COMP 322: Parallel and Concurrent Programming

Lecture 4: Lazy Computation

Zoran Budimlić and Mack Joyner
{zoran, mjoyner}@rice.edu

http://comp322.rice.edu

Lazy computation

Lazy evaluation

Logging library
```
Log.i(TAG,      "current input: " + input.toString()); // eager
```

Lazy evaluation

Logging library
```
Log.i(TAG,      "current input: " + input.toString()); // eager
Log.i(TAG, ()→"current input: " + input.toString()); // lazy
```

Lazy evaluation

Logging library
```
Log.i(TAG,      "current input: " + input.toString()); // eager

Log.i(TAG, ()→"current input: " + input.toString()); // lazy
```

Why lazy?

Maybe it's expensive to compute something.

Maybe you won't actually need it (e.g., if you disabled logging).

The idea: defer computation of a value until (and if) you need it.

Related idea: Memoization

The idea: compute a value <u>once</u> when you need it, then save it.

Deep, powerful idea in computer science (e.g., dynamic programming).

Lazy memo implementation (simplified)

```java
public class Lazy<T> {
  private T contents;
  private Supplier<T> supplier;

  private Lazy(Supplier<T> supplier) {
    contents = null;
    this.supplier = supplier;
  }

  public T get() {
    if (contents ≠ null) {
      return contents;
    }

    if (supplier ≠ null) {
      contents = supplier.get();
      supplier = null;
    }

    return contents;
  }
}
```

Lazy memo implementation (simplified)

```java
public class Lazy<T> {
    private T contents;
    private Supplier<T> supplier;

    private Lazy(Supplier<T> supplier) {
        contents = null;
        this.supplier = supplier;
    }

    public T get() {
        if (contents ≠ null) {
            return contents;
        }

        if (supplier ≠ null) {
            contents = supplier.get();
            supplier = null;
        }

        return contents;
    }
```

> Private constructor (as usual) plus a factory method (Lazy.*of*)

Lazy memo implementation (simplified)

```java
public class Lazy<T> {
  private T contents;
  private Supplier<T> supplier;

  private Lazy(Supplier<T> supplier) {
    contents = null;
    this.supplier = supplier;
  }

  public T get() {
    if (contents ≠ null) {
      return contents;
    }

    if (supplier ≠ null) {
      contents = supplier.get();
      supplier = null;
    }

    return contents;
  }
}
```

If we've already computed the answer, return it.

Lazy memo implementation (simplified)

```java
public class Lazy<T> {
  private T contents;
  private Supplier<T> supplier;

  private Lazy(Supplier<T> supplier) {
    contents = null;
    this.supplier = supplier;
  }

  public T get() {
    if (contents ≠ null) {
      return contents;
    }

    if (supplier ≠ null) {
      contents = supplier.get();
      supplier = null;
    }

    return contents;
  }
}
```

Call the lambda (once), get the result, forget the lambda.

Lazy memo implementation (simplified)

```java
public class Lazy<T> {
  private T contents;
  private Supplier<T> supplier;

  private Lazy(Supplier<T> supplier) {
    contents = null;
    this.supplier = supplier;
  }

  public T get() {
    if (contents ≠ null) {
      return contents;
    }

    if (supplier ≠ null) {
      contents = supplier.get();
      supplier = null;
    }

    return contents;
  }
}
```

Lazy lists

A <u>lazy</u> list is:

A head value

A lambda returning another lazy list ("tail-value function" or "tail supplier")

Or:

An Empty list

```java
static <T> LazyList<T> cons(T head, Supplier<LazyList<T>> tailSupplier) {
    return new LazyCons<>(head, tailSupplier);
}
```

Implementing lazy lists (simplified)

```java
class LazyCons<T> implements LazyList<T> {
    final T head;
    final Lazy<LazyList<T>> tail;

    LazyCons(T head, Supplier<LazyList<T>> tail) {
        this.head = head;
        this.tail = Lazy.of(tail);
    }

    public T head() {
        return head;
    }

    public LazyList<T> tail() {
        return tail.get();
    }
```

Implementing lazy lists (simplified)

```java
class LazyCons<T> implements LazyList<T> {
    final T head;
    final Lazy<LazyList<T>> tail;

    LazyCons(T head, Supplier<LazyList<T>> tail) {
        this.head = head;
        this.tail = Lazy.of(tail);
    }

    public T head() {
        return head;
    }

    public LazyList<T> tail() {
        return tail.get();
    }
```

a lambda that will return the tail

Implementing lazy lists (simplified)

```java
class LazyCons<T> implements LazyList<T> {
    final T head;
    final Lazy<LazyList<T>> tail;

    LazyCons(T head, Supplier<LazyList<T>> tail) {
        this.head = head;
        this.tail = Lazy.of(tail);
    }

    public T head() {
        return head;
    }

    public LazyList<T> tail() {
        return tail.get();
    }
}
```

> Build a memo around the tail supplier so that we only call the lambda once

Implementing lazy lists (simplified)

```java
class LazyCons<T> implements LazyList<T> {
    final T head;
    final Lazy<LazyList<T>> tail;

    LazyCons(T head, Supplier<LazyList<T>> tail) {
        this.head = head;
        this.tail = Lazy.of(tail);
    }

    public T head() {
        return head;
    }

    public LazyList<T> tail() {
        return tail.get();
    }
}
```

tail() hides the implementation details

The payoff? Infinite lists!

```java
// Make a LazyList of integers starting from i, skipping by step
public static LazyList<Integer> from(int i, int step) {
    return cons(i,()→from(i+step, step));
}

// Make a LazyList consisting of all the same elements
public static <T> LazyList<T> continually(T s) {
    return cons(s,() → continually(s));
}

var wholeNumbers = from(0, 1); // 0, 1, 2, 3…, runs quickly

var evens = wholeNumbers.filter(x → x % 2 == 0); // 0, 2, 4, 6…, runs quickly

var squares = wholeNumbers.map(x → x * x); // 0, 1, 4, 9…, runs quickly

var zeros = continually(0); // 0, 0, 0, 0…, runs quickly

var alsoEvens = from(0, 2); // runs quickly

var yetAnotherEvens = wholeNumbers.map(x → x * 2); // runs quickly

assertEquals(evens.take(20), alsoEvens.take(20)); // runs pretty quickly

assertEquals(evens.take(20), yetAnotherEvens.take(20)); // runs pretty quickly
```

But be careful...

```
var evens = wholeNumbers.filter(x → x % 2 == 0); // 0, 2, 4, 6...
var alsoEvens = from(0, 2);

assertEquals(evens, alsoEvens); // never finishes!
```

You can't do any operation that requires the entire list!

No length of an infinite list (will never terminate).

You can't fold an infinite list (will never terminate).

You can't test list equality (`.equals()` will go forever as well).

But other operations are just fine.

map, filter, etc.: run in _constant time_, return a new lazy list.

And if you *take()* a finite number of elements from an infinite list, you can do anything with it

fold, length, equality, etc.

```java
class LazyCons<T> implements LazyList<T> {
  public LazyList<T> filter(Predicate<T> predicate) {
    if (predicate.test(headVal)) {
      return cons(headVal, () -> tail().filter(predicate));
    } else {
      return tail().filter(predicate);
    }
  }
}
```

Lazy Map

```
class LazyCons<T> implements LazyList<T> {
  public <R> LazyList<R> map(Function<T, R> f) {
    return cons(f.apply(headVal), ()→tail.get().map(f));
  }
}
```

How about *take()*? That's lazy too!

```
class LazyCons<T> implements LazyList<T> {
  public LazyList<T> take(int n) {
    if (n < 1) {
      return empty();
    } else {
      return cons(headVal, () → tail().take(n - 1));
    }
  }
}
```

fold() cannot be lazy

```java
public <U> U foldRight(U zero, BiFunction<T, U, U> operator) {
  return operator.apply(headVal, tail().foldRight(zero, operator));
}
```

Exact same implementation as *GList*

fold() is a ***terminal*** operation

When to be lazy? When to be eager?

- Laziness almost always wins (in big-O)
  - But the memoization does have a cost.
    - If your lists have millions of entries, this starts to matter.
    - But at that point, maybe you shouldn't be using lists.

- Some programming languages are *extremely* lazy.
  - No value ever computed until it's ultimately needed.
  - Yet still your computation runs efficiently.

- Java is only lazy when you explicitly use lambdas.

Summary

Sometimes you want to be lazy

- If it's too expensive to be eager

- If it's possible that you'll never need the value

- If it's hard to keep track of which values you have already computed

Defer the computation to when (and if) you need it

Very easy to do in functional programming

Map, take, etc. run in **constant** time.

Filter is a bit more complicated, but runs in constant time most of the time

Allows you to create logically infinite data structures

Need to be careful with **terminal** operations (fold, length, etc.) on infinite data structures