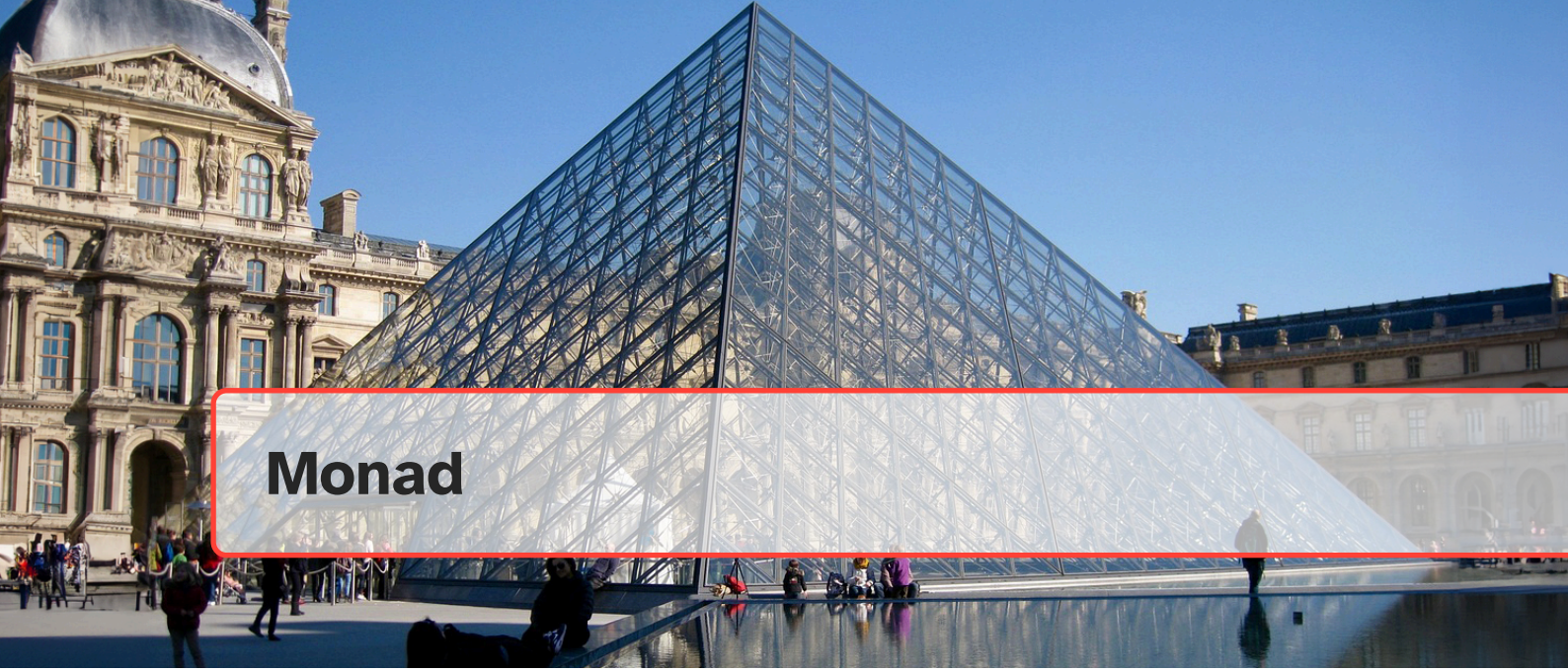


Monad

Preliminary

The following questions are not going to be discussed. Answers is given without explanation. You are encouraged to discuss them on Ed.

1. Consider the `IntMonad` class below. The class encapsulates a single **`int`** value. The implementation of `flatMap` is incomplete.

IntMonad

```
class IntMonad {
  private int v;

  private IntMonad(int v) {
    this.v = v;
  }

  static IntMonad of(int v) {
    return new IntMonad(v);
  }

  IntMonad flatMap(Function<Integer, IntMonad> map) {
    // : different implementation here
  }
}
```

Now, consider the following 3 versions of the possible method body for `flatMap`. For each implementation, determine which of the monad laws[1] are violated and explain. Recap the monad laws.

- Left Identity Law

  ∀ `x`, `f`: `Monad.of(x).flatMap(y -> f(y))` ≡ `f(x)`
- Right Identity Law

  ∀ `m`: `m.flatMap(x -> Monad.of(x))` ≡ `m`
- Associativity Law

  ∀ `m`, `f`, `g`: `m.flatMap(x -> f(x)).flatMap(y -> g(y))` ≡
  `m.flatMap(x -> f(x).flatMap(y -> g(y)))`

For each of the implementation below, represent the result of evaluating each part of the monad laws (i.e., evaluate left-hand side and right-hand side of ≡ independently) as `[v]` if the value is `v`. Our representation of `v` can be a mathematical representation.

(a) `return IntMonad.of(this.v);`

(i) Evaluate the left identity law. Is it violated?
(ii) Evaluate the right identity law. Is it violated?
(iii) Evaluate the associativity law. Is it violated?

(b) `return map.apply(Math.max(0, this.v));`

(i) Evaluate the left identity law. Is it violated?
(ii) Evaluate the right identity law. Is it violated?
(iii) Evaluate the associativity law. Is it violated?

2. What is the returned value for each of the pipelines below?

(a) **Pipeline 1**

```
Stream.of(1, 2, 3, 4)
      .reduce(0, (a, x) -> (2 * a) + x, (a1, a2) -> a1 + a2);
```

(b) **Pipeline 2**

```
Stream.of(1, 2, 3, 4)
      .parallel()
      .reduce(0, (a, x) -> (2 * a) + x, (a1, a2) -> a1 + a2)
```

Replace the accumulator and combiner with the two lambdas below.

Accumulator

```
BiFunction<Integer, Integer, Integer> accumulator = (a, x) -> {
  System.out.println("accumulating: (2 * " + a + ") + " + x);
  return 2*a + x;
}
```

Combiner

```
BinaryOperator<Integer> combiner = (a1, a2) -> {
  System.out.println("combining: " + a1 + " + " + a2);
  return a1 + a2;
}
```

Use your observation to explain the difference in the results above.

Discussion

The following questions are problem sets to be discussed during recitation. You are expected to attempt them before coming to the recitation. You may be randomly called to present your answer during recitation. You are encouraged to discuss them on Ed.

1. Consider the following monad.

Monad<T>

```
import cs2030s.fp.Transformer;

class Monad<T> {
  private T x;

  private Monad(T x) {
    this.x = x;
  }

  public static <T> Monad<T> of(T x) {
```

```
    return new Monad<>(x);
  }

  public T get() {
    return x;
  }

  public <R> Monad<R> flatMap(Transformer<? super T, ? extends Monad<? extends R>> f)
{
    return new Monad<>(f.transform(this.x).get());
  }

  public <R> Monad<R> map(Transformer<? super T, ? extends R> f) {
  return this.flatMap(/* your code here */);
  }
}
```

(a) Complete the implementation of `map` using only `flatMap` so that the resulting `Monad<T>` satisfies the functor laws[2].

(b) Show that the monad `Monad<T>` preserves composition and therefore meets one of the requirements of being a functor.

2. Recall the method `estimatePi` from Lab 0:

Monte Carlo

```
public static double estimatePi(int numOfPoints, int seed) {
  RandomPoint.setSeed(seed);
  Circle c = new Circle(new Point(0.5, 0.5), 0.5);
  int n = 0;
  for (int i = 0; i < numOfPoints; i++) {
    Point p = new RandomPoint(0, 1, 0, 1);
    if (c.contains(p)) {
      System.out.println(p);
      n++;
    }
  }
  return 4.0 * n / numOfPoints;
}
```

We determine whether each random point is contained within the circle independently of other points, and therefore the operation can be parallelized. Rewrite the method using parallel stream so that we estimate the value of π by checking if each point is contained in the given circle in parallel. Try different values of `numOfPoints`. Does parallelization speed up the computation of π?

Note: Due to parallelization, the order of random points processed might be different and thus the estimated π will not be exactly the same as the sequential version.

Exercise

The following questions are to be discussed on Ed as no answer will be given. ***Use only concepts that you have learnt****. You are expected to attempt them before the next lecture as the concepts may be used and/or mentioned in the lecture. The concept may also be tested for assessment.*

1. Recap the `Compute<T>` interface and `summer` method.

Compute<T>

```
interface Compute<T> {
  boolean isRecursive();
  Compute<T> recurse();
  T evaluate();
}
```

Summer Method

```
static long summer(long n) {
  Compute<Long> result = sum(n, 0, "A");
  while (result.isRecursive()) {
    result = result.recurse();
  }
  return result.evaluate();
}
```

The method uses the following modified `sum` method. We added a print to check which version is running.

Sum

```
static Compute<Long> sum(long n, long s, String v) {
  System.out.println(v + ": " + n + " , " + s);
  if (n == 0) {
    return new Base<>(() -> s);
  } else {
    return new Recursive<>(() -> sum(n - 1, n + s, v));
  }
}
```

What we want to do now is to reimplement `summer` sequentially and in parallel. You should try to extend this further to more instances in parallel. First, we create two instances of `Compute<Long>` class with version named `"A"` and `"B"`. Then, we compose the two instances to produce the following behavior.

(a) Implement `summerSequential` to get the following behavior.

Sequential

```
jshell> summerSequential(5)
A: 5 , 0
A: 4 , 5
A: 3 , 9
A: 2 , 12
A: 1 , 14
A: 0 , 15
B: 5 , 0
B: 4 , 5
B: 3 , 9
B: 2 , 12
B: 1 , 14
B: 0 , 15
```

(b) Implement `summerConcurrent` to get the following behavior.

Concurrent

```
jshell> summerConcurrent(5)
A: 5 , 0
B: 5 , 0
A: 4 , 5
B: 4 , 5
A: 3 , 9
B: 3 , 9
A: 2 , 12
B: 2 , 12
A: 1 , 14
B: 1 , 14
A: 0 , 15
B: 0 , 15
```

Bibliography

[1] E. Moggi, “Notions of computation and monads,” *Information and Computation*, vol. 93, no. 1, pp. 55–92, 1991.

[2] S. MacLane, “Categories for the Working Mathematician,” New York: Springer-Verlag, 1971, p. ix+262.