Structural Induction on Trees

Principles of Functional Programming

Structural Induction on Trees

Structural induction is not limited to lists; it applies to any tree structure.

The general induction principle is the following:

To prove a property $P(t)$ for all trees t of a certain type,

- show that $P(l)$ holds for all leaves l of a tree,
- for each type of internal node t with subtrees $s_1, \ldots, s_n$, show that $P(s_1) \wedge \ldots \wedge P(s_n)$ *implies* $P(t)$.

Example: IntSets

Recall our definition of `IntSet` with the operations `contains` and `incl`:

```
abstract class IntSet:
  def incl(x: Int): IntSet
  def contains(x: Int): Boolean

object Empty extends IntSet:
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = NonEmpty(x, Empty, Empty)
```

Example: IntSets (2)

```
case class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet:

  def contains(x: Int): Boolean =
    if x < elem then left.contains(x)
    else if x > elem then right.contains(x)
    else true

  def incl(x: Int): IntSet =
    if x < elem then NonEmpty(elem, left.incl(x), right)
    else if x > elem then NonEmpty(elem, left, right.incl(x))
    else this
```

The Laws of IntSet

What does it mean to prove the correctness of this implementation?

One way to define and show the correctness of an implementation consists of proving the laws that it respects.

In the case of `IntSet`, we have the following three laws:

For any set `s`, and elements `x` and `y`:

```
Empty.contains(x)        =  false
s.incl(x).contains(x)    =  true
s.incl(x).contains(y)    =  s.contains(y)         if x != y
```

(In fact, we can show that these laws completely characterize the desired data type).

Proving the Laws of IntSet (1)

How can we prove these laws?

Proposition 1: `Empty.contains(x) =  false`.

Proof: According to the definition of `contains` in `Empty`.

Proving the Laws of IntSet (2)

Proposition 2: `s.incl(x).contains(x) = true`

Proof by structural induction on `s`.

$\boxed{\textbf{Base case: } \texttt{Empty}}$

```
Empty.incl(x).contains(x)
```

Proving the Laws of IntSet (2)

Proposition 2: `s.incl(x).contains(x) = true`

Proof by structural induction on `s`.

Base case: `Empty`

```
  Empty.incl(x).contains(x)

  =   NonEmpty(x, Empty, Empty).contains(x) // by definition of Empty.incl
```

Proving the Laws of IntSet (2)

Proposition 2: `s.incl(x).contains(x) = true`

Proof by structural induction on `s`.

$\boxed{\textbf{Base case: } \texttt{Empty}}$

```
Empty.incl(x).contains(x)

=   NonEmpty(x, Empty, Empty).contains(x) // by definition of Empty.incl

=   true                                  // by definition of NonEmpty.contains
```

Proving the Laws of IntSet (3)

Induction step: `NonEmpty(x, l, r)`

```
NonEmpty(x, l, r).incl(x).contains(x)
```

Proving the Laws of IntSet (3)

Induction step: `NonEmpty(x, l, r)`

```
NonEmpty(x, l, r).incl(x).contains(x)

=   NonEmpty(x, l, r).contains(x)        //  by definition of NonEmpty.incl
```

Proving the Laws of IntSet (3)

Induction step: `NonEmpty(x, l, r)`

```
NonEmpty(x, l, r).incl(x).contains(x)

=   NonEmpty(x, l, r).contains(x)        //  by definition of NonEmpty.incl

=   true                                 //  by definition of NonEmpty.contains
```

Proving the Laws of IntSet (4)

$\boxed{\textbf{Induction step: } \texttt{NonEmpty(y, l, r)} \textbf{ where } \texttt{y < x}}$

```
NonEmpty(y, l, r).incl(x).contains(x)
```

Proving the Laws of IntSet (4)

Induction step: `NonEmpty(y, l, r)` **where** `y < x`

```
NonEmpty(y, l, r).incl(x).contains(x)

=   NonEmpty(y, l, r.incl(x)).contains(x) // by definition of NonEmpty.incl
```

Proving the Laws of IntSet (4)

Induction step: `NonEmpty(y, l, r)` **where** `y < x`

```
NonEmpty(y, l, r).incl(x).contains(x)

=   NonEmpty(y, l, r.incl(x)).contains(x) // by definition of NonEmpty.incl

=   r.incl(x).contains(x)                 // by definition of NonEmpty.contains
```

Proving the Laws of IntSet (4)

Induction step: `NonEmpty(y, l, r)` **where** `y < x`

```
NonEmpty(y, l, r).incl(x).contains(x)

=   NonEmpty(y, l, r.incl(x)).contains(x) // by definition of NonEmpty.incl

=   r.incl(x).contains(x)                 // by definition of NonEmpty.contains

=   true                                  // by the induction hypothesis
```

Proving the Laws of IntSet (4)

$\boxed{\textbf{Induction step: } \texttt{NonEmpty(y, l, r)} \textbf{ where } \texttt{y < x}}$

```
NonEmpty(y, l, r).incl(x).contains(x)

=   NonEmpty(y, l, r.incl(x)).contains(x) // by definition of NonEmpty.incl

=   r.incl(x).contains(x)                 // by definition of NonEmpty.contains

=   true                                  // by the induction hypothesis
```

$\boxed{\textbf{Induction step: } \texttt{NonEmpty(y, l, r)} \textbf{ where } \texttt{y > x}}$ is analogous

Proving the Laws of IntSet (5)

Proposition 3: If `x != y` then

```
xs.incl(y).contains(x)  =  xs.contains(x).
```

Proof by structural induction on `s`. Assume that `y < x` (the dual case `x < y` is analogous).

Base case: `Empty`

```
Empty.incl(y).contains(x)                    // to show: =  Empty.contains(x)
```

Proving the Laws of IntSet (5)

Proposition 3: If `x != y` then

```
xs.incl(y).contains(x)  =  xs.contains(x).
```

Proof by structural induction on `s`. Assume that `y < x` (the dual case `x < y` is analogous).

Base case: `Empty`

```
Empty.incl(y).contains(x)                    // to show: =  Empty.contains(x)

=   NonEmpty(y, Empty, Empty).contains(x)  // by definition of Empty.incl
```

Proving the Laws of IntSet (5)

Proposition 3: If `x != y` then

```
xs.incl(y).contains(x)  =  xs.contains(x).
```

Proof by structural induction on `s`. Assume that `y < x` (the dual case `x < y` is analogous).

Base case: `Empty`

```
Empty.incl(y).contains(x)                    // to show: =  Empty.contains(x)

=   NonEmpty(y, Empty, Empty).contains(x)  // by definition of Empty.incl

=   Empty.contains(x)                      // by definition of NonEmpty.contain
```

Proving the Laws of IntSet (6)

For the inductive step, we need to consider a tree `NonEmpty(z, l, r)`. We distinguish five cases:

1. `z = x`
2. `z = y`
3. `z < y < x`
4. `y < z < x`
5. `y < x < z`

First Two Cases: `z = x`, `z = y`

Induction step: `NonEmpty(x, l, r)`

```
NonEmpty(x, l, r).incl(y).contains(x)  // to show: =  NonEmpty(x, l, r).contain
```

First Two Cases: z = x, z = y

Induction step: `NonEmpty(x, l, r)`

```
NonEmpty(x, l, r).incl(y).contains(x)  // to show: =  NonEmpty(x, l, r).contain

=   NonEmpty(x, l.incl(y), r).contains(x) // by definition of NonEmpty.incl
```

First Two Cases: `z = x`, `z = y`

Induction step: `NonEmpty(x, l, r)`

```
NonEmpty(x, l, r).incl(y).contains(x)  // to show: =  NonEmpty(x, l, r).contain

=   NonEmpty(x, l.incl(y), r).contains(x) // by definition of NonEmpty.incl

=   true                                  // by definition of NonEmpty.contains
```

First Two Cases: z = x, z = y

Induction step: NonEmpty(x, l, r)

```
NonEmpty(x, l, r).incl(y).contains(x)  // to show: =  NonEmpty(x, l, r).contain

=   NonEmpty(x, l.incl(y), r).contains(x) // by definition of NonEmpty.incl

=   true                                  // by definition of NonEmpty.contains

=   NonEmpty(x, l, r).contains(x)         // by definition of NonEmpty.contains
```

First Two Cases: `z = x, z = y`

Induction step: `NonEmpty(x, l, r)`

```
NonEmpty(x, l, r).incl(y).contains(x)  // to show: =  NonEmpty(x, l, r).contain

=   NonEmpty(x, l.incl(y), r).contains(x) // by definition of NonEmpty.incl

=   true                                  // by definition of NonEmpty.contains

=   NonEmpty(x, l, r).contains(x)         // by definition of NonEmpty.contains
```

Induction step: `NonEmpty(y, l, r)`

```
NonEmpty(y, l, r).incl(y).contains(x)     // to show: =  NonEmpty(y, l, r).cont
```

First Two Cases: `z = x, z = y`

Induction step: `NonEmpty(x, l, r)`

```
NonEmpty(x, l, r).incl(y).contains(x)  // to show: =  NonEmpty(x, l, r).contain

=   NonEmpty(x, l.incl(y), r).contains(x) // by definition of NonEmpty.incl

=   true                                  // by definition of NonEmpty.contains

=   NonEmpty(x, l, r).contains(x)         // by definition of NonEmpty.contains
```

Induction step: `NonEmpty(y, l, r)`

```
NonEmpty(y, l, r).incl(y).contains(x)     // to show: =  NonEmpty(y, l, r).cont

=    NonEmpty(y, l, r).contains(x)        // by definition of NonEmpty.incl
```

Case z < y

Induction step: `NonEmpty(z, l, r)` **where** `z < y < x`

```
NonEmpty(z, l, r).incl(y).contains(x)  // to show: =  NonEmpty(z, l, r).contain
```

Case z < y

Induction step: `NonEmpty(z, l, r)` **where** `z < y < x`

```
NonEmpty(z, l, r).incl(y).contains(x)  // to show: =  NonEmpty(z, l, r).contain

=   NonEmpty(z, l, r.incl(y)).contains(x)  // by definition of NonEmpty.incl
```

Case z < y

Induction step: `NonEmpty(z, l, r)` **where** `z < y < x`

```
NonEmpty(z, l, r).incl(y).contains(x)  // to show: =  NonEmpty(z, l, r).contain

=   NonEmpty(z, l, r.incl(y)).contains(x)  // by definition of NonEmpty.incl

=   r.incl(y).contains(x)                  // by definition of NonEmpty.contain
```

Case z < y

Induction step: NonEmpty(z, l, r) **where** z < y < x

```
NonEmpty(z, l, r).incl(y).contains(x)  // to show: =  NonEmpty(z, l, r).contain

=   NonEmpty(z, l, r.incl(y)).contains(x)  // by definition of NonEmpty.incl

=   r.incl(y).contains(x)                  // by definition of NonEmpty.contain

=   r.contains(x)                          // by the induction hypothesis
```

Case z < y

Induction step: `NonEmpty(z, l, r)` **where** `z < y < x`

```
NonEmpty(z, l, r).incl(y).contains(x)  // to show: =  NonEmpty(z, l, r).contain

=   NonEmpty(z, l, r.incl(y)).contains(x)  // by definition of NonEmpty.incl

=   r.incl(y).contains(x)                  // by definition of NonEmpty.contain

=   r.contains(x)                          // by the induction hypothesis

=   NonEmpty(z, l, r).contains(x)          // by definition of NonEmpty.contain
```

Case y < z < x

Induction step: `NonEmpty(z, l, r)` **where** `y < z < x`

```
NonEmpty(z, l, r).incl(y).contains(x)   // to show: =  NonEmpty(z, l, r).contai
```

Case y < z < x

Induction step: `NonEmpty(z, l, r)` **where** `y < z < x`

```
NonEmpty(z, l, r).incl(y).contains(x)   // to show: =  NonEmpty(z, l, r).contai

=   NonEmpty(z, l.incl(y), r).contains(x)  // by definition of NonEmpty.incl
```

Case y < z < x

Induction step: `NonEmpty(z, l, r)` **where** `y < z < x`

```
NonEmpty(z, l, r).incl(y).contains(x)   // to show: =  NonEmpty(z, l, r).contai

=   NonEmpty(z, l.incl(y), r).contains(x)  // by definition of NonEmpty.incl

=   r.contains(x)                          // by definition of NonEmpty.contain
```

Case y < z < x

Induction step: `NonEmpty(z, l, r)` **where** `y < z < x`

```
NonEmpty(z, l, r).incl(y).contains(x)   // to show: =  NonEmpty(z, l, r).contai

=   NonEmpty(z, l.incl(y), r).contains(x)  // by definition of NonEmpty.incl

=   r.contains(x)                          // by definition of NonEmpty.contain

=   NonEmpty(z, l, r).contains(x)          // by definition of NonEmpty.contain
```

Case `x < z`

Induction step: `NonEmpty(z, l, r)` **where** `y < x < z`

```
NonEmpty(z, l, r).incl(y).contains(x)   // to show: =  NonEmpty(z, l, r).contai
```

Case `x < z`

Induction step: `NonEmpty(z, l, r)` **where** `y < x < z`

```
NonEmpty(z, l, r).incl(y).contains(x)   // to show: =  NonEmpty(z, l, r).contai

=   NonEmpty(z, l.incl(y), r).contains(x)  // by definition of NonEmpty.incl
```

Case x < z

Induction step: `NonEmpty(z, l, r)` **where** `y < x < z`

```
NonEmpty(z, l, r).incl(y).contains(x)   // to show: =  NonEmpty(z, l, r).contai

=   NonEmpty(z, l.incl(y), r).contains(x)  // by definition of NonEmpty.incl

=   l.incl(y).contains(x)                  // by definition of NonEmpty.contain
```

Case x < z

Induction step: `NonEmpty(z, l, r)` **where** `y < x < z`

```
NonEmpty(z, l, r).incl(y).contains(x)   // to show: =  NonEmpty(z, l, r).contai

=   NonEmpty(z, l.incl(y), r).contains(x)  // by definition of NonEmpty.incl

=   l.incl(y).contains(x)                  // by definition of NonEmpty.contain

=   l.contains(x)                          // by the induction hypothesis
```

Case x < z

Induction step: `NonEmpty(z, l, r)` **where** `y < x < z`

```
NonEmpty(z, l, r).incl(y).contains(x)   // to show: =  NonEmpty(z, l, r).contai

=   NonEmpty(z, l.incl(y), r).contains(x)  // by definition of NonEmpty.incl

=   l.incl(y).contains(x)                  // by definition of NonEmpty.contain

=   l.contains(x)                          // by the induction hypothesis

=   NonEmpty(z, l, r).contains(x)          // by definition of NonEmpty.contain
```

These are all the cases, so the proposition is established.

Exercise (Hard)

Suppose we add a function `union` to `IntSet`:

```
abstract class IntSet:
  ...
  def union(other: IntSet): IntSet

object Empty extends IntSet:
  ...
  def union(other: IntSet) = other

class NonEmpty(x: Int, l: IntSet, r: IntSet) extends IntSet:
  ...
  def union(other: IntSet): IntSet = l.union(r.union(other)).incl(x)
```

Exercise (Hard)

The correctness of `union` can be translated into the following law:

Proposition 4:

```
xs.union(ys).contains(x)  =  xs.contains(x) || ys.contains(x)
```

Show proposition 4 by using structural induction on `xs`.

Lazy Lists

Principles of Functional Programming

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] extends Seq[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 (simplified) `LazyList` concern `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 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?

O Nothing
O 1
O 1 2 3
O 1 2 3 4
O 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?

```
O          Nothing
O          1
X          1 2 3
O          1 2 3 4
O          1 2 3 4 5 6 7 8 9
```

Lazy Evaluation

Principles of Functional Programming

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`?

```
O     zyxzyx                    O     xzyz
O     xyzz                      O     zyzz
O     something else
```

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`?

O	zyxzyx	X	xzyz
O	xyzz	O	zyzz
O	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.

RealWorld 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])
```

Computing with Infinite Sequences

Principles of Functional Programming

Infinite Lists

You saw that the elements of a lazy list are computed only when they are 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

You saw that the elements of a lazy list are computed only when they are 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

You saw that the elements of a lazy list are computed only when they are 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)
```

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?

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

Case Study

Principles of Functional Programming

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 4. Produce a glass filled with 6 units of water.

Strategy

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)
```

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.