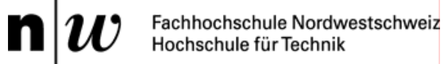
Infinite Streams in Java

Dominik Gruntz
University of Applied Sciences Northwestern Switzerland
Institute for Mobile and Distributed Systems



Infinite Streams in Java

- **Infinite Streams**
  - Conceptually contain infinitely many terms
  - Arbitrary many terms can be accessed on demand
  - fibs = 0, 1, 1, 2, 3, 5 , 34,
- **Ingredients**
  - Closures
  - Lazy Evaluation / strict functions
  - Fixed point definitions

Example: Streams in Haskell

- **Lazy Evaluation & non-strict functions**

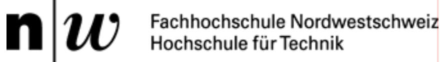
```
> integersFrom :: Integer -> [Integer]
> integersFrom n = n : integersFrom (n+1)
```

- **Mapping functions on a stream**

```
> map  (^2) (integersFrom 0)
```

- **Recursive Definitions**

```
> int = 0 : (map succ int)
```

Outline

1. **Definition of Streams in Java**
   - *Lazy Evaluation in Java*
2. **Stream Interface**
   - *Functors in Java*
3. **Fixed Point Definitions**
   - *Fixed Points in Java*
4. **Application: Power Series**
5. **Groovy as User Interface**
6. **Lessons Learned**

Linked Stream Implementation

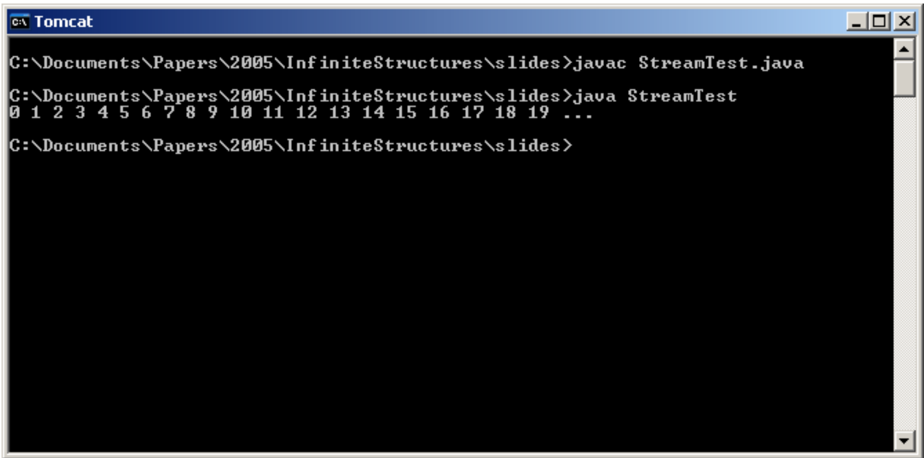
Stream
getHead()
getTail()
head
Object
LazyTail
eval()

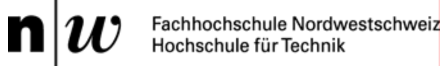
```
public class Stream {

    public interface LazyTail { Stream eval(); }

    private Object head;
    private LazyTail tail;

    public Stream(Object head, LazyTail tail) {
        this.head = head;
        this.tail = tail;
    }

    public Object getHead(){ return head; }
    public Stream getTail(){ return tail.eval(); }
}
```


Linked Stream Implementation

```
public class StreamTest {

    static Stream integersFrom(final int n){
        return new Stream(n,
            new Stream.LazyTail(){
                public Stream eval(){return integersFrom(n+1);}
            }
        );
    }

    public static void main(String[] args) {
        Stream s = integersFrom(0);
        for(int i=0; i<20; i++, s = s.getTail()){
            System.out.print(s.getHead() + " ");
        }
       System.out.println("...");
    }
}
```

Linked Stream Implementation

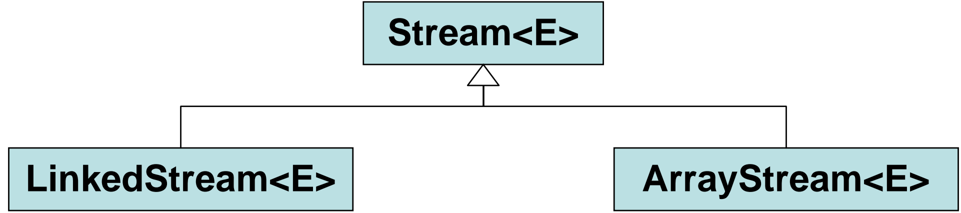




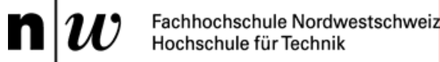
Interface Stream<E>

```
public interface Stream<E> {
   interface Selector<T> { boolean select(T x); }
   interface UnaryFunctor <T,R> { R eval(T x); }
   interface BinaryFunctor<T1,T2,R> { R eval(T1 x, T2 y); }

   E getHead();
   E get(int index);
   Stream<E> getTail();

   java.util.Iterator<E> iterator();
   String toString(int order);

   Stream<E> prepend(E x);  // [x : this]
   Stream<E> select(Selector<? super E> s);
   <R>   Stream<R> map(UnaryFunctor <? super E, R> f);
   <T,R> Stream<R> zip(BinaryFunctor<? super E, ? super T, R> f,
                                Stream<T> stream2);
}
```

Interface Implementations



- **LinkedStream**
  - Head & Tail representation
    - Tail is lazily evaluated
  - Access: O(n)
  - Corresponds to LinkedList
- **ArrayStream**
  - Map representation
    - index -> element
    - Terms are lazily evaluated
  - Access: O(1)
  - Corresponds to ArrayList



Class LinkedStream<E> (1/2)

```
public class LinkedStream<E> implements Stream<E> {

   public interface LazyTail<E> { LinkedStream<E> eval(); }

   private E head;
   private LinkedStream<E> tail; //assigned on demand
   private LazyTail<E> lazyTail; //delayed tail rule
   public LinkedStream(E head, LazyTail<E> tail) {
      this.head = head; this.lazyTail = tail;
   }

   public E getHead(){ return head; }
   public LinkedStream<E> getTail(){
      if(tail==null){tail = lazyTail.eval(); lazyTail = null;}
      return tail;
   }
   ...
```

Class LinkedStream<E> (2/2)

- **get**

```
public E get(int n) {
   if (n < 0) throw new IndexOutOfBoundsException( );
   Stream<E> stream = this;
   while (n > 0) { stream = stream.getTail(); n--; }
   return stream.getHead();
}
```

- **map**

```
public <R> LinkedStream<R> map(
                        final UnaryFunctor<? super E, R> f) {
   return new LinkedStream<R>(
      f.eval(getHead()),
      new LazyTail<R>(){
         public LinkedStream<R> eval(){return getTail().map(f);}
      }
   );
}
```


Class ArrayStream<E> (1/2)

```
public class ArrayStream<E> implements Stream<E> {

   public interface Coefficients<E> { E get(int index); }

   private Coefficients<E> lazyCoeff;
   private HashMap<Integer,E> coeff = new HashMap<Integer,E>();

   public ArrayStream(Coefficients<E> c){ this.lazyCoeff = c; }

   public E get(int index){
      if (index < 0) throw new IndexOutOfBoundsException);
      Integer n = index;
      if(!coeff.containsKey(n))
         coeff.put(n,lazyCoeff.get(index));
      return coeff.get(n);
   }
```

Class ArrayStream<E> (2/2)

- **prepend**

```
public Stream<E> prepend(final E head){
   return new ArrayStream<E>(
      new Coefficients<E>(){
         public E get(int index){
            if(index == 0) return head;
            else return ArrayStream.this.get(index-1);
         }
      }
   );
}
```


Fixed Point Definitions of Streams

- **Recursive Definitions**
  - Stream definitions often depend on itself
  - Streams can be defined as fixed point of a map on streams

- **Example**
  - fibs → 0 : (+ fibs (1 : fibs))

  - fibs = 0 … 0 : (+ [0 …] [1 …])
  - fibs = 0 1 … 0 : (+ [0 1 …] [1 0 …])
  - fibs = 0 1 1 … 0 : (+ [0 1 1 …] [1 0 1 …])
  - fibs = 0 1 1 2 … 0 : (+ [0 1 1 2 …] [1 0 1 1 …])
  - fibs = 0 1 1 2 3 … 0 : (+ [0 1 1 2 3 …] [1 0 1 1 2 …])
  - fibs = 0 1 1 2 3 5 … 0 : (+ [0 1 1 2 3 5 …] [1 0 1 1 2 3 …])

Fixed Point Definitions in Java

- **Functor to define maps on streams**

```
interface StreamMap<T> {
   Stream<T> map(Stream<T> stream);
}
```

- **Factory method to create streams as fixed points of a map**

```
static <T> Stream<T> fixedPoint(StreamMap<T> map){
   ...
}
```


Fixed Point Definitions in Java

- **ones = 1 : ones**

```
Stream<Integer> ones = fixedPoint(
   new StreamMap<Integer>(){
      public Stream<Integer> map(Stream<Integer> s){
         return s.prepend(1);
      }
   });
```

- **integers = 0 : (map succ integers)**

```
Stream<Integer> integers = fixedPoint(
   new StreamMap<Integer>(){
      public Stream<Integer> map(Stream<Integer> s){
         return s.map(
            new UnaryFunctor<Integer, Integer>(){
               public Integer eval(Integer x){return x+1;}
            }
         ).prepend(0);
      }
   });
```

Fixed Point Definitions in Java

- **fibs = 0 : (+ fibs (1 : fibs))**

```
class Add implements
            Stream.BinaryFunctor<Integer, Integer, Integer> {
   public Integer eval(Integer x, Integer y){ return x+y; }
};

Stream<Integer> fibonacci = fixedPoint(
   new StreamMap<Integer>() {
      public Stream<Integer> map(Stream<Integer> f){
         return f.zip(new Add(), f.prepend(1)).prepend(0);
      }
   }
);
```


Fixed Point Definitions in Java

- **fibs = 0 : (+ fibs (1 : fibs))**

```
Stream<Integer> fibonacci = fixedPoint(
   new StreamMap<Integer>() {
      public Stream<Integer> map(Stream<Integer> f){
         return f.zip(
            new Stream.BinaryFunctor<Integer,Integer,Integer>()
            {
               public Integer eval(Integer x, Integer y){
                  return x+y;
               }
            },
            f.prepend(1)).prepend(0);
      }
   }
);
```

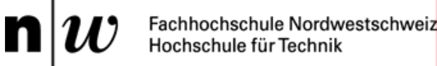
Fixed Point Implementation

- **LinkedStream**

```
public static <T> Stream<T> fixedPoint(StreamMap<T> map){
   LinkedStream<T> s1 = new LinkedStream<T>();
   LinkedStream<T> s2 = (LinkedStream<T>)map.map(s1);
   s1.head     = s2.head;
   s1.lazyTail = s2.lazyTail;
   return s2;
}
```

- **ArrayStream**

```
public static <T> Stream<T> fixedPoint(StreamMap<T> map){
   ArrayStream<T> s1 = new ArrayStream<T>();
   ArrayStream<T> s2 = (ArrayStream<T>)map.map(s1);
   s1.lazyCoeff = s2.lazyCoeff;
   return s2;
}
```

Fixed Point Implementation

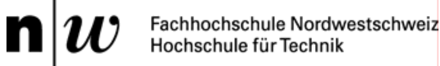
```
public Stream<Integer> map(Stream<Integer> s){
   return s.map(
      new UnaryFunctor<Integer, Integer>(){
         public Integer eval(Integer x){return x+1;}
      }
   ).prepend(0);
}
```

- **Issues**
  - With LinkedList, the head element of s is accessed in s.map
    => head element in LinkedList has to be evaluated lazily as well in LinkedList implementation
  - Map must not create new (concrete) instances of streams but rather restrict on calling methods of the Stream interface

Infinite PowerSeries

$$x - \frac{1}{6}x^3 + \frac{1}{120}x^5 - \frac{1}{5040}x^7 + \frac{1}{362880}x^9 + O(x^{10})$$

```
public interface PowerSeries {

   public Rational coeff(int index);

   public PowerSeries add(PowerSeries value);
   public PowerSeries subtract(PowerSeries value);
   public PowerSeries negate();
   public PowerSeries multiply(Rational value);
   public PowerSeries multiply(PowerSeries value);
   public PowerSeries divide(PowerSeries value);

   public PowerSeries diff();
   public PowerSeries integrate();
   public PowerSeries integrate(Rational c);
   public PowerSeries shift(Rational c);
}
```

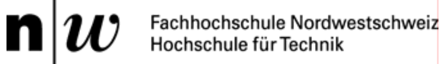



Infinite PowerSeries

```
public class PowerSeries {
   private ArrayStream<Rational> coefficients;

   public PowerSeries(ArrayStream<Rational> coefficients){
      this.coefficients = coefficients;
   }

   public PowerSeries diff(){ // derivative
      return new PowerSeries(
         new ArrayStream<Rational>(
            new ArrayStream.Coefficients<Rational>(){
               public Rational get(int k){
                  return coeff(k+1).multiply(new Rational(k+1));
               }
            }
         )
      );
   }
```

Infinite PowerSeries

```
interface PowerSeriesMap {
   PowerSeries map(PowerSeries series);
}

public PowerSeries fixedpoint(PowerSeriesMap map){
   PowerSeries s1 = new PowerSeries();
   PowerSeries s2 = map.map(s1);
   s1.coefficients = s2.coefficients;
   return s2;
}
```

Groovy Interface

- **Groovy Scripting Language (JSR 241)**
  - Easy notion for closures
  - Java classes can directly be called
  - Special Groovy classes for streams and power series
    - Groovy operators are implemented
    - Groovy closures are interpreted
- **Demo**

Lessons Learned

- **Lazy Evaluation / Functors in Java**
  - Java 5 (generics) simplified notation of Functors & lazy evaluation
  - Notation is still “ugly”
  - Wish: Simple notation for functors

- **Groovy**
  - Groovy is really useful as a scripting environment