Streams

Roger L. Costello
September 2010

Introduction

Streams have the potential to take XML to the next level by facilitating the processing of huge XML documents.

A stream is a sequence of data. For example, a series of bank deposits and withdrawals is a stream. The sequence of integers between 10,000 and 100,000,000 is a stream. There may even be infinite streams.

What distinguishes a stream from an ordinary sequence is *delayed evaluation*. The idea is this: construct a stream only partially and pass the partial construction to the program that consumes the stream. If the consumer attempts to access a part of the stream that has not yet been constructed, the stream will automatically construct just enough more to produce the required part, thus preserving the illusion that the entire stream exists.

Example #1

An XML document records an individual's bank deposits and withdrawals for a period of time:

```
<transactions>
    <withdrawal>30</withdrawal>
    <deposit>40</deposit>
    <deposit>50</deposit>
    <withdrawal>20</withdrawal>
    <withdrawal>45</withdrawal>
    <deposit>80</deposit>
    ...
</transactions>
```

Start at the first transaction:

```
<withdrawal>30</withdrawal>
```

The following sibling transactions are provided only as the consuming program needs them, i.e., the transactions are provided on demand.

Suppose the start balance is 100. The consuming program asks for the balance after the first transaction. Here it is:

```
    70
```

The consuming program then asks for the balance after the next transaction. This results in providing the next transaction (i.e., visiting the following sibling) and processing it. Here is the new balance:

```
   110
```

And so forth.

The transactions are provided as the consuming program requests them. This on-demand processing is well-suited to a SAX-style XML processor.

Example #2

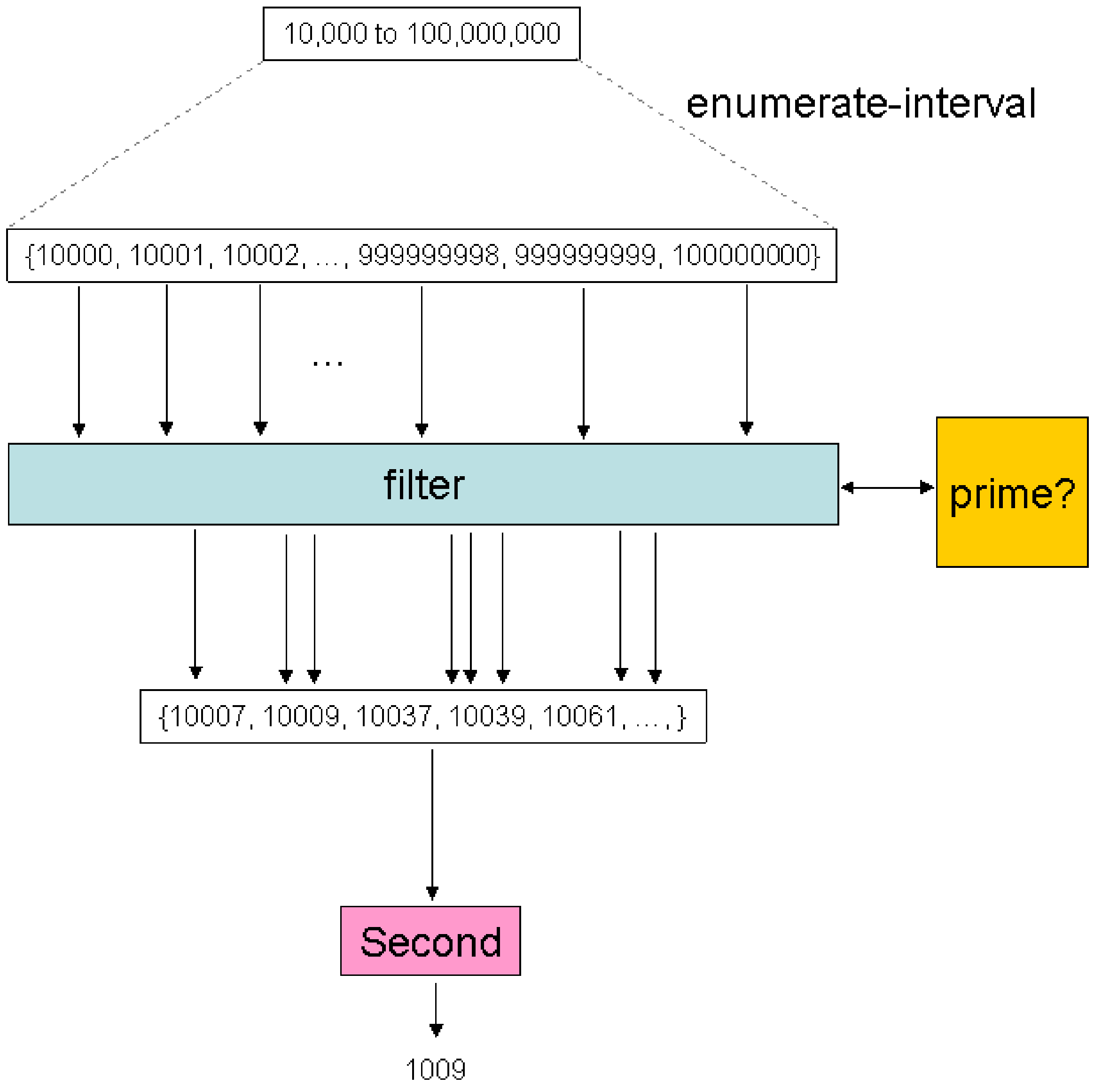
What is the second prime number in the interval from 10,000 to 100,000,000?

Let's consider three approaches to solving this question.

Approach #1: Use Sequence Manipulations

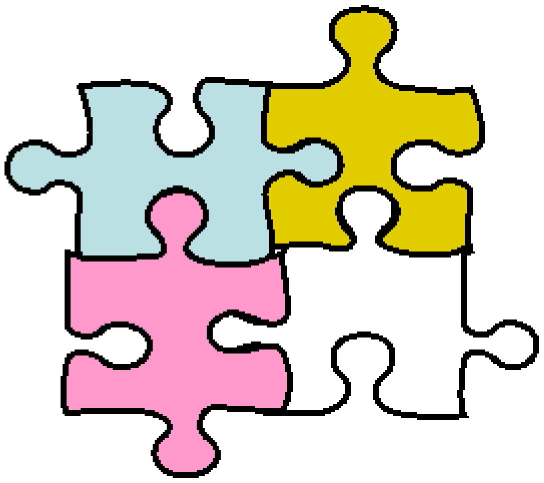
Enumerate all the values in the interval. Filter them using a prime test. This leaves only the prime values. Select the second prime value.

Here's a graphic that illustrates the process:



Advantage

This approach is elegant. It solves the problem by assembling four reusable sequence manipulating components:



That's nice!

Disadvantage

The computational overhead is outrageous. We construct a sequence of almost a hundred million integers, filter this sequence by testing each item for primality, and then ignore almost all of the result.

Approach #2: Use Incremental Computation

Starting at 10,000 test each number for primality and stop at the second prime number. Here is pseudocode for this:

```
NumPrimes = 0
Number = 10000
while (Number < 100000000) {
  if (prime(Number)) then NumberPrimes++
  if (NumPrimes == 2) then exit
}
output(Number)
```

Notice the use of assignment statements to keep a record of the number of primes encountered, i.e., this is a stateful solution.

Advantage

This approach is very efficient computationally.

Disadvantage

The program is an arguably ad hoc set of operations. It doesn't use a set of building blocks, manipulated with a uniform set of operations.

Approach #3: Use Streams

Streams allow the use sequence manipulations (i.e., building blocks) without incurring the cost of manipulating sequences in their entirety. With streams we can achieve the best of both worlds: We can formulate programs elegantly as sequence manipulations, while attaining the efficiency of incremental computation.

A stream implementation automatically and transparently interleaves the construction of the stream with its use. Here’s how:

Consider this enumerate-interval function:

```
enumerate-interval(10000, 100000000)
```

It will generate all the integers from 10,000 to 100,000,000.

The stream version of enumerate-interval just returns the first integer (10,000) and a *promise to compute the rest if requested*. This:

```
stream-enumerate-interval(10000, 100000000)
```

generates this:

`{10000, `*`promise to compute 10001 to 100000000 if requested`*`}`

The first value (10000) is checked for primality. It is not prime so it is discarded. The next number is requested, which generates

`{10001, `*`promise to compute 10002 to 100000000 if requested`*`}`

The first value (10001) is checked for primality. It is not prime so it is discarded.

And so forth.

Since only the numbers that are actually needed are produced, it is efficient like the incremental computation approach. And it has elegance because the stream manipulating components can be assembled just like puzzle pieces or Lego blocks.

At the heart of a stream implementation is delayed evaluation. You can think of delayed evaluation as "demand-driven" programming, whereby each stage in the stream process is activated only enough to satisfy the next stage. We write procedures as if the streams existed "all at once" when, in reality, the computation is performed incrementally.

Recall the bank transaction example above. We proceed in the same fashion as the `stream-enumerate-interval` example. However, rather than using a `stream-enumerate-interval` component, we use a `stream-enumerate-following-siblings` component. It returns the first transaction (i.e., a `<withdraw>` element or `<deposit>` element) and a promise to provide the following transactions if requested.

Streams are a Rich Subject

I have just barely touched on the subject. There is much more to know about streams. Here is a sampling from the "wizard book" (see the Reference section below for details on this book):

Infinite Streams: Above I described how to support the illusion of streams of complete data even though, in actuality, we compute only as much of the stream as we need to access. This technique can be exploited to represent sequences efficiently as streams, even if the sequences are very long. What is more striking, we can use streams to represent sequences that are infinitely long.

New Module Boundaries: The stream approach can be illuminating because it allows systems to be built with different boundaries than systems organized around assignment or state variables. For example, we can think of an entire time series (or signal) as a focus of interest, rather than the values of the state variables at different moments.

Stateless Programming: One of the major benefits of the assignment statement is that we can increase the modularity of our systems by encapsulating, or "hiding," parts of the state of a large system within local variables. However, there are drawbacks inherent in introducing assignment (such as the thorny problems of constraining the order of events and of synchronizing multiple processes). Stream models can provide an equivalent modularity without the use of assignment. Stream functions are stateless.

Acceleration Techniques: We can transform a stream with a *sequence accelerator* that converts a sequence of approximations to a new sequence that converges to the same value as the original, only faster. Although these acceleration techniques could be implemented without using streams, the stream formulation is particularly elegant and convenient because the entire sequence of states is available to us as a data structure that can be manipulated with a uniform set of operations.

Streams of Streams: We can merge two streams or *n* streams. We can create a stream of streams (i.e., a *tableau*) in which each stream is the transform of the preceding one.

Implementing Streams

XSLT 2.0 has sequences. This makes it particularly well-suited to implement streams.

Here's how to implement the `stream-enumerate-interval` function:

I defined a stream namespace:

```
xmlns:stream="http://www.data-structures.org/stream/"
```

I created a function `enumerate-interval` in the stream namespace:

```
<xsl:function name="stream:enumerate-interval" as="item()*">
       <xsl:param name="low" as="xs:integer"/>
       <xsl:param name="high" as="xs:integer"/>

         ...
</xsl:function>
```

It is invoked with two integers, like this:

```
stream:enumerate-interval(10000, 100000000)
```

This stream is produced:

```
(10000, <stream:delay/>, <stream:enumerate-interval/>, 10001, 100000000)
```

`10000` is the first value of the interval. It is obtained using the `stream:first` function:

```
stream:first(stream:enumerate-interval(10000, 100000000))
```

Returns:

```
10000
```

The `<stream:delay>` element indicates that the function has delayed computing the rest of the interval but it *promises to compute the rest of the interval if requested*.

The `<stream:enumerate-interval/>` element identifies the function to be invoked, should there be a need to compute the rest of the interval.

`10001` is the start of the rest of the interval. `100000000` is the end of the interval.

The function, `stream:rest`, returns the rest of the stream. For example:

```
stream:rest(stream:enumerate-interval(10000, 100000000))
```

Returns:

```
(10001, <stream:delay/>, <stream:enumerate-interval/>, 10002, 100000000)
```

Thus, the second value in the interval is the first value of the rest of the stream:

```
stream:first(
      stream:rest(
            stream:enumerate-interval(10000, 100000000)
      )
)
```

Returns:

```
      10001
```

Recall Example #2: What is the second prime number in the interval from 10,000 to 100,000,000?

Using the functions just described, the problem is elegantly and efficiently solved:

```
stream:first(
      stream:rest(
            stream:rest(
                  stream:filter(
                        $prime,
                        stream:enumerate-interval(10000,
                        100000000)
                  )
            )
      )
)
```

The variable `$prime` identifies a function that, given a number, returns true if it is prime and false otherwise.

The `stream:filter` function takes two arguments: (1) An operator (e.g., `prime`), and a stream. It applies the operator to each value in the stream (of course, it does this "on demand" just like `stream:enumerate-interval`)

Recall Example #1: Process each bank transaction.

This is elegantly and efficiently solved using streams:

```
stream:iterator(
    stream:accumulate(
        $credit-debit,
        100,
        stream:enumerate-following-
        siblings(/transactions/child::*[1])
    )
)
```

The variable `$credit-debit` identifies a function that, given a starting balance and either a `<withdraw>` element or a `<deposit>` element, returns the balance minus the withdrawal amount or plus the deposit amount.

The `stream:accumulate` function takes three arguments: (1) An operator (e.g., `credit-debit`), an initial value (e.g., `100`), and (3) a stream. It applies the operator to each value in the stream, using the initial value as the starting value (of course, it does this "on demand").

The `stream:iterator` function takes one argument: A stream. It recursively requests each value in the stream.

Sample Stream Programs

Generate the odd values in the range 1 to 100:

```
stream:filter($odd, (stream:enumerate-interval(1, 100)))
```

Apply (i.e., map) the absolute value function to the integers in the interval -5 to 5:

```
stream:map($stream-abs, stream:enumerate-interval(-5, 5))
```

What is the hundredth prime number in the range 1 to 1,000,000:

```
stream:ref(
    stream:filter(
        $prime,
        stream:enumerate-interval(1, 1000000)
    ),
    100
)
```

Stream Library

I created a library of functions for manipulating streams.

Here are the stream functions:

stream:empty(*stream***)**: returns true if the stream is empty, false otherwise.

stream:first(*stream***)**: returns the first value in the stream.

stream:rest(*stream***)**: recall that a stream contains a first value and a *promise to evaluate the rest if requested*. Invoke this function when you want to cash in on that promise. That is, this function evaluates the rest of the stream. Actually, it returns the first value in the rest and a *promise to evaluate the rest if requested.*

stream:ref(*stream, i***)**: returns the i'th value in the stream. Example: if the stream is the interval 1 to 10 then stream:ref(stream, 2) returns 2.

stream:iterator(*stream, N***)**: returns the first N values in the stream.

stream:iterator(*stream***)**: returns all the values in the stream (it keeps recursively processing the stream until it reaches an empty stream).

stream:map(*proc, stream***)**: applies (i.e., maps) the procedure, proc, to each value in the stream. Actually, if applies proc to the first value in the stream and returns it along with a *promise to apply proc to the rest if requested*.

stream:enumerate-interval(*low, high***)**: returns the integers from low to high. Actually, it returns the first integer and a *promise to enumerate the rest if requested*.

stream:enumerate-following-siblings(*node***)**: returns the (element) node and all its following siblings. Actually, it returns the first node and a *promise to provide the rest if requested*.

stream:filter(*predicate, stream***)**: this applies predicate to each value in the stream. Actually, it applies the predicate to the first value in the stream and a *promise to apply the predicate to the rest if requested*. Example: suppose the stream is 1 to 10 and the predicate is odd (i.e., is the number an odd number?). Then stream:filter(odd, stream) returns 1, 3, 5, 7, 9.

stream:accumulate(*op, initial, stream***)**: this applies the operation, op, to the stream using "initial" as the starting value. Actually, it applies op to the first value in the stream and a *promise to apply it to the rest if requested*. For example, suppose op is "multiply," the initial value is 1, and the stream is the integers from 1 to 10. Then stream:accumulate(multiply, 1, stream) produces this stream: 1 * 2 * 3 * 4 * ... * 10.

stream:abs(*stream***)**: this returns the absolute value of each value in the stream. Actually, it returns the absolute value of the first value in the stream and a *promise to return the absolute value of the rest if requested*.

stream:credit-debit(*balance, stream***)**: this is used exclusively with an XML document containing a sequence of sibling <withdraw> and <deposit> elements. It goes through each node in the sequence and credits/debits the balance. Actually, it credits/debits the first node along with a *promise to do the rest if requested*.

Here's a zip file containing the stream library along with a test file:

http://www.xfront.com/stream/stream-library.zip

I hope you'll give it a try. Let me know of any bugs. If you add more functions please send them along.

Reference and Acknowledgement

Everything I know about streams I learned from this fabulous book (a.k.a. the wizard book):

Structure and Interpretation of Computer Programs by Abelson, Sussman, and Sussman.

Thanks Dimitre Novatchev. I used your technique for creating functions that manipulate functions extensively in the implementation of streams.