Lazy Evaluation

Lazy Evaluation

The proposed implementation suffers from a serious potential performance problem: If `tail` is called several times, the corresponding stream 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`

Lazy Vals and Streams

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

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

Seeing it in Action

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

```
(streamRange(1000, 10000) filter isPrime) apply 1
```

Seeing it in Action

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

```
      (streamRange(1000, 10000) filter isPrime) apply 1

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

Seeing it in Action

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

```
     (streamRange(1000, 10000) filter isPrime) apply 1

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

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

Evaluation Trace (2)

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

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

Evaluation Trace (2)

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

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

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

Evaluation Trace (2)

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

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

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

-->  (if (isPrime(C1.head)) 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, streamRange(1000 + 1, 10000))` to `C1`.

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

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

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

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

Evaluation Trace (3)

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

Evaluation Trace (3)

```
-->> (if (false) 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) 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

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

The evaluation sequence continues like this until:

Evaluation Trace (3)

```
-->> (if (false) 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

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

The evaluation sequence continues like this until:

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

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

Evaluation Trace (4)

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

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

Evaluation Trace (4)

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

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

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

-->  if (1 == 0) 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 `Stream[T]`:

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

Evaluation Trace (4)

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

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

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

-->  if (1 == 0) 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, streamRange(1009 + 1, 10000))` to `C2`.

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

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

 -->  if (1 == 0) 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, streamRange(1009 + 1, 10000))` to `C2`.

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

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

 -->  if (1 == 0) 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

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

Evaluation Trace (5)

The process continues until

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

Evaluation Trace (5)

The process continues until

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

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

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

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

Evaluation Trace (5)

The process continues until

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

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

Let `C3` be a shorthand for `cons(1013, streamRange(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

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

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

Let `C3` be a shorthand for `cons(1013, streamRange(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 stream necessary to compute the result has been constructed.