Blocking Synchronization: Streams

Vijay Saraswat

(Dec 10, 2012)

1 Streams

Streams provide a very simple abstraction for determinate parallel computation.

The intuition for streams is already present in the Unix "pipes" abstraction. Imagine a graph, with sequential code running at each vertex, and with "pipes" corresponding to its edges. A pipe transmits a (potentially infinite) sequence of values from source to destination, i.e. it can be thought of as a point-to-point, loss-less, order-preserving communication channel. The sequential code at a vertex can perform a *read* operations on incoming pipes, and *write* operations on outgoing pipes. We shall also permit the code to *close* an outgoing pipe.

For now we consider each pipe to have unbounded capacity. This means that the producer for the pipe (the sequential code running at the vertex that is the source of the pipe) may run arbitrarily far ahead of the consumer for the pipe (the sequential code running at the vertex that is the sink of the pipe).

A read operation retrieves the next value from the pipe, blocking until such time as there is a value, or the stream has been closed. (If the stream is closed then the operation will throw a `StreamClosedException`.)

A write operation simply appends the item to the end of the pipe.

The basic abstractions can be specified thus. In the code segments that follow, we have removed various comments and `log` statements. The actual code may be found in the repository.

A `Stream[T]` is a stream of values of type `T`, and it permits a user to get the elements of the stream, waiting until it has one. It throws a `StreamClosedException` if the stream has been closed.

```
1
2 public interface Stream[T] {
3   operator this():T;
4 }
```

A `Source[T]` is a handle to a stream that can be used to push elements on to the stream, and to close the stream.

```
1 public interface Source[T] {
2   operator this()=(t:T):void;
3   def close():void;
4 }
```

A `Spring[T]` is just a `Stream[T]` or a `Source[T]`.

```
1 public interface Spring[T] extends Stream[T], Source[T] {}
```

```
1 public class StreamClosedException(s:Any) extends Exception {}
```

1.1 Implementation of streams

Now let us consider an implementation of `Stream`.

1.1.1 Monitor

First we start with the implementation of a *monitor* abstraction. We show how to work wround the limitation of the current current X10 implementation of `atomic` and `when` (they use a single place-global lock and hence limit concurrency). We directly use `Lock` to implement an atomic block abstraction.

A monitor may be used to ensure atomic execution of conditional code blocks by multiple activities executing simultaneously. There may be multiple monitors in a given place, and they can be operated on simultaneously with no interference across monitors. It is the responsibility of the programmer to determine which activity should use which monitor.

The code is structured along familiar lines. A lock is used to guard operations on monitor. Since a conditional atomic operation may suspend, a suspension mechanism is maintained internally. The field `threads` keeps a list of suspended threads, there can be at most `MAX_THREADS` of these.

```
1 public class Monitor {
2   protected val lock = new Lock();
3   protected val threads =
4     new Rail[Worker](x10.lang.Runtime.MAX_THREADS);
5   protected var size:Int = 0;
6 ...
```

Note that the `lock()` operation increases and decreases parallelism explicitly. This is because the `lock.lock()` call may suspend, blocking the current worker, and preventing it from executing other asyncs. The call `increaseParallelism()` permits the run-time to create another thread (or activate a frozen threads) so that a thread is available to take the place of the thread that is about to be suspended on the lock. On return from the call `lock.lock()`, the invocation of `decreaseParallelism()` ensures that if the worker pool has more workers than it nominally should have, one of the idle workers is frozen.

```
1   protected def lock() {
2     if (! lock.tryLock()) {
3       Runtime.increaseParallelism();
4       lock.lock();
5       Runtime.decreaseParallelism(1);
6     }
7   }
8   protected def unlock() { lock.unlock();}
```

The routine `on[T](cond:()=>Boolean, action:()=>T):T` is the work horse for the monitor. An activity executing this method will block until such time as `cond()` evaluates to `true`. It will then execute `action()`. `cond` should be side-effect free; it may be evaluated an unknown number of times. However, `action()` will be evaluated only once. The last execution of `cond()` and the execution of `action()` are guaranteed to be done in a single step wtih respect to any other `on` operations on this monitor.

```
1  public def on[T](cond:()=>Boolean, action:()=>T):T {
2    try {
3      lock();
4      while (!cond()) {
5        val thisWorker = Runtime.worker();
6        val s = size;
7        threads(size++)=thisWorker;
8        while(threads(s)==thisWorker) {
9          unlock();
10         Worker.park();
11         lock();
12       }
13     }
14     val result=action();
15     val m=size;  // now awaken all suspended workers
16     for (var i:Int = 0; i<m; ++i) {
17       size--;
18       threads(size).unpark();
19       threads(size)=null;
20     }
21     return result;
22   } finally {
23     unlock();
24   }
25 }
```

Using this operation many convenient operations can be defined. `awaken()` will awaken all workers, if any, waiting on this monitor.

```
1  static val TRUE = ()=>true;
2  static val NOTHING = ()=>Unit();
3
4  public def awaken() { on(TRUE, NOTHING); }
5  public def await(cond:()=>Boolean) { on(cond, NOTHING);        }
6  public def atomicBlock[T](action:()=>T):T =on(TRUE, action);
```

1.1.2 Bounded buffer

Using `Monitor` one can build a bounded buffer.

```
1  package pppp.util;
2  import x10.compiler.NonEscaping;
3  import pppp.util.Logger;
4  public class BBuffer[T](N:Int) {
5      protected val data:Rail[T];
6      protected var nextVal:Int=0;
7      protected var size:Int=0;
8      protected val monitor = new Monitor();
9      protected var name:String;
10
11     public def this(N:Int){T haszero}{
12         this(N,Zero.get[T]());
13     }
14     public def this(N:Int, t:T) {
15         property(N);
16         data = new Rail[T](N, t);
```

```
17      }
18  ...
```

Adding an element to the buffer is done as follows. One defines a condition that checks for space, and an operation that adds to the buffer.

```
1   protected def hasSpace():Boolean = size < N;
2   protected def add(t:T):Unit {
3     var nextSlot:Int = nextVal+size;
4     if (nextSlot >= N) nextSlot %=N;
5     data(nextSlot)=t;
6     size++;
7     return Unit();
8   }
9   public operator this()=(t:T):void{
10    monitor.on[Unit](()=> hasSpace(),()=>add(t));
11  }
```

Dually, one defines the operation for getting a value from the buffer using a condition that checks that there is a value, and an operation that actually removes the value.

```
1   protected def awaken() { monitor.awaken(); }
2   protected def hasValue():Boolean = size > 0;
3   protected def get():T {
4     val result = data(nextVal);
5     if (++nextVal >= N) nextVal %= N;
6     size--;
7     return result;
8   }
9   public operator this():T = {
10    val t = monitor.on(()=> hasValue(), ()=>get());
11    t
12  }
```

1.1.3 Stream implementation

Finally we can define an implementation of `Spring[T]`. The `close()` operation schedules an atomic block that unconditionally sets `closed` to be `true`. Adding an element to the stream and obtaining an element from the stream is done by using the inherited apply and assignment operators. The key is that the `hasSpace()` and `hasValue()` conditions are over-ridden to check whether the stream is closed.

```
1
2   public class BoundedStreamImp[T] extends BBuffer[T]
3     implements Spring[T]{
4     public static val DEFAULT_SIZE=2000;
5     protected var closed:Boolean=false;
6     public def this(){T haszero}{
7       this(null, DEFAULT_SIZE, Zero.get[T]());
8     }
9     public def this(s:String){T haszero}{
10      this(s, DEFAULT_SIZE, Zero.get[T]());
11    }
```

```
12    public def this(s:String, N:Int, zero:T) {
13      super(N, zero);
14      this.setName(s);
15    }
16
17    public def close():void {
18      monitor.atomicBlock(()=>{closed=true; Unit()});
19    }
20    public def isOpen():Boolean = !(closed && size==0);
21
22    protected def hasSpace():Boolean {
23      if (closed) throw new StreamClosedException(this);
24      return super.hasSpace();
25    }
26    protected def hasValue():Boolean {
27      if (!isOpen()) throw new StreamClosedException(this);
28      return super.hasValue();
29    }
30  }
```

2 Operators on streams

We can now define different kinds of operators on streams.

`ConstraintSTream` produces an infinite stream containing a pre-specified element.

```
1  public class ConstantStream[T] implements Stream[T] {
2    val k:T;
3    public def this(k:T){this.k=k;}
4    public operator this():T =k;
5    public def toString()= "<ConstantStream " + k + ">";
6  }
```

Now we consider the `FBy` ("followed by") stream. It takes as input a stream `o` and prefixes a given rail of elements before it.

```
1  public class FBy[T](a:Rail[T],b:Stream[T])
2    implements Stream[T]{
3    public def this(a:T)  { this(new Rail[T](1,a));}
4    public def this(a:Rail[T]){ this(a,null as Stream[T]);}
5
6    public def this(x:IntRange){T==Int}{
7      this(new Rail[Int](x.max-x.min+1, (i:Int)=>x.min+i));
8    }
9
10   public def this(a:Rail[T], b:Stream[T]){
11     property(a, b);
12   }
13   public def this(x:T,b:Stream[T]) {
14     this(new Rail[T](1,x),b);
15   }
16
17   var i:Int=0;
18   public operator this():T= {
19     if (i < a.size) {
20       val item = a(i++);
```

```
21        return item;
22      }
23      if (b !=null) {
24        val item = b();
25        return item;
26      }
27      throw new StreamClosedException(this);
28    }
29    @NonEscaping
30      public final def toString()= "<" + a +
31          (b==null ? "" : " -> " + b) + ">";
32  }
```

A `FilteredStream` removes all elements fromt he input stream taht satisfy the given condition.

```
1  public class FilteredStream[T](f:(T)=>Boolean, aS:Stream[T])
2    implements Stream[T] {
3    public operator this():T  {
4        var a:T=aS();
5        while (! f(x)) a=aS(); // may throw StreamClosedException
6        return a;
7      }
8    public def toString()="<" + aS + " filteredBy " + f +">";
9  }
```

An `OpStream1` applies a unary function on the input stream:

```
1  public class OpStream1[S,T](f:(S)=>T, aS:Stream[S])
2    implements Stream[T] {
3    public operator this():T =f(aS());
4    public def toString()="<" + f + "(" + aS + ")>";
5  }
```

Similarly one can define `OpStream2`.

The class `XDucer` provides certain useful *transducers* on streams. A transducer takes as input one or more streams and produces as output zero or more streams.

The `copy` transducer copies elements from the input stream to each one of a given rail of streams:

```
1    public static def copy[T,X](source:Stream[T], sinks:Rail[X])
2     {X <: Source[T]}{
3      async
4        try {
5          while (true) {
6            val x = source();
7            for (o in sinks.values()) o()=x;
8          }
9        } catch (z: StreamClosedException) {
10       } finally {
11         for (o in sinks.values()) o.close();
12       }
13   }
```

The `print` transducer prints out the given stream, `k` elements at a time, on the given `Printer`.

```
1  public static def print[T](ix:Stream[T]):void {
2    print(Console.OUT, ix, 10);
3  }
4  public static def print[T](ix:Stream[T], k:Int):void {
5    print(Console.OUT, ix, k);
6  }
7  public static def print[T](p:Printer,ix:Stream[T],k:Int):void {
8    async {
9      try {
10       var n:Int=0;
11       while(true) {
12         n++;
13         if (n % k == 0) p.println(ix());
14         else p.print(ix() + " ");
15       }
16     } catch (z: StreamClosedException) {
17       Logger.log("print catches exception. Terminates.");
18     }
19     p.println();
20   }
21 }
```

3 Examples of stream programs

3.1 Twice

Here is a simple program that takes as input a stream of integers, and doubles it. It explicitly constructs a codeBoundedStreamImp.

```
1  public class SimpleStreamExample {
2    public static def main(s:Rail[String]) {
3      val N = s.size > 0 ? Int.parseInt(s(0)) : 100;
4      val time = System.nanoTime();
5      finish {
6        XDucer.print(twice(gen(N)), 10);
7      }
8      Logger.info(()=>"Time: "
9        + ((System.nanoTime()-time)*1.0)/(1000*1000*1000)
10       + " s.");
11   }
12
13   static def gen(N:Int):Stream[Int] = new FBy[Int](2..N);
14   static def twice(nums:Stream[Int]):Stream[Int] {
15     val s = new BoundedStreamImp[Int]();
16     async {
17       try {
18         while (true) {
19           val item = nums();
20           s()=2*item;
21         }
22       } catch (StreamClosedException) {
23         s.close();
24       }
25     }
```

```
26      return s;
27    }
28  }
```

The same program can be written using an `OpStream1` instance.

```
1   static def twice(aS:Stream[Int]):Stream[Int]
2     =new OpStream1((x:Int)=>2*x,aS);
```

3.2 Hamming

The Hamming sequence is an ordered sequence containing precisely the numbers $2^i \times 3^j \times 5^k$ for $i, j, k \geq 0$. Thus the sequence starts out as

```
1  1 2 3 4 5 6 8 9 10 ...
```

This code can be written in a simple fashion, recognizing that the stream can be defined recursively.

```
1   public class Hamming {
2     static def omerge(n_:Int, aS:Stream[Int],
3       bS:Stream[Int]):Stream[Int] {
4       val s  = new BoundedStreamImp[Int]("omerge(" + aS
5         + "," + bS+")");
6       async {
7         var a:Int=aS(), b:Int=bS();
8         try {
9           var n:Int=n_;
10          while (n-- >0) {
11            val item = Utils.min(a,b);
12            s()=item; // output before consuming
13            if (a==item) a=aS();
14            if (b==item) b=bS();
15          }
16        } catch (StreamClosedException) {
17        } finally {
18          s.close();}
19        }
20      return s;
21    }
22
23    static def kmult(k:Int, aS:Stream[Int]):Stream[Int] =
24       new OpStream1[Int,Int]((x:Int)=>x*k, aS);
25
26    static def hamming(n:Int):Stream[Int] {
27      val hx:Rail[Spring[Int]] =
28        new Rail[Spring[Int]](4, (i:Int)=>
29              new BoundedStreamImp[Int]("_hamming(" + i+")"));
30      XDucer.copy(new FBy[Int](1,
31        omerge(n-1, kmult(2,hx(0)),
32          omerge(n-1, kmult(3,hx(1)),
33            kmult(5,hx(2))))),
34        hx);
35      return hx(3);
36    }
37
```

```
38    public static def main(args:Array[String](1)) {
39      if (args.size < 1) {
40        Console.ERR.print("Usage: Hamming <N>\n");
41        return;
42      }
43      finish {
44        Logger.log(()=> "Starting hamming.");
45        XDucer.print(hamming(Int.parseInt(args(0))));
46      }
47      Console.OUT.println("...done.");
48    }
49  }
```

3.2.1 The Sieve of Eratosthenes

The Sieve of Eratosthenes is a way of generating the sequence of prime numbers. It takes as input a stream of increasing numbers. It declares the first number it sees as a prime, and then filters out (from the remainder of the stream) all numbers that are multiples of this number, and then passes this resulting stream into another recursive instance of itself. Thus it (lazily) sets up a cascading chain of filters, one for each prime.

The initial input to the sieve is the sequence of integers starting from 2.

```
1   public class Sieve {
2     public static def main(s:Rail[String]) {
3       val N = s.size > 0 ? Int.parseInt(s(0)) : 100;
4       val time = System.nanoTime();
5       finish {
6         XDucer.print(primes(gen(N)), 10);
7       }
8       Console.OUT.println("Time: "
9         + ((System.nanoTime()-time)*1.0)/(1000*1000*1000)
10        + " s.");
11    }
12
13    static def gen(N:Int):Stream[Int] = new FBy[Int](2..N);
14    static def primes(nums:Stream[Int]):Stream[Int] {
15      try {
16        val prime = nums();
17        val s = new FByPush[Int](prime, ()=> primes(sieve(prime, nums)),
18          "prime(" + prime+")");
19        s.run();
20        return s;
21      } catch (z: StreamClosedException) {
22        Logger.debug(()=>"Primes(" + nums + ") throws exception.");
23        throw z;
24      }
25    }
26
27    static def sieve(prime:Int, nums:Stream[Int]):Stream[Int] {
28      val s = new BoundedStreamImp[Int]();
29      async {
30        Logger.log(()=> "Starting sieve(" + prime + ")");
31        try {
32          while (true) {
```

```
33            val item = nums();
34            if (item % prime != 0) s()=item;
35          }
36        } catch (StreamClosedException) {
37          s.close();
38        }
39      }
40      return s;
41    }
42  }
```

Exercise 3.1 *Can* `sieve` *be written using one of the operations on streams we have defined earlier?*

4 Kahn networks

Kahn networks are model of concurrent computation obtained by connecting sequential agents through Unix-style first-in first-out pipes [Kah74].

A Kahn networks consists of a collection of nodes connected to other nodes through streams. Each node is associated with an agent. The agent can read one or more values from one or more sinks, perform an arbitrary sequential computation and write one or more values on one or more sources.

Typically agents have bounded state. If an agent must operate on unbounded state it may dynamically spawn a whole new subnetwork at any time. That is, Kahn networks are not static. Indeed, a Kahn network may grow unboundedly over time.

The key restriction in Kahn networks is that an agent is sequential – it may do only one thing at a time. In particular, if it is waiting for input to arrive on a particular sink, it cannot simultaneously be waiting for input to arrive on another stream. That is one cannot program a *parallel or* function:

```
1  /**
2   Send true on the output as soon as true arrives on either sink,
3   without waiting for a value to arrive on the other sink.
4  */
5  def por(a:Sink[Boolean], b:Sink[Boolean]):Sink[Boolean];
```

The fundamental result in Kahn's seminal paper is that Kahn networks are determinate. Each agent can be modeled as a function from the sequence of input streams to the sequence of output streams.

References

[Kah74] Gilles Kahn. The Semantics of Simple Language for Parallel Programming. In *IFIP Congress*, pages 471–475, 1974.