Week 10: Stream Computation

In the past two weeks, we have introduced assignment and stateful objects.

We have seen how to model state change in real objects by assignments to variables in computation.

That means, time changes in the real world are modelled by time changes in program execution (stretched out or compressed, but the relative order is the same).

Is there another way?

Can we model changing behavior in the real world by immutable functions?

In mathematics: Of course!

A time-changing quantity is simply modelled by a function $f(t)$ with a time parameter t.

The same can be done in computation. Here is the principle:

- Instead of overwriting a variable with lists of successive values, we construct a list of the variable's successive values.
- I.e. $\textbf{\textit{var}}\ x : T$ becomes $\textbf{\textit{val}}\ x : \mathit{Stream}[T]$. Here, streams are list-like sequences of values.

Streams are Delayed Lists

An advantage of the stream model is that we can make use of the higher-order list processing functions.

This often simplifies computation.

Example: The imperative way to compute the sum of all prime numbers in an interval goes as follows.

```
def sumPrimes(lo: int, hi: int): int = {
    var i = lo;
    var acc = 0;
    while (i < hi) {
        if (isPrime(i)) acc = acc + i;
        i = i + 1;
    }
    acc
}
```

The variable *i* “steps through” all values of the interval $[lo .. hi{-}1]$.

A more functional way is to represent the list of values of variable *i* directly as *range*(*lo*, *hi*). Then the function can be rewritten as follows.

```
def sumPrimes(lo: int, hi: int) =
  sum(range(lo, hi) filter isPrime);
```

No contest which program is shorter and clearer!

However, the functional program is also considerably less efficient since it constructs a list of all numbers in the interval, and then another one for the prime numbers.

Even worse from an efficiency point of view is the following example:

To find the second prime number between *1000* and *10000*:

```
range(1000, 10000) filter isPrime at 1
```

Here, the list of all numbers between *1000* and *10000* is constructed.

Most of that list is never inspected!

Delayed Evaluation

However, we can obtain efficient execution for examples like these by a trick:

> Avoid computing the tail of a sequence unless that tail is actually necessary for the computation.

We define a new class for such sequences, which is called *Stream*.

Using Streams

Streams are created using a constant *empty* and a binary constructor *cons*. Both are defined in module *Stream*. **Example:**

```
val xs = Stream.cons(1, Stream.cons(2, Stream.empty))  // the stream (1, 2)

def range(start: int, end: int): Stream[int] =
  if (start ≥ end) empty
  else cons(start, range(start + 1, end));
```

(In fact, *range* is pre-defined as above in module *scala.Stream*.)

Even though *Stream.range* and *List.range* look alike, their execution behavior is completely different:

- *Stream.range* immediately returns with a *Stream* object whose first element is *start*.
- All other elements are computed only when they are demanded by calling the *tail* method (which might be never at all).

Streams can be accessed just as lists, using *isEmpty*, *head* and *tail*. However, there is no pattern matching available on streams.

Example: To print all elements in a stream:

```
print (xs: Stream [a] ): unit =
    if (xs.isEmpty ) ( ) else { System.out.println (xs.head ); print (xs.tail ) }
```

Furthermore, streams support almost all of the methods defined on lists.

Example: To find the 2nd prime number between *1000* and *10000*:

```
Stream.range (1000, 10000 ) filter isPrime at 1
```

Consing and Appending Streams

Streams do not support methods :: and :::, because these require that their right operand be evaluated first.

The infix operator :: is replaced by the constructor function *Stream.cons*. I.e. instead of *x* :: *xs* one writes *Stream.cons*(*x*, *xs*).

The infix operator ::: is replaced by the operator *append*. I.e. instead of *xs* ::: *ys* one writes *xs append ys*. Here, *append* is defined as follows:

```
def append[b >: a](def rest: Stream[b]): Stream[b];
  if (isEmpty) rest
  else Stream.cons(head, tail.append(rest));
```

- Note that *append* takes as argument a stream whose elements can be of a supertype of the elements of the receiver.
- This is expressed by the lower bound >: *a* of the type variable *b*.
- The result is in every case a stream whose elements are the maximum of their operand element types.

Implementation of *Stream*

The implementation of class *Stream* is very similar to the implementation of class *List*. There is an abstract base class *Stream*, defined as follows.

```
trait Stream[+a] extends Seq[a] {
  def isEmpty: boolean;
  def head: a;
  def tail: Stream[a];
  ...
}
```

The other methods in *Stream* are defined in terms of these three methods. They are all modelled after the methods in class *List*:

```
  def length: int;
  def append[b >: a](def rest: Stream[b]): Stream[b];
  def elements: Iterator[a];
  def init: Stream[a];
  def last: a;
  def take(n: int): Stream[a];
```

```
def drop(n: int): Stream[a];
def apply(n: int): a;
def at(n: int): a;
def takeWhile(p: a => Boolean): Stream[a];
def dropWhile(p: a => Boolean): Stream[a];
def map[b](f: a => b): Stream[b];
def foreach(f: a => unit): unit;
def filter(p: a => Boolean): Stream[a];
def forall(p: a => Boolean): Boolean;
def exists(p: a => Boolean): Boolean;
def foldLeft[b](z: b)(f: (b, a) => b): b;
def foldRight[b](z: b)(f: (a, b) => b): b;
def reduceLeft[b >: a](f: (b, b) => b): b;
def reduceRight[b >: a](f: (b, b) => b): b;
def flatMap[b](f: a => Stream[b]): Stream[b];
def reverse: Stream[a];
def copyToArray[b >: a](xs: Array[b], start: int): int;
def zip[b](that: Stream[b]): Stream[Pair[a, b]];
def print: unit;
def toString(): String;
```

Concrete Streams

There are also two concrete implementations of streams, which are produced by *Stream.empty* and *Stream.cons*.

These might be defined as follows (see below for an optimization).

```
object Stream {
  val empty: Stream[All] = new Stream[All] {
    def isEmpty = true;
    def head: All = error("head of empty stream");
    def tail: Stream[All] = error("tail of empty stream");
  }
  def cons[a](hd: a, def tl: Stream[a]) = new Stream[a] {
    def isEmpty = false;
    def head: a = hd;
    def tail: Stream[a] = tl;
  }...
```

- The only important difference between the implementations of *List* and *Stream* concerns the status of *tl*, the second parameter of *Stream.cons*.
- This parameter is prefixed with ***def***; that's why a corresponding argument will not be evaluated when passed to *Stream.cons*.
- Instead such an argument will be evaluated every time *tail* is called on the stream object.

Lazy Evaluation

- The presented implementation has a potential efficiency problem: If *tail* is called many times on a *Stream.cons* object, the expression computing the rest of the stream will be evaluated many times.
- This problem can be avoided by storing the result of the first evaluation of *tail*, and reusing the stored result every time *tail* is called afterwards.

Exercise: Write a new version of *Stream.cons* that implements this improvement.

The Stream Implementation in Action

To see that the stream implementation behaves indeed efficiently, we trace the execution of the example which computes the 2nd prime number between *1000* and *10000*.

(We are back to purely functional programming, so we can use the simple substitution model).

$$Stream.range\,(1000,\ 10000\,).filter\,(isPrime\,).at\,(1\,)$$

$\rightarrow$ *by expansion of* $Stream.range$

$$\begin{array}{l}(\textbf{if}\ \ (1000 \geq 10000\,)\ \ Stream.empty \\ \textbf{else}\ Stream.cons\,(1000,\ Stream.range\,(1000 + 1,\ 10000\,)\,)\,) \\ .filter\,(isPrime\,).at\,(1\,)\end{array}$$

$\rightarrow$ *by evaluation of* if

$$Stream.cons\,(1000,\ Stream.range\,(1000 + 1,\ 10000\,)\,).filter\,(isPrime\,).at\,(1\,)$$

$\rightarrow$ ...

...

$\rightarrow$ *by expansion of* filter,
abbreviating C_1 = *Stream.cons(1000, Stream.range(1000 + 1, 10000))*
(**if** (C_1.*isEmpty*) ***this***
else if (*isPrime*(C_1.*head*)) *Stream.cons*(C_1.*head*, C_1.*tail.filter*(*isPrime*))
else C_1.*tail.filter*(*isPrime*)).*at*(*1*)

$\rightarrow$ *by evaluation of* if

(**if** (*isPrime*(C_1.*head*)) *Stream.cons*(C_1.*head*, C_1.*tail.filter*(*isPrime*))
else C_1.*tail.filter*(*isPrime*)).*at*(*1*)

$\rightarrow$ *by evaluation of* head

(**if** (*isPrime*(*1000*)) *Stream.cons*(C_1.*head*, C_1.*tail.filter*(*isPrime*))
else C_1.*tail.filter*(*isPrime*)).*at*(*1*)

$\rightarrow$ *by evaluation of* isPrime

(**if** (**false**) *Stream.cons*(C_1.*head*, C_1.*tail.filter*(*isPrime*))
else C_1.*tail.filter*(*isPrime*)).*at*(*1*);

$\rightarrow$ *by evaluation of* if

C_1.*tail.filter*(*isPrime*).*at*(*1*);

$\ldots$

$\rightarrow$ *by evaluation of* tail

Stream.range(1001, 10000).filter(isPrime).at(1)

The process continues like that until we have reached

Stream.range(1009, 10000).filter(isPrime).at(1)

$\rightarrow$ *by evaluation of* filter,

abbreviating $C\_2 = Stream.cons(1009, Stream.range(1009 + 1, 10000))$
Stream.cons(1009, C_2*.tail.filter(isPrime)).at(1)*

$\rightarrow$ *by evaluation of* at

C_2*.tail.filter(isPrime)).at(0)*

$\rightarrow$ *by evaluation of* at, *expansion of* C_2

Stream.range(1010, 10000).filter(isPrime).at(0)

The process continues again until:

Stream.range(1013, 10000).filter(isPrime).at(0)

→ *by evaluation of* filter,

abbreviating *C_3 = Stream.cons(1013, Stream.range(1013 + 1, 10000))*
Stream.cons(1013, C_3*.tail.filter(isPrime)).at(0)*

→ *by evaluation of* at

1013

We see that only the necessary amount of list elements have been created.

Infinite Streams

We have seen that all elements of a stream (except the first) are evaluated only if they are needed by the computation.

This makes it also possible to define infinite streams!

For instance, here is the stream consisting of all integers, or all integers starting from some number *n*.

```
def from (n: int): Stream[int] = Stream.cons(n, from(n+1));
val integers = from(0);
```

Using integers, we can also define other infinite streams, such as the stream of multiples of 4:

```
val multiplesOfFour = integers map (x => x * 4)
```

Or the stream consisting of all numbers not divisible by 7:

```
val noSevens = integers filter (x => x % 7 != 0)
```

A more interesting infinite stream is the stream of all prime numbers.

We will construct this stream using the technique of *Sieve of Erathostenes*.

- Start with the integers beginning with *2*, which is the first prime.
- Eliminate all multiples of *2* from the list.
- The first element of the remaining list is *3*, which is prime.
- Eliminate all multiples of *3* from the list.
- Continue in the same way. At each step, the first number in the list is prime, and we eliminate all multiples of it.

This process is defined by the following function.

```
def sieve(s: Stream[int]): Stream[int] =
  Stream.cons(s.head, sieve(s.tail filter (x => x % s.head != 0)));
```

```
val primes = sieve(from(2));
```

Now to find the list of all prime numbers, simply print it (this will take a while (–;).

```
primes.print
```

Iterations as Stream Processes

Streams with delayed evaluation are a powerful modelling tool, since they let us compose modules of a system in ways which are not possible with state variables.

For instance, it is possible to think of an entire time series as a focus of interest, rather than the value of the variable in a particular moment.

In particular, whole iterations of processes can be made into objects that can be manipulated themselves.

An example is the square root function presented in Lecture 1 of this course.

This function computed square roots by applying iteratively the function.

```
def improve(guess: double, x: double) = (guess + x / guess) / 2;
```

We can now compute the infinite stream of square root approximations:

```
def sqrtStream(x: double): Stream[double] = {
    val guesses = Stream.cons(1, guesses map (guess => improve(guess, x)));
    guesses
}
```

Then we have:

```
> sqrtStream print
1
1.5
1.4166666666665
...
```

Another example computes the value of π, based on the alternating series

$$\frac{\pi}{4} \quad = \quad 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots$$

We first take the stream of summands of the series.

```
def piSummands(n: int): Stream[double] =
  Stream.cons(1.0/(n as double), piSummands(n + 2) map (x => -x));
```

We next take the stream of *partial sums* of all summands and multiply by *4*:

```
val piStream = partialSums(piSummands(1)) map (x => x * 4)
```

This gives us a stream of approximations to π, but the convergence rate is slow:

```
> piStream print
4.0, 2.666666666666667, 3.466666666666667, 2.8952380952380956, 3.3396825396825254,
2.9760461760461776, 3.2837384837384844, 3.017071817071817, ...
```

Exercise: The function *partialSums* computes for a given stream *xs* a new stream $s_1, s_2, s_3, \ldots,$ where each s_i is given by:

$$s_i \quad = \quad \sum_{j=1}^{i} xs_j$$

Complete the following implementation of *partialSums*.

```
def partialSums(s: Stream[double]): Stream[double] =
  Stream.cons(s.head, partialSums(s.tail) map ?? );
```

Transforming Streams

- So far, our use of streams was not so much different from updating state variables.
- But the streams let us use some interesting tricks.
- For instance, we can use a *sequence accelerator* that converts a sequence of approximations into one which converges faster.
- One such accelerator, due to the Swiss 18th century mathematician Leonhard Euler, works well with sequences that are partial sums of an alternating series.
- In Euler's accelerator, if the original series has terms s_i, the transformed series t has terms

$$t_i \quad = \quad s_{i+1} - \tfrac{(s_{i+1}-s_i)^2}{s_{i-1}-2s_i+s_{i+1}}$$

Thus, if the original series is a stream of values, its Euler transform is given by:

```
def eulerTransform(s: Stream[double]): Stream[double] = {
    val s0 = s(0);
    val s1 = s(1);
    val s2 = s(2);
    Stream.cons(
        s2 - (s2 - s1) * (s2 - s1) / (s0 - 2 * s1 + s2),
        eulerTransform(s.tail));
}
```

We can demonstrate Euler acceleration with our sequence of approximations to *pi*.

```
> eulerTransform(piStream).print
3.166666666666667, 3.1333333333333337, 3.1452380952380956, 3.13968253968253396,
3.142712842712843, 3.140881340881341, 3.142071817071817, 3.1412548236077646, ...
```

Tableaus

Even better, we can accelerate the accelerated series, and recursively accelerate that and so on.

More precisely, we can create a stream of streams (called a *tableau*), where each stream is the transform of the preceding one:

$$\begin{array}{l}
\textbf{\textit{def}}\ tableau\,(transform : Stream\,[double] \Rightarrow Stream\,[double], \\
\qquad\qquad\qquad s : Stream\,[double]\,) : Stream\,[Stream\,[double]\,] = \\
\quad Stream.cons\,(s,\ tableau\,(transform,\ transform\,(s)\,)\,);
\end{array}$$

For Euler-transform, the tableau has the form:

$$\begin{array}{cccccc}
s_{00} & s_{01} & s_{02} & s_{03} & s_{04} & \ldots \\
 & s_{10} & s_{11} & s_{12} & s_{13} & \ldots \\
 & & s_{20} & s_{21} & s_{22} & \ldots \\
 & & & \ldots & &
\end{array}$$

Diagonalization

Finally, we can form a sequence by taking the first term in each row of the tableau:

```
def accelerate(transform: Stream[double] => Stream[double],
               s: Stream[double]): Stream[double] =
  tableau(transform, s) map (x => x.head);
```

We can demonstrate this kind of "super-acceleration" of the π sequence:

```
> accelerate(eulerTransform, piStream)
4.0, 3.166666666666667, 3.142105263157895, 3.141599357319004,
3.1415927140337785, 3.1415926539752927, 3.141592653591176,
3.141592653589777, 3.141592653589794, ...
```

Of course, one could implement these acceleration techniques without using streams.

But the stream formulation is particularly elegant and concise, because the entire sequence of states is available to us as data at the same time.

Iterators

Streams avoid the construction of unnecessary list elements.

But they still carry the overhead of having to construct lists.

A more lightweight imperative version are *iterators*.

An iterator also represents a sequence of elements which are evaluated on demand.

But rather than constructing this sequence explicitly, it returns successive elements with a function *next*.

The two most important functions defined by an iterator are given here:

```
trait Iterator[+a] {
    abstract def hasNext: boolean;
    abstract def next: a;
}
```

hasNext returns ***true*** iff the iterator has more elements.

next returns the next element of the iterator, and at the same time advances the iterator by one.

Important: *next* for iterators and *head* for streams are not the same: When called repeatedly, *next* returns successive elements of the iterator, whereas *head* always returns the same element.

Iterators are used often in C++ and Java; for instance they form an essential ingredient of the collections library in *java.util*.

Except for *head*, *tail*, and *isEmpty*, which are replaced by *next* and *hasNext*, iterators support essentially the same methods as streams.

In particular, there is *length*, *append*, *take* and *drop*, *map* and *flatMap*, *foreach*, *fold* and *foldRight*, *zip*, etc.

Iterator Subclasses

Here are two subclasses of iterators.

The first class represents the empty iterator which has no elements.

```
final class EmptyIterator[c] extends Iterator[c] {
    def hasNext = false;
    def next: c = error("next on empty iterator");
}
```

The second class represents an iterator which returns all elements of a given list.

```
final class ListIterator[a](xs: List[a]) extends Iterator[a] {
    var cur: List[a] = xs;
    def hasNext: boolean = cur.nonEmpty;
    def next: a = { val x = cur.head; cur = cur.tail; x }
}
```

But subclasses like these are not used that often since most iterators are constructed directly.

Example: Here is the iterator analogue of *from*:

```
def from (n: int): Iterator[int] = new Iterator[int] {
  var cur = n - 1;
  def hasNext = true;
  def next = { cur = cur + 1; cur }
}
```

Using Iterators

Code using higher-order functions on streams can often be converted directly to iterators.

Example: Computing prime numbers using an iterator:

```
from(2) filter isPrime
```

It is also possible to mix iterators and streams:

```
def sieve(it: Iterator[int]): Stream[int] = {
  val head = it.next;
  Stream.cons(head, sieve(it filter (x => x % head != 0)))
}
sieve(from(2))
```

But: With iterators, one needs to be careful that the same iterator is never executed more than once (because the first use will in fact change it!).

In particular, the series acceleration techniques such as the Euler transform cannot be used unchanged, since they access a sequence several times.

Implementation of Iterator Methods

Here are the implementations of some methods in class *Iterator*.

Here, *Iterator*.***this*** indicates the identity of the enclosing *Iterator* object (instead of the object created by the ***new*** template).

```
trait Iterator[a] {
  def append(that: Iterator[a]): Iterator[a] = new Iterator[a] {
    def hasNext = Iterator.this.hasNext || that.hasNext;
    def next = if (Iterator.this.hasNext) Iterator.this.next else that.next;
  }

  def map[b](f: a => b) = new Iterator[b] {
    def hasNext: boolean = Iterator.this.hasNext;
    def next: b = f(Iterator.this.next);
  }
```

Exercise: Implement the method

$$\textbf{\textit{def}}\ \textit{foreach}\,(f\colon\ (a\,)\textit{unit}\,)\colon\ \textit{unit}$$

which executes the given function f for all elements of the iterator.

Exercise: Implement a method

$$\textbf{\textit{def}}\ \textit{dropWhile}\,(p\colon\ (a\,)\textit{boolean}\,)\colon\ \textit{Iterator}\,[a\,]$$

which returns the iterator resulting from the current iterator by removing all leading elements for which the predicate p is true.

Iterators vs Streams

Iterators and streams both represent sequences and their elements are both evaluated on demand.

So, which abstraction is preferable?

In short, iterators are a “poor-man”s version of streams.

Their advantages are:

- They can be faster since no *cons* objects are constructed.
- They can be implemented even in impoverished languages without call-by-need parameters or higher-order functions.

Their disadvantages are:

- Order dependence: one needs to be very careful in what order *next* operations are called.
- No reuse: once an iterator is advanced, the old state is lost; so one cannot step through an iterator more than one time.