 = ⟨A_1, g_1, s_1⟩ and X_2 = ⟨A_2, g_2, s_2⟩

- filter constructs a new stream by removing all the element x_i which satisfies relation p:x_i = F(false) out of a stream X = <x_1, x_2, -- >

  <filter, p>:⟨A, g, s⟩ ⇨ ⟨A', g', s⟩

  where A' = g':A

  and g' = <while, not·p·s, g>

Corresponding to the usual set operations, we can define stream-union, stream-difference and stream-intersection functions, provided that elements of the streams are ordered by some binary predicate "orderp".

Let X and Y be streams.

- stream-union

  <stunion, orderp>:<X, Y>

- stream-difference

  <stdiff, orderp>:<X, Y>

- stream-intersection

  <stintx, orderp>:<X, Y>

These functions only map a stream to a stream and do not initiate computation for generating elements of the new stream. Thus, in programming with these functions, our reasoning stays at a stream level.

Observe that a new triple is created by cascading functions using composition and construction. We virtually build machinery for lazy evaluation not into data domain, but into functional domain.

§6 Reduction functions

FP provides right-insert and left-insert functions for reducing a finite sequence to an element. In the case of streams, we can only reduce the stream from left to right, and furthermore we need to specify the termination condition. We provide two functions; "redn" and "redp".

That is,

<redn, n, f, c>:<x_1, x_2, —, x_n, —>

= <f—<f:<f:<c, x_1>, x_2>, —, x_n>

<redp, p, f, c>:<x_1, x_2, —, x_n, —>

= not·p·f:<c,x_1> -> f:<c,x_1>; <redp, p, f, f:<c,x_1>>:<x_2, —, x_n, —, >

c is usually a left identity element of binary function f. "redn" terminates by counting number of application of f, and "redp" by applying predicate p to the intermediate result.

Function "prefix" defined in §3 may be defined as

<prefix, n> = <redn, n, apndr, nil>.

Wadler gave four program transformation rules involving α and reduce operators on finite sequence [10], to avoid immediate expansion of sequence. In our treatment, self-optimization is automatically in effect. For example,

$\beta f \cdot \beta g$:⟨A, g, s⟩

⇨ ⟨A, g, f·g·s⟩

⇨ βh:⟨A,g,s⟩ where h = f·g

Furthermore, we can apply a general transformation rule, say Ω, separately on a generator and a selector, before reduction actually takes place,

i.e. Ω:⟨A,g,s⟩ ⇨ ⟨A, Ω:g, Ω:s⟩.

§7 Examples of programs with streams

A cartesian product of two streams and a prime generator are considered in this section because they give a flavor of stream programming using some of the primitive functions introduced so far.

Cartesian product of two streams

A straightforward method for taking cartesian product of two finite sequences is by:

cp = αdistl·distr (8)

To remove outermost level of grouping, cp may be left-composed by a function flatten (= /(/apndl·apndr)). Corresponding to (8), we could have a function which builds a cartesian product of two streams:

stcp' = βstdistl,stdistr

However, <prefix,n>·stcp':<X,Y> where X=<x_1, —, x_n, —> and Y = <y_1, —, y_n, —> is unsatisfactory because it only generates first n streams

<<x_1,y_1>, <x_1,y_2>, — >, <<x_2,y_1>, <x_2,y_2>, — >, —, <<x_n,y_1>, <x_n,y_2>, — >.

Rather, we would like to have a cartesian product

<x_1, y_1>

<x_1,y_2>, <x_2,y_1>

<x_1,y_3>, <x_2,y_2>, <x_3, y_1>

.

.

.

To do that, we start with X and successively generate a stream stdistl:<x_i,Y>, i=1, 2, — while removing the first element of the already generated streams, as illustrated in Fig. 2. The change in A_i is shown in Fig. 3.

Thus, we have a function "stcp", which satisfies the following relations.

stcp:<X, Y> ⇨ ⟨A, g, s⟩

where

s = αfirst·3

g = [rest·1, apndl·[stdistl·[first·1, 2], αrest·3]]

A = <rest:X, Y, stdistl:<first:X, Y>>

The k-th iteration, i.e. first·rest^{k-1}·stcp:<X, Y> generates a sequence consisting of all pairs $<x_i, y_j>$ where i+j-1=k. Finally, flatten·<prefix, n>·stcp:<X,Y> generates the first n(n+1)/2 elements of the cartesian products, as desired.

Figure 2 Cartesian product of streams

X =	x_1	x_2	x_3	x_4	x_5	—
	(x_1, y_1)	(x_1, y_2)	(x_1, y_3)	(x_1, y_4)	(x_1, y_5)	—
		(x_2, y_1)	(x_2, y_2)	(x_2, y_3)	(x_2, y_4)	—
			(x_3, y_1)	(x_3, y_2)	(x_3, y_3)	—
				(x_4, y_1)	(x_4, y_2)	—
					(x_5, y_1)	—

Figure 3 Change of seed in cartesian product

<X, Y, stdistl:<first:X, Y>>
<rest:X, Y, stdistl:<first:(rest:X),Y>
<rest^2:X, Y, stdistl:<first:(rest^2:X),Y>
.
.

prime generator

A stream of prime numbers, "primes", is defined in the following (9) -(11), using the sieve of Eratosthenies. We first define integer stream starting from 2, "from2" in (9). Given a stream $x_1, x_2, \cdots$, "sieve" in (11) generates a new stream with all the multiple of x_1 removed. The first elements of successively generated streams are primes.

from2 = <gen, add1, id>:2 (9)

primes = <gen, sieve, first>:from2 (10)

sieve = apply·[(((filter, ((comp, $\overline{\text{neq0}}$, ((bu, $\overline{\text{rem}}$, first)))))), rest] (11)

where <bu, f, x>:y = f:<x, y>,

neq0 = not·eq0

rem:<x,y> = remainder of x/y

Here, we have extended FP by incorporating hyper-meta-composition rule [11].

((g, f_1, —, f_n)):x:y = <g, f_1:x, —, f_n:x>:y

e.g. ((comp, f, g)):x:y = (f:x)·(g:x):y.

§8 Implementation of stream processing functions

All the stream processing functions can be built up from "first", "rest" and other basic FP primitives. "first" and "rest" are embedded in FFP system, a formal system for FP, since they require function apply, i.e. apply·[f, x] = f:x. We first introduce auxiliary functions "genfun", "selfun" and "seed" which, given a stream, returns a generator, a selector, and a seed, respectively.

Now, from (6) and (7), we have following definitions:

first = apply·[selfun, seed]

and rest = apply·[((gen, genfun, selfun)), apply·[genfun, seed]]

Implementation details of the other functions are given in [11].

§9 Relation to recursion-to-iteration program transformations

While programming with recursion is based on a stack, programming with a stream is based on a first-in-first-out queue. The former is believed to be less efficient generally, because the stack is scanned twice, forward from right to left, and then backward. On the other hand iteration generally scans the queue once from left to right. Here the notion of right-left is understood to be in accordance with that of right-insert and left-insert functions. Elimination of recursions has been the object of study chiefly because of this efficiency consideration. However, simply changing programs symbolically obscures the behaviour of programs. Introduction of streams can give a clear interpretation.

Kieburtz and Shultis [12] proved the following as a special case of a dynamic programming transformation.

fib = p -> $\overline{1}$; plus·i (12)

i = p·sub1 -> [$\overline{1}$, $\overline{1}$]; h·i·sub1

p = or·[eq0, eq1]

h = [plus, 1]

"i" in the second line of (12) is essentially the iteration loop of Fibonacci number generation with "h" as a generator.

<i:1, i:2, ---, > is indeed the stream given in (1) with the selector being replaced by id.

We see that programming with streams enables us to start at the same level of program (12) without mechanical transformation of programs.

Another interesting scheme of recursion-to-iteration transformation is the following [12];

f = p -> k; h·[i, f·j] is transformed

to f = 1:<while, not·p·2, [h'·[1, i·2], j·2]>·[g, id] (13)

The above transformation is possible under the following condition:

(i) $\forall$ a, b, c, $\exists$ h'

h·[a, h'·[b, c]] = h'·[h·[a, b], c]

(ii) $\forall$ a, h·[a, k] = h'·[k, a]

(iii) k·j = j

We are interested in the sequence of values

<f:e, f·j^{-1}:e, f·j^{-2}:e, ---, > (14)

where we choose seed e, such that e satisfies p:e = T. We assume the existence of inverse of j.

The stream (14) is realized by ⟨A, g, 1⟩ where g = [h'·[1, i·2], j^{-1}·2], and A = <g:e, e>.

Now it is easy to give intuitive interpretation to (13). We start with a stream βi:<j^{-1}:e, j^{-2}:e, ---, > by reducing this stream using <redp, h', $\overline{T}$, g:e> while at the same time leaving intermediate results in another stream, which is the stream

given in (14). Therefore, it is essential that we can find h' such that we can proceed from left to right rather than right to left using h.

§10 Conclusion

We showed "stream programming" paradigm using basic stream processing functions. Iteration can be given wider meaning by forming the series of values which is computed by iteration into a stream. With streams, we can give a clearer interpretation to programs than to concentrating each value generated in the iterative loop.

The introduction of streams in functional programming can actually reduce the complexity of the computation involved. The decrease in the complexity results because the use of streams can algorithmically transform programs from recursive forms to iterative forms. The infiniteness of streams demands that actual computation involved be postponed to the time of reduction to finite object, i.e. the time of application of "redp", "redn" or "prefix". Stream level computation only induces functional composition and construction. This fact enables functional algebra to play more role towards optimization of programs.

References

1. W.H. Burge Recursive Programming Techniques, Addison-Wesley, Reading (Mass), 1975
2. P. Henderson Functional Programming, Application and Implementation Prentice/Hall International, London, 1980
3. R.M. Keller, FEL (Function-Equation Language) Programmer's Guide AMPS Technical Memorandum No. 7, Department of Computer Science, Univ. of Utah, March, 1982
4. M. Sassa, I. Nakata, et. al Programming with streams (in Japanese) Proc. 1982 Fall Congress of Japan Information Processing Society, Oct. 1982
5. P.J. Landin A correspondence between ALGOL 60 and Church's lambda Notation:Part 1, CACM 8(2), 89-101, Feb. 1965
6. P. Henderson and J. M. Morris "A lazy evaluator", Proc. 3rd POPL Symposium, 1976
7. J. Backus, Can programming be liberated from von Neuman style? A functional style and its algebra of programs, CACM 21(8), 613-641, Aug. 1978
8. G. Mago' A Network of Microprocessors to Execute Reduction Languages, International Journal of Computer and Information Sciences, two parts, 8(5), pp. 349-385, 1979, and 3(6), pp. 257-266, 1979
9. A.T. Cohen and T.J. Myers Towards an Algebra of Nondeterministic Programs, 235-242, Conf. Record of the 1982 ACM Symposium on LISP and Functional Programming
10. P. Wadler, Applicative Style of Programming, Program Transforma tion, and List Operators, Proc. of the 1981 Conf. on Functional Programming Languages and Computer Architecture, pp. 25-32, Oct. 1981
11. J. Tanaka and T. Ida Stream Extension for FP-like Language, Submitted to a technical journal
12. R.B. Kieburtz and J. Shultis Transformation of FP Programs Schemes, Proc. of the 1981 Conf. on Functional Programming Languages and Computer Architecture, pp. 41-48, Oct. 1981