CS2030 Lecture 9

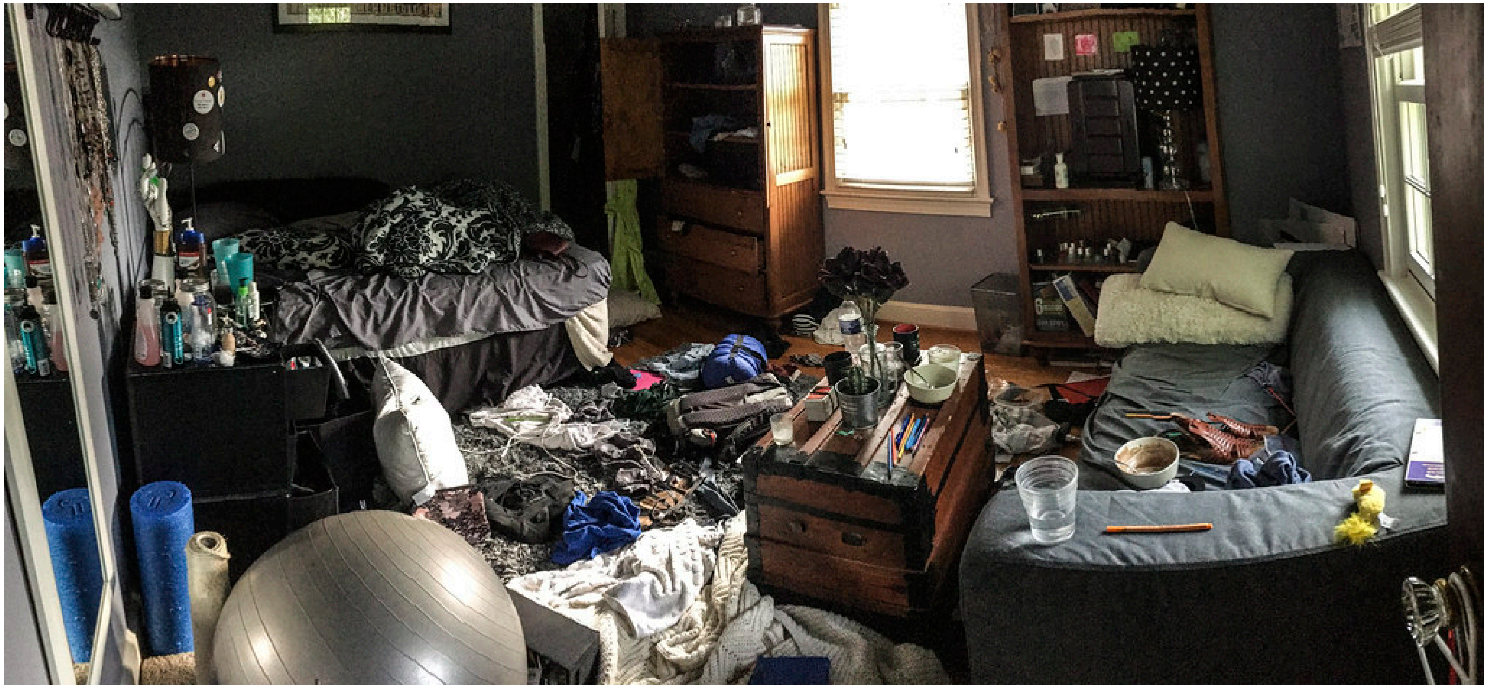
The Art of Being Lazy: 1

Terence Sim

Eager vs Lazy Evaluation

Eager evaluation: computation is immediately carried out when the expression is encountered.

Lazy evaluation: computation is postponed, and carried out only upon demand.



"teen's messy room" by woodleywonderworks is licensed under CC BY 2.0

Benefits of Lazy Evaluation

Avoids unnecessary computation

Amortizes time complexity (when used with memoization (aka caching))

Allows for infinite data structures

New abstraction

`freeze(<expr>) => <expr>` is not evaluated now, but postponed indefinitely. `<expr>` is wrapped in a *thunk.*

`thaw(thunk) =>` forces the evaluation of the `<expr>` inside the thunk.

`thaw(freeze(<expr>)) ≡ <expr>`

Lazy Lists

`public class LazyList<T>`

<u>Constructors</u>:

`LLmake(a, b) ≡ new LazyList(a, freeze(b))`

Note that a is eagerly evaluated, but b is frozen.

`makeEmpty()`: returns an empty `LazyList`

<u>Methods</u>:

`head()`: returns `a`

`tail()`: returns `thaw(freeze(b))`

`isEmpty()`: returns `true` if `LazyList` is empty, otherwise `false`

Lazy Lists: example

Demo

```
LazyList<Integer> ll =
      LLmake(1, LLmake(2, LazyList.makeEmpty()));

System.out.println(ll);
=> head: 1
tail: thunk$1/0x0000000800060840@2f410acf

System.out.println(ll.tail());
=> head: 2
tail: thunk$16/0x0000000800066440@2a18f23c
```

Start with empty `LL` and add to the front (the head).

cf `ArrayList`, which starts with empty `AL` and adds to the back.

Integers from a (inclusive) to b (exclusive)

Demo

```
LazyList<Integer> intRange(int a, int b) {
    if (a >= b)
        return LazyList.makeEmpty();
    return LLmake(a, intRange(a+1, b)); }

var l2 = intRange(4, 12);
System.out.println(l2);
System.out.println(l2.tail());
System.out.println(l2.tail().tail());
=> head: 4
tail: thunk$17/0x0000000800065840@16c0663d
head: 5
tail: thunk$17/0x0000000800065840@23223dd8
head: 6
tail: thunk$17/0x0000000800065840@4ec6a292
```

map: apply function to every element

Demo

```
public <R> LazyList<R> map(Function<T,R> f) {
    if (this.isEmpty())
        return LazyList.makeEmpty();
    else
        return LLmake(f.apply(this.head()),
                            this.tail().map(f));
}
```

Note the use of recursion: we recursively apply `map` to the `tail` of the list.

But this is frozen because it is the 2nd argument of `LLmake`.

Compute the squares of 1,2,3 .. 10

Demo

```
intRange(1,11).map(x -> x*x).print();
=> (* 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, *)
```

Generate a LazyList of integers from 1 to 10

Square every element

Print it

Note the declarative style: no loops!

`filter`: keep elements that satisfy predicate

Demo

```
public LazyList<T> filter(Predicate<T> pred) {
    if (this.isEmpty())
        return this;
    else if (pred.test(this.head()))
        return LLmake(this.head(),
                      this.tail().filter(pred));
    else
        return this.tail().filter(pred);  }

intRange(1,21).filter(x-> x%2==0).print();
=> (* 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, *)
```

Fermat’s puzzle

26 is an integer such that one less is a perfect square, and one more is a perfect cube.

Is there any other such number?

```
 intRange(1, 50000)
            .filter(x-> isSquare(x-1))
            .filter(x-> isCube(x+1))
            .print();


=> (* 26, *)
```

But is it really lazy?

Instrumentation refers to the measure of a product's performance, to diagnose errors, and to write trace information. https://en.wikipedia.org/wiki/Instrumentation_(computer_programming)

```
<T> Predicate<T> predInstrument(Predicate<T> p,
                                String msg) {
      return x-> { System.out.println(msg + x);
                   return p.test(x); };
}
```

Let’s wrap our predicate with a print statement to see what is going on.

Instrumenting map

Do the same for our map

```
<T,U> Function<T,U> funcInstrument(Function<T,U> f
                                   String msg) {
        return x-> { System.out.println(msg + x);
                     return f.apply(x);};
}
```

Now use them to investigate

Demo

```
var ll =
  intRange(1, 21)
    .filter(predInstrument(x->x%2==0, "filter: "))
    .map(funcInstrument(x->x*x, "map: "));

=> filter: 1
filter: 2
map: 2
```

Predicate is called twice, and Function called once. Can you understand why?

Compare with ArrayList

Demo

```
List<Integer> list = new ArrayList<>();
for (int i=1; i<21; i++)
    list.add(i);


var result =
  listMap(funcInstrument(x->x*x, "map: "),
          listFilter(predInstrument(x->x%2==0,
                                    "filter: "),
                     list));
```

output

Demo

```
filter: 1
filter: 2
filter: 3
filter: 4
filter: 5
filter: 6
filter: 7
filter: 8
filter: 9
filter: 10
filter: 11
filter: 12
filter: 13
filter: 14
filter: 15
filter: 16
```

```
filter: 17
filter: 18
filter: 19
filter: 20
map: 2
map: 4
map: 6
map: 8
map: 10
map: 12
map: 14
map: 16
map: 18
map: 20
```

Let’s print our LazyList

Demo

```
intRange(1, 21)
    .filter(predInstrument(x->x%2==0, "filter: "))
    .map(funcInstrument(x->x*x, "map: ")).print();
```

```
filter: 1
filter: 2
map: 2
(* 4, filter: 3
filter: 4
map: 4
16, filter: 5
filter: 6
map: 6
36, filter: 7
```

```
  .  .  .
map: 16
256, filter: 17
filter: 18
map: 18
324, filter: 19
filter: 20
map: 20
400, *)
```

Notice how printing thaws the frozen computation!

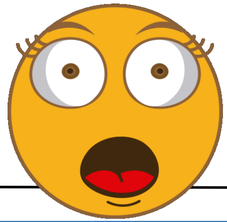
add: adding 2 LazyLists element-wise

Demo

```
public LazyList<T> add(LazyList<T> other,
                       BinaryOperator<T> add) {
  if (this.isEmpty() || other.isEmpty())
    return LazyList.makeEmpty();
  else
    return LLmake(add.apply(this.head(),
                            other.head()),
                  this.tail().add(other.tail(),
                                  add)); }


var ll = intRange(1, 11);
var twos = ll.map(x->2);
ll.add(twos, (x,y)->x+y).print();
=> (* 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, *)
```

Infinite LazyLists!



Demo

```
LazyList<Integer> integersFrom(int n) {
    return LLmake(n, integersFrom(n+1)); }
integersFrom(1).limit(50).print();


=> (* 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27,
28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
41, 42, 43, 44, 45, 46, 47, 48, 49, 50, *)
```

```
LazyList<T> limit(long maxSize) {
   if (maxSize==0)
         return LazyList.makeEmpty();
   return LLmake(this.head(),
                    this.tail().limit(maxSize - 1));   }
```

Prime numbers

Of course, one way to check if a number n is prime is to see if n can be found in a list, L, of primes. To generate L, an ancient method called the Sieve of Eratosthenes may be used:

1. Start by listing all integers greater than 1. Call this list C. Also, let L be an empty list.
2. Take the first number $p = 2$ in C, and add it to L. This is the first prime.
3. In C, cross out all multiples of p.
4. Let p be the next uncrossed number in C. This is the next prime. Add it to L, and repeat from Step 3.

	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~

Prime sieve

Demo

```
LazyList<Integer> sieve(LazyList<Integer> s) {
   return LLmake(s.head(),
                 sieve(s.tail()
                        .filter(x->
                               x%s.head()!=0)));
}

var primes = sieve(integersFrom(2));
```

Another way to get integers

Demo

```
LazyList<Integer> integers;

integers = LLmake(1, integers.map(x->x+1));
```

Fibonacci numbers

Demo

The infinite sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

The next number is the sum of the previous two.

```
LazyList<Integer> fib;
fib = LLmake(0,
             LLmake(1,
                    fib.add(fib.tail(),
                            (x,y)->x+y)));
```

0	1	1	2	3	5			← fib
		0	1	1	2	3	5	← fib
		1	1	2	3	5		← fib.tail()

To be Continued