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:

```scala
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])
```