



香港科技大學

THE HONG KONG UNIVERSITY OF
SCIENCE AND TECHNOLOGY

Principles of Programming Languages (Lecture 11)

COMP 3031, Fall 2025

Lionel Parreaux

Lazy Lists

Collections and Combinatorial Search

We've seen a number of immutable collections that provide powerful operations, in particular for combinatorial search.

For instance, to find the second prime number between 1000 and 10000:

```
(1000 to 10000).filter(isPrime)(1)
```

This is *much* shorter than the recursive alternative:

```
def secondPrime(from: Int, to: Int) = nthPrime(from, to, 2)
def nthPrime(from: Int, to: Int, n: Int): Int =
  if from >= to then throw Error("no prime")
  else if isPrime(from) then
    if n == 1 then from else nthPrime(from + 1, to, n - 1)
  else nthPrime(from + 1, to, n)
```

Performance Problem

But from a standpoint of performance,

```
(1000 to 10000).filter(isPrime)(1)
```

is pretty bad; it constructs *all* prime numbers between 1000 and 10000 in a list, but only ever looks at the first two elements of that list.

Reducing the upper bound would speed things up, but risks that we miss the second prime number all together.

Delayed Evaluation

However, we can make the short-code efficient by using a trick:

Avoid computing the elements of a sequence until they are needed for the evaluation result (which might be never)

This idea is implemented in a new class, the LazyList.

Lazy lists are similar to lists, but their elements are evaluated only *on demand*.

Defining Lazy Lists

Lazy lists are defined from a constant `LazyList.empty` and a constructor `LazyList.cons`.

For instance,

```
val xs = LazyList.cons(1, LazyList.cons(2, LazyList.empty))
```

They can also be defined like the other collections by using the object `LazyList` as a factory.

```
LazyList(1, 2, 3)
```

The `to(LazyList)` method on a collection will turn the collection into a lazy list:

```
(1 to 1000).to(LazyList)    > res0: LazyList[Int] = LazyList(<not computed>)
```

LazyList Ranges

Let's try to write a function that returns `(lo until hi).to(LazyList)` directly:

```
def lazyRange(lo: Int, hi: Int): LazyList[Int] =  
  if lo >= hi then LazyList.empty  
  else LazyList.cons(lo, lazyRange(lo + 1, hi))
```

Compare to the same function that produces a list:

```
def listRange(lo: Int, hi: Int): List[Int] =  
  if lo >= hi then Nil  
  else lo :: listRange(lo + 1, hi)
```

Comparing the Two Range Functions

The functions have almost identical structure yet they evaluate quite differently.

- ▶ `listRange(start, end)` will produce a list with `end - start` elements and return it.
- ▶ `lazyRange(start, end)` returns a single object of type `LazyList`.
- ▶ The elements are only computed when they are needed, where “needed” means that someone calls `head` or `tail` on the lazy list.

Methods on Lazy Lists

LazyList supports almost all methods of List.

For instance, to find the second prime number between 1000 and 10000:

```
LazyList.range(1000, 10000).filter(isPrime)(1)
```

LazyList Cons Operator

The one major exception is `::`.

`x :: xs` always produces a list, never a lazy list.

There is however an alternative operator `#::` which produces a lazy list.

```
x #:: xs == LazyList.cons(x, xs)
```

`#::` can be used in expressions as well as patterns.

Implementation of Lazy Lists

The implementation of lazy lists is quite subtle.

As a simplification, we consider for now that lazy lists are only lazy in their tail. `head` and `isEmpty` are computed when the lazy list is created.

This is not the actual behavior of lazy lists, but makes the implementation simpler to understand.

Here's the trait `TailLazyList`:

```
trait TailLazyList[+A]:  
  def isEmpty: Boolean  
  def head: A  
  def tail: TailLazyList[A]  
  ...
```

As for lists, all other methods can be defined in terms of these three.

Implementation of Lazy Lists (2)

Concrete implementations of lazy lists are defined in the TailLazyList companion object. Here's a first draft:

```
object TailLazyList:
  def cons[T](hd: T, tl: => TailLazyList[T]) = new TailLazyList[T]:
    def isEmpty = false
    def head = hd
    def tail = tl
    override def toString = "LazyList(" + hd + ", ?)"

  val empty = new TailLazyList[Nothing]:
    def isEmpty = true
    def head = throw NoSuchElementException("empty.head")
    def tail = throw NoSuchElementException("empty.tail")
    override def toString = "LazyList()"
```

Difference to List

The only important difference between the implementations of `List` and our (simplified) lazy list data type concerns `tl`, the second parameter of `TailLazyList.cons`.

For lazy lists, this is a by-name parameter.

That's why the second argument to `TailLazyList.cons` is not evaluated at the point of call.

Instead, it will be evaluated each time someone calls `tail` on a `TailLazyList` object.

Other LazyList Methods

The other lazy list methods are implemented analogously to their list counterparts.

For instance, here's filter:

```
extension [T](xs: TailLazyList[T])  
  def filter(p: T => Boolean): TailLazyList[T] =  
    if xs.isEmpty then xs  
    else if p(xs.head) then cons(xs.head, xs.tail.filter(p))  
    else xs.tail.filter(p)
```

Exercise

Consider this modification of lazyRange.

```
def lazyRange(lo: Int, hi: Int): TailLazyList[Int] =  
  print(lo+" ")  
  if lo >= hi then TailLazyList.empty  
  else TailLazyList.cons(lo, lazyRange(lo + 1, hi))
```

When you write lazyRange(1, 10).take(3).toList
what gets printed?

- ☐ Nothing
- ☐ 1
- ☐ 1 2 3
- ☐ 1 2 3 4
- ☐ 1 2 3 4 5 6 7 8 9

Exercise

Consider this modification of lazyRange.

```
def lazyRange(lo: Int, hi: Int): TailLazyList[Int] =  
  print(lo+" ")  
  if lo >= hi then TailLazyList.empty  
  else TailLazyList.cons(lo, lazyRange(lo + 1, hi))
```

When you write lazyRange(1, 10).take(3).toList
what gets printed?

- ☐ Nothing
- ☐ 1
- ☐ 1 2 3
- ☒ 1 2 3 4
- ☐ 1 2 3 4 5 6 7 8 9

Lazy Evaluation

Lazy Evaluation

The proposed implementation suffers from a serious potential performance problem: If `tail` is called several times, the corresponding lazy list will be recomputed each time.

This problem can be avoided by storing the result of the first evaluation of `tail` and re-using the stored result instead of recomputing `tail`.

This optimization is sound, since in a purely functional language an expression produces the same result each time it is evaluated.

We call this scheme *lazy evaluation* (as opposed to *by-name evaluation* in the case where everything is recomputed, and *strict evaluation* for normal parameters and `val` definitions.)

Lazy Evaluation in Scala

Haskell is a functional programming language that uses lazy evaluation by default.

Scala uses strict evaluation by default, but allows lazy evaluation of value definitions with the `lazy val` form:

```
lazy val x = expr
```

Exercise:

Consider the following program:

```
def expr =  
  val x = { print("x"); 1 }  
  lazy val y = { print("y"); 2 }  
  def z = { print("z"); 3 }  
  z + y + x + z + y + x
```

expr

If you run this program, what gets printed as a side effect of evaluating expr?

0 zyxzyx

0 xyzz

0 something else

0 xzyz

0 zyzz

Exercise:

Consider the following program:

```
def expr =  
  val x = { print("x"); 1 }  
  lazy val y = { print("y"); 2 }  
  def z = { print("z"); 3 }  
  z + y + x + z + y + x
```

expr

If you run this program, what gets printed as a side effect of evaluating expr?

- | | | | |
|---|----------------|---|------|
| 0 | zyxzyx | X | xzyz |
| 0 | xyzz | 0 | zyzz |
| 0 | something else | | |

Lazy Vals and Lazy Lists

Using a lazy value for tail, TailLazyList.cons can be implemented more efficiently:

```
def cons[T](hd: T, tl: => LazyList[T]) = new TailLazyList[T]:  
  def head = hd  
  lazy val tail = tl  
  ...
```

Seeing it in Action

To convince ourselves that the implementation of lazy lists really does avoid unnecessary computation, let's observe the execution trace of the expression:

```
lazyRange(1000, 10000).filter(isPrime).apply(1)
```

Seeing it in Action

To convince ourselves that the implementation of lazy lists really does avoid unnecessary computation, let's observe the execution trace of the expression:

```
lazyRange(1000, 10000).filter(isPrime).apply(1)
```

```
--> (if 1000 >= 10000 then empty           // by expanding lazyRange  
    else cons(1000, lazyRange(1000 + 1, 10000))  
    .filter(isPrime).apply(1))
```


Seeing it in Action

To convince ourselves that the implementation of lazy lists really does avoid unnecessary computation, let's observe the execution trace of the expression:

```
lazyRange(1000, 10000).filter(isPrime).apply(1)
```

```
--> (if 1000 >= 10000 then empty           // by expanding lazyRange  
    else cons(1000, lazyRange(1000 + 1, 10000))  
    .filter(isPrime).apply(1))
```

```
--> cons(1000, lazyRange(1000 + 1, 10000)) // by evaluating if  
    .filter(isPrime).apply(1)
```

Evaluation Trace (2)

Let's abbreviate `cons(1000, lazyRange(1000 + 1, 10000))` to `C1`.

```
C1.filter(isPrime).apply(1)
```

Evaluation Trace (2)

Let's abbreviate `cons(1000, lazyRange(1000 + 1, 10000))` to `C1`.

```
C1.filter(isPrime).apply(1)
```

```
--> (if C1.isEmpty then C1                                // by expanding filter
     else if isPrime(C1.head) then cons(C1.head, C1.tail.filter(isPrime))
     else C1.tail.filter(isPrime))
     .apply(1)
```

Evaluation Trace (2)

Let's abbreviate `cons(1000, lazyRange(1000 + 1, 10000))` to `C1`.

```
C1.filter(isPrime).apply(1)
```

```
--> (if C1.isEmpty then C1                      // by expanding filter
     else if isPrime(C1.head) then cons(C1.head, C1.tail.filter(isPrime))
     else C1.tail.filter(isPrime))
     .apply(1)
```

```
--> (if isPrime(C1.head) then cons(C1.head, C1.tail.filter(isPrime))
     else C1.tail.filter(isPrime))             // by eval. if
     .apply(1)
```

Evaluation Trace (2)

Let's abbreviate `cons(1000, lazyRange(1000 + 1, 10000))` to `C1`.

```
C1.filter(isPrime).apply(1)
```

```
--> (if C1.isEmpty then C1 // by expanding filter
     else if isPrime(C1.head) then cons(C1.head, C1.tail.filter(isPrime))
     else C1.tail.filter(isPrime))
     .apply(1)
```

```
--> (if isPrime(C1.head) then cons(C1.head, C1.tail.filter(isPrime))
    else C1.tail.filter(isPrime))      // by eval. if
    .apply(1)
```

```
--> (if isPrime(1000) then cons(C1.head, C1.tail.filter(isPrime))
      else C1.tail.filter(isPrime))      // by eval. head
      .apply(1)
```

Evaluation Trace (3)

```
-->> (if false then cons(C1.head, C1.tail.filter(isPrime)) // by eval. isPrime  
      else C1.tail.filter(isPrime))  
      .apply(1)
```

Evaluation Trace (3)

```
-->> (if false then cons(C1.head, C1.tail.filter(isPrime)) // by eval. isPrime
      else C1.tail.filter(isPrime))
      .apply(1)
```

```
--> C1.tail.filter(isPrime).apply(1) // by eval. if
```

Evaluation Trace (3)

```
-->> (if false then cons(C1.head, C1.tail.filter(isPrime)) // by eval. isPrime
      else C1.tail.filter(isPrime))
      .apply(1)
```

```
--> C1.tail.filter(isPrime).apply(1) // by eval. if
```

```
-->> lazyRange(1001, 10000) // by eval. tail
      .filter(isPrime).apply(1)
```

The evaluation sequence continues like this until:

Evaluation Trace (3)

```
-->> (if false then cons(C1.head, C1.tail.filter(isPrime)) // by eval. isPrime
      else C1.tail.filter(isPrime))
      .apply(1)
```

```
--> C1.tail.filter(isPrime).apply(1) // by eval. if
```

```
-->> lazyRange(1001, 10000) // by eval. tail
      .filter(isPrime).apply(1)
```

The evaluation sequence continues like this until:

```
-->> lazyRange(1009, 10000)
      .filter(isPrime).apply(1)
```

```
--> cons(1009, lazyRange(1009 + 1, 10000)) // by eval. lazyRange
      .filter(isPrime).apply(1)
```

Evaluation Trace (4)

Let's abbreviate `cons(1009, lazyRange(1009 + 1, 10000))` to `C2`.

```
C2.filter(isPrime).apply(1)
```

Evaluation Trace (4)

Let's abbreviate `cons(1009, lazyRange(1009 + 1, 10000))` to `C2`.

```
C2.filter(isPrime).apply(1)
```

```
--> cons(1009, C2.tail.filter(isPrime)).apply(1)
```

Evaluation Trace (4)

Let's abbreviate `cons(1009, lazyRange(1009 + 1, 10000))` to `C2`.

```
C2.filter(isPrime).apply(1)
```

```
--> cons(1009, C2.tail.filter(isPrime)).apply(1)
```

```
--> if 1 == 0 then cons(1009, C2.tail.filter(isPrime)).head // by eval. apply  
    else cons(1009, C2.tail.filter(isPrime)).tail.apply(0)
```

Assuming `apply` is defined like this in `LazyList[T]`:

```
def apply(n: Int): T =  
  if n == 0 then head  
  else tail.apply(n - 1)
```

Evaluation Trace (4)

Let's abbreviate `cons(1009, lazyRange(1009 + 1, 10000))` to `C2`.

```
C2.filter(isPrime).apply(1)
```

```
-->> cons(1009, C2.tail.filter(isPrime)).apply(1)           // by eval. filter
```

```
-->  if 1 == 0 then cons(1009, C2.tail.filter(isPrime)).head // by eval. apply
     else cons(1009, C2.tail.filter(isPrime)).tail.apply(0)
```

```
-->  cons(1009, C2.tail.filter(isPrime)).tail.apply(0)      // by eval. if
```

Evaluation Trace (4)

Let's abbreviate `cons(1009, lazyRange(1009 + 1, 10000))` to `C2`.

```
C2.filter(isPrime).apply(1)
```

```
-->> cons(1009, C2.tail.filter(isPrime)).apply(1)           // by eval. filter
```

```
-->  if 1 == 0 then cons(1009, C2.tail.filter(isPrime)).head // by eval. apply
     else cons(1009, C2.tail.filter(isPrime)).tail.apply(0)
```

```
-->  cons(1009, C2.tail.filter(isPrime)).tail.apply(0)      // by eval. if
```

```
-->  C2.tail.filter(isPrime).apply(0)                        // by eval. tail
```

Evaluation Trace (4)

Let's abbreviate `cons(1009, lazyRange(1009 + 1, 10000))` to `C2`.

```
C2.filter(isPrime).apply(1)
```

```
-->> cons(1009, C2.tail.filter(isPrime)).apply(1)           // by eval. filter
```

```
-->  if 1 == 0 then cons(1009, C2.tail.filter(isPrime)).head // by eval. apply
     else cons(1009, C2.tail.filter(isPrime)).tail.apply(0)
```

```
-->  cons(1009, C2.tail.filter(isPrime)).tail.apply(0)      // by eval. if
```

```
-->  C2.tail.filter(isPrime).apply(0)                        // by eval. tail
```

```
-->  lazyRange(1010, 10000).filter(isPrime).apply(0)      // by eval. tail
```

Evaluation Trace (5)

The process continues until

...

--> lazyRange(1013, 10000).filter(isPrime).apply(0)

Evaluation Trace (5)

The process continues until

```
...  
--> lazyRange(1013, 10000).filter(isPrime).apply(0)  
  
--> cons(1013, lazyRange(1013 + 1, 10000))           // by eval. lazyRange  
    .filter(isPrime).apply(0)
```

Let C3 be a shorthand for `cons(1013, lazyRange(1013 + 1, 10000)).`

```
== C3.filter(isPrime).apply(0)
```

Evaluation Trace (5)

The process continues until

```
...  
--> lazyRange(1013, 10000).filter(isPrime).apply(0)  
  
--> cons(1013, lazyRange(1013 + 1, 10000))           // by eval. lazyRange  
    .filter(isPrime).apply(0)
```

Let C3 be a shorthand for `cons(1013, lazyRange(1013 + 1, 10000))`.

```
==  C3.filter(isPrime).apply(0)  
  
-->> cons(1013, C3.tail.filter(isPrime)).apply(0)    // by eval. filter
```

Evaluation Trace (5)

The process continues until

```
...  
--> lazyRange(1013, 10000).filter(isPrime).apply(0)  
  
--> cons(1013, lazyRange(1013 + 1, 10000))           // by eval. lazyRange  
    .filter(isPrime).apply(0)
```

Let C3 be a shorthand for `cons(1013, lazyRange(1013 + 1, 10000))`.

```
==  C3.filter(isPrime).apply(0)  
  
-->> cons(1013, C3.tail.filter(isPrime)).apply(0)    // by eval. filter  
  
--> 1013                                              // by eval. apply
```

Only the part of the lazy list necessary to compute the result has been constructed.

Real-World Lazy List

The simplified implementation shown for LazyList has a lazy tail, but not a lazy head, nor a lazy isEmpty.

The real implementation is lazy for all three operations.

To do this, it maintain a lazy state variable, like this:

```
class LazyList[+T](init: => State[T]):  
  lazy val state: State[T] = init
```

```
enum State[T]:  
  case Empty  
  case Cons(hd: T, tl: LazyList[T])
```

Scala's Standard Lazy List

The standard Scala implementation for LazyList is lazy

- ▶ in its *structure* (its tail & whether the list is empty or not)
- ▶ but not in individual 'head' elements

Scala's Standard Lazy List

The standard Scala implementation for LazyList is lazy

- ▶ in its *structure* (its tail & whether the list is empty or not)
- ▶ but not in individual 'head' elements

It maintain a lazy “state” field, similar to what we saw:

```
final class LazyList[+A]  
  private(private[this] var lazyState: () => LazyList.State[A])  
  extends AbstractSeq[A], ... // many extensions  
  
private sealed trait State[+A]:  
  def head: A  
  def tail: LazyList[A]
```

<https://www.scala-lang.org/api/3.x/scala/collection/immutable/LazyList.html>

Computing with Infinite Sequences

Lazy Lists: Examples

Consider:

```
val xs1 = ??? #: LazyList.empty
```

```
val xs2 = 1 #: (??? : LazyList[Int])
```

What do these do?

```
xs1.isEmpty
```

```
xs2.isEmpty
```

```
xs2.tail.isEmpty
```


Infinite Lists

We saw that the structure of a lazy list is computed only when needed to produce a result.

This opens up the possibility to define infinite lists!

For instance, here is the (lazy) list of all integers starting from a given number:

```
def from(n: Int): LazyList[Int] = n #:: from(n + 1)
```

The list of all natural numbers:

Infinite Lists

We saw that the structure of a lazy list is computed only when needed to produce a result.

This opens up the possibility to define infinite lists!

For instance, here is the (lazy) list of all integers starting from a given number:

```
def from(n: Int): LazyList[Int] = n #:: from(n + 1)
```

The list of all natural numbers:

```
val nats = from(0)
```

The list of all multiples of 4:

Infinite Lists

We saw that the structure of a lazy list is computed only when needed to produce a result.

This opens up the possibility to define infinite lists!

For instance, here is the (lazy) list of all integers starting from a given number:

```
def from(n: Int): LazyList[Int] = n #:: from(n + 1)
```

The list of all natural numbers:

```
val nats = from(0)
```

The list of all multiples of 4:

```
nats.map(_ * 4)
```

Operations on Infinite Lists

Recall:

```
def from(n: Int): LazyList[Int] = n #:: from(n + 1)
val nats = from(0)
```

What does `nats.size` return?

Operations on Infinite Lists

Recall:

```
def from(n: Int): LazyList[Int] = n #:: from(n + 1)
val nats = from(0)
```

What does `nats.size` return?

Divergence: the expression fails to terminate!

Operations on Infinite Lists

Recall:

```
def from(n: Int): LazyList[Int] = n #:: from(n + 1)
val nats = from(0)
```

What does `nats.size` return?

Divergence: the expression fails to terminate!

How about `nats.toList`? (`toList` converts into standard Scala List)

Operations on Infinite Lists

Recall:

```
def from(n: Int): LazyList[Int] = n #:: from(n + 1)
val nats = from(0)
```

What does `nats.size` return?

Divergence: the expression fails to terminate!

How about `nats.toList`? (`toList` converts into standard Scala List)

Divergence again.

And `nats.drop(1).take(10).toList`?

Operations on Infinite Lists

Recall:

```
def from(n: Int): LazyList[Int] = n #:: from(n + 1)
val nats = from(0)
```

What does `nats.size` return?

Divergence: the expression fails to terminate!

How about `nats.toList`? (`toList` converts into standard Scala List)

Divergence again.

And `nats.drop(1).take(10).toList`?

Only latter is **OK**; produces `List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)`

The Sieve of Eratosthenes

The Sieve of Eratosthenes is an ancient technique to calculate prime numbers.

The idea is as follows:

- ▶ Start with all integers from 2, the first prime number.
- ▶ Eliminate all multiples of 2.
- ▶ The first element of the resulting list is 3, a prime number.
- ▶ Eliminate all multiples of 3.
- ▶ Iterate forever. At each step, the first number in the list is a prime number and we eliminate all its multiples.

The Sieve of Eratosthenes in Code

Here's a function that implements this principle:

```
def sieve(s: LazyList[Int]): LazyList[Int] =  
  s.head #:: sieve(s.tail.filter(_ % s.head != 0))  
  
val primes = sieve(from(2))
```

To see the list of the first N prime numbers, you can write

```
primes.take(N).toList
```

Back to Square Roots

Our previous algorithm for square roots always used a `isGoodEnough` test to tell when to terminate the iteration.

With lazy lists we can now express the concept of a converging sequence without having to worry about when to terminate it:

```
def sqrtSeq(x: Double): LazyList[Double] =  
  def improve(guess: Double) = (guess + x / guess) / 2  
  lazy val guesses: LazyList[Double] = 1 #:: guesses.map(improve)  
  guesses
```

Termination

We can add isGoodEnough later.

```
def isGoodEnough(guess: Double, x: Double) =  
  ((guess * guess - x) / x).abs < 0.0001  
  
sqrtSeq(2).filter(isGoodEnough(_, 2))
```

Exercise:

Consider two ways to express the infinite list of multiples of a given number N:

```
val xs = from(1).map(_ * N)
```

```
val ys = from(1).filter(_ % N == 0)
```

Which of the two lazy lists generates its results faster?

- ☐ from(1).map(_ * N)
- ☐ from(1).filter(_ % N == 0)
- ☐ there's no difference

Exercise:

Consider two ways to express the infinite list of multiples of a given number N:

```
val xs = from(1).map(_ * N)
```

```
val ys = from(1).filter(_ % N == 0)
```

Which of the two lazy lists generates its results faster?

- X from(1).map(_ * N)
- 0 from(1).filter(_ % N == 0)
- 0 there's no difference

Case Study

The Water Pouring Problem

- ▶ You are given some glasses of different sizes.
- ▶ Your task is to produce a glass with a given amount of water in it.
- ▶ You don't have a measure or balance.
- ▶ All you can do is:
 - ▶ fill a glass (completely)
 - ▶ empty a glass
 - ▶ pour from one glass to another until the first glass is empty or the second glass is full.

Example task:

You have two glasses. One holds 7 units of water, the other 3. Produce a glass filled with 6 units of water.

Strategy

Represent state space as transition graph

Explore state paths starting from smaller ones

Avoid looping back

States and Moves

Representations:

Glass: Int (glasses are numbered 0, 1, 2)

State: Vector[Int] (one entry per glass)

I.e. Vector(2, 3) would be a state where we have two glasses that have 2 and 3 units of water in it.

Moves:

Empty(glass)

Fill(glass)

Pour(from, to)

States and Moves

```
type Glass = Int
type State = Vector[Int]

enum Move:
  case Empty(glass: Glass)
  case Fill(glass: Glass)
  case Pour(from: Glass, to: Glass)
```

Actions

```
class Pouring(capacity: Vector[Int]):  
  
  extension (move: Move)  
    def change(prev: State): State = move match  
      case Empty(glass) => prev.updated(glass, 0)  
      case Fill(glass) => prev.updated(glass, capacity(glass))  
      case Pour(from, to) =>  
        val amount = prev(from).min(capacity(to) - prev(to))  
        prev.updated(from, prev(from) - amount).updated(to, prev(to) + amount)  
  
  val glasses = 0 until capacity.length  
  val actions = ( for glass <- glasses yield Empty(glass) ) ++  
    ( for glass <- glasses yield Fill(glass) ) ++  
    ( for from <- glasses; to <- glasses if from != to yield Pour(from, to) )
```

Paths

```
class Path(history: List[Move], val endState: State):  
  
  def extend(action: Move) =  
    Path(action :: history, action.change(endState))  
  
  override def toString =  
    (history.reverse.mkString(" ")) + " --> " + endState  
  
val initialState = capacity.map(_ => 0)  
val initialPath = Path(Nil, initialState)
```

Paths (2)

```
def from(current: Set[Path], explored: Set[State]): LazyList[Set[Path]] =  
  if (current.isEmpty) LazyList.empty  
  else  
    val more: Set[Path] =  
      for  
        start <- current  
        next <- actions.map(start.extend)  
        if !explored.contains(next.endState)  
      yield next  
    current #:: from(more, explored ++ (more.map(_.endState)))  
  
lazy val pathSets = from(Set(initialPath), Set(initialState))
```

Solutions

```
def solutions(target: Int): LazyList[Path] =  
  for  
    paths <- pathSets  
    path <- paths  
    if path.endState.contains(target)  
  yield path  
  
end Pouring
```

Variants

In a program of the complexity of the pouring program, there are many choices to be made.

Choice of representations.

- ▶ Specific classes for moves and paths, or some encoding?
- ▶ Object-oriented methods, or naked data structures with functions?

The present elaboration is just one solution, and not necessarily the shortest one.

Guiding Principles for Good Design

- ▶ Name everything you can.
- ▶ Put operations into natural scopes.
- ▶ Keep degrees of freedom for future refinements.

How Other Languages Do It

How Other Languages Do It

Haskell

Some languages, like Haskell, use laziness by default.

So Haskell lists are automatically lazy. (Note that Haskell uses `:` for cons.)

```
let guesses = 1 : map improve guesses
```

How Other Languages Do It

Haskell

Some languages, like Haskell, use laziness by default.

So Haskell lists are automatically lazy. (Note that Haskell uses `:` for cons.)

```
let guesses = 1 : map improve guesses
```

OCaml

Languages like OCaml are eager but allow defining *cyclic* data structures.

```
let rec alternate = 0 :: 1 :: alternate
```

To define lazy lists, OCaml requires wrapping the lazy parts in functions.
(These functions are often referred to as “thunks”.)

```
let rec guesses = Stream.cons (fun () -> (1, Stream.map improve guesses))
```