Lazy Evaluation

Lazy Evaluation

- The proposed implementation suffers from a serious 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)
- Roughly, laziness means: do things as late as possible and never do them twice
- Lazy evaluation is a very powerful principle because it avoids both unnecessary or repeated computations
- Haskell is a functional programming language that uses lazy evaluation by default
- Scala does not use lazy evaluation every time because it is quite unpredictable in when computations happen and how much space they take; this is mostly caused by the mutable side effects
- Scala uses strict evaluation by default, but allows lazy evaluation of value definitions with the lazy val form: ***lazy val x = expr***
- The behavior of ***lazy val*** and ***def*** diverge from the second time the variables are called; for ***def x***, every time you call, the expression is reevaluated, whereas for ***lazy val***, the expression is reused every time except for the first time, when it is evaluated

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? – ***xzyz****, val x is evaluated when it is declared, and never afterwards, lazy val is evaluated when first called and never afterwards and def z is evaluated each time it is called.*

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
    …
}
```

Evaluation Trace

- 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
else cons(1000, streamRange(1000 + 1, 10000))
.filter(isPrime).apply(1)
--> cons(1000, streamRange(1000 + 1, 10000))
.filter(isPrime).apply(1)
Let's abbreviate cons(1000, streamRange(1000 + 1, 10000)) to C1
--> C1.filter(isPrime).apply(1)
--> (if (C1.isEmpty) C1
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))
.apply(1)
--> if (isPrime(1000)) cons(C1.head, C1.tail.filter(isPrime))
else C1.tail.filter(isPrime))
.apply(1)
--> if (false) cons(C1.head, C1.tail.filter(isPrime))
else C1.tail.filter(isPrime))
.apply(1)
--> C1.tail.filter(isPrime)).apply(1)
--> streamRange(1001, 10000)
.fliter(isPrime).apply(1)
The evaluation sequence continues like this until:
--> streamRange(1009, 10000)
.filter(isPrime).apply(1)
--> cons(1009, streamRange(1009 + 1, 10000))
.filter(isPrime).apply(1)
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
else cons(1009, C2.tail.filter(isPrime)).tail.apply(0)
--> cons(1009, C2.tail.filter(isPrime)).tail.apply(0)
--> C2.tail.filter(isPrime).apply(0)
--> streamRange(1010, 10000).filter(isPrime).apply(0)
The process continues until:
--> streamRange(1013, 10000).filter(isPrime).apply(0)
--> cons(1013, streamRange(1013 +1, 10000)).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)
--> 1013
Only the part of the stream necessary to compute the result has been constructed