CS2030 Lecture 10

Functional Programming Concepts

Henry Chia (hchia@comp.nus.edu.sg)

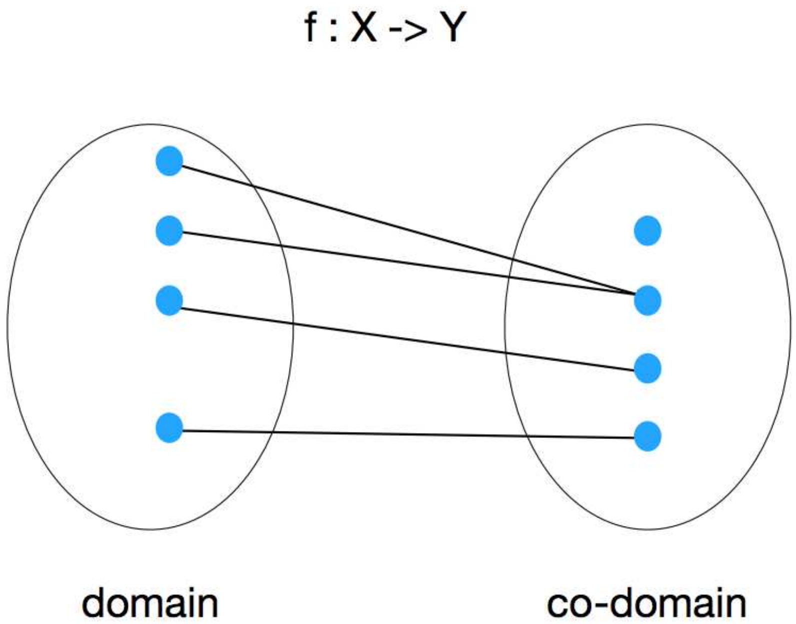
Semester 2 2019 / 2020

Lecture Outline

- Pure function
  - function composition
  - higher order function
  - no side effects
  - cross-barrier state manipulator
- Functor and Monad
  - functor and monad laws

Function

- A *function* is a mapping from a set of inputs X (domain) to a set of outputs Y (co-domain), $f : X \to Y$.
  - Every input in the domain maps to exactly one output
  - Multiple inputs can map to the same output
  - Not all values in the co-domain are mapped



- Function composition, $(f \circ g)(x) = f(g(x))$

Pure Function

- A *pure function* is one that takes in arguments and returns a deterministic value, with no other *side effects*:
  - Program input and output
  - Throwing exceptions
  - Modifying external state
- Absence of side-effects is a necessary condition for referential transparency, i.e. any expression can be replaced by its resulting value, without changing the property of the program
- Are the following functions pure?

```
int p(int x, int y) {
    return x + y;
}

int q(int x, int y) {
    return x / y;
}
```

```
int r(int i) {
    return this.count + i;
}

void s(List<Integer> queue, int i) {
    queue.add(i);
}
```

Higher Order Functions

- Passing functions (e.g. `Function<T,R>` with `R apply(T t)`)

```
jshell> Function<Integer,Integer> f = x -> x + 1
f ==> $Lambda$16/0x0000000800b7840@5e3a8624

jshell> Function<Integer,Integer> g = x -> Math.abs(x) * 10
g ==> $Lambda$17/0x0000000800b7c40@604ed9f0

jshell> f.apply(2)
$.. ==> 3

jshell> int sumList(List<Integer> list, Function<Integer,Integer> f) {
   ...> int sum = 0;
   ...> for (Integer item : list) { sum += f.apply(item); }
   ...> return sum; }
|  created method sumList(List<Integer>,Function<Integer,Integer>)

jshell> sumList(List.of(1, -2, 3), f)
$.. ==> 5

jshell> sumList(List.of(1, -2, 3), g)
$.. ==> 60

jshell> sumList(List.of(1, -2, 3), x -> f.apply(g.apply(x)))
$.. ==> 63
```

- Notice the application of the *abstraction principle*
- Can even be a cross-barrier state manipulator

Pure Functions.. or *Pure Fantasy?*

- Side-effects (necessary evil) should be handled within a *context*

```java
import java.util.List;
import java.util.ArrayList;
import java.util.function.Function;

class IList<T> {
    private final List<T> list;

    IList(List<T> list) {
        this.list = new ArrayList<>();
        for (T item : list) {
            this.list.add(item);
        }
    }

    <R> IList<R> map(Function<T, R> f) {
        ArrayList<R> newList = new ArrayList<>();
        for (T item : list) {
            newList.add(f.apply(item));
        }
        return new IList<R>(newList);
    }

    List<T> get() {
        return list;
    }
}
```

```
jshell> IList<String> list = new IList<>(
   ...> Arrays.asList("abc", "d", "ef"))
list ==> IList@5c3bd550

jshell> list.map(x -> x.length()).get()
$.. ==> [3, 1, 2]
```

- Just like missing values are handled within `Optional`'s context, `IList` handles immutable list mapping within it's own context

Functor

- `Optional`/`IList` are functors (value in a box) with method(s) of the form: `<R> Functor<R> map(Function<T,R> f)`

$$\boxed{\text{c}} \overset{f}{\Rightarrow} \boxed{\text{f(c)}} \text{ where } f \text{ is } x \to f(x), \quad \Rightarrow \text{ denotes the map}$$

- A functor must obey the two functor laws:
  - **Identity**: if f is an identity function $x \to x$, then the resulting functor should be unchanged:

$$\boxed{\text{c}} \overset{f}{\Rightarrow} \boxed{\text{c}} \text{ with } f(x) = x$$

  - **Associative**: if $f = g \circ h$, then the resulting functor should be the same as calling f with h and then with g

$$\boxed{\text{c}} \overset{g \circ h}{\Longrightarrow} \boxed{\text{g(h(c))}} \equiv \boxed{\text{c}} \overset{h}{\Rightarrow} \boxed{\text{h(c)}} \overset{g}{\Rightarrow} \boxed{\text{g(h(c))}}$$

Functor

```
jshell> list.get().equals(list.map(x -> x).get())
$.. ==> true

jshell> Function<String,Integer> f = x -> x.length()
f ==> $Lambda$17/0x00000008000b7440@6a41eaa2

jshell> Function<Integer,Double> g = x -> x * Math.PI
g ==> $Lambda$18/0x00000008000b6840@6093dd95

jshell> list.map(f).map(g).get().equals(list.map(x -> g.apply(f.apply(x))).get())
$.. ==> true
```

- Recall `Circle.getCircle(..).map(c -> c.getPoint())` when trying to map `Optional<Circle>` to `Optional<Point>`
  - Upon mapping, an `Optional<Optional<Point>>` is obtained instead
  - Need a way to merge contexts (in this case, the `Optional`'s context of handling **`null`**/missing values)

Monad

- `Optional` is also a monad (value in a box with some context) with the following methods:
  - `Monad<T> of(T value)` (or simply the `Monad` constructor) that creates the Monad with an empty context
  - `<R> Monad<R> flatMap(Function<T,Monad<R>> f)` that provides the flat-mapping, denoted $\Rightarrow$

$$\boxed{\text{c}} \overset{f}{\Rightarrow} \boxed{\text{f(c)}} \text{ where } f \text{ is } x \rightarrow \boxed{\text{f(x)}}$$

- Different shades represent different degrees of context merging
- Just like functor laws, there are monad laws. Suppose
  - `Monad(x)` gives $\boxed{\text{x}}$, i.e. wraps x with an empty context
  - `monad` is a constant represented by $\boxed{\text{c}}$, i.e. a monad with some fixed value and context

Monad

- Given functions f denoted $x \to \boxed{\text{f(x)}}$, g denoted $x \to \boxed{\text{g(x)}}$, and $g \circ f$ denoted $x \to \boxed{\text{g(f(x))}}$
- **Left identity**: `Monad.of(x).flatMap(f)` $\equiv$ `f.apply(x)`

$$\boxed{\text{x}} \xrightarrow{f} \boxed{\text{f(x)}} \equiv x \to \boxed{\text{f(x)}} \equiv f \text{ ,} \quad \text{outcome is just } f(x)$$

- **Right identity**: `monad.flatMap(x -> Monad.of(x))` $\equiv$ `monad`

$$\boxed{\text{c}} \xrightarrow{f_0} \boxed{\text{c}} \text{ where } f_0 \text{ is } x \to \boxed{\text{x}}$$

- **Associative**: `monad.flatMap(f).flatMap(g)` $\equiv$ `monad.flatMap(x -> f.apply(x).flatMap(g))`

$$\boxed{\text{c}} \xrightarrow{f} \boxed{\text{f(c)}} \xrightarrow{g} \boxed{\text{g(f(c))}} \equiv \boxed{\text{c}} \xrightarrow{g \circ f} \boxed{\text{g(f(c))}}$$

Monad Case Study

```java
class DoubleString {
    Double x;
    String log;

    DoubleString(double x, String log) {
        this.x = x;
        this.log = log;
    }

    static DoubleString of(double x) {
        return new DoubleString(x, "");
    }

    DoubleString flatMap(Function<Double,DoubleString> f) {
        DoubleString ds = f.apply(this.x);
        return new DoubleString(ds.x, this.log + ds.log);
    }

    @Override
    public String toString() {
        return x + ":" + log;
    }
}
```

```java
DoubleString sinAndLog(double x) { return new DoubleString(Math.sin(x), " called sin"); }
DoubleString cubeAndLog(double x) { return new DoubleString(x*x*x, " called cube"); }
```

Monad Case Study

```
jshell> Function<Double,DoubleString> f = x -> sinAndLog(x)
f ==> $Lambda$14/486898233@26be92ad

jshell> Function<Double,DoubleString> g = x -> cubeAndLog(x)
g ==> $Lambda$15/575593575@14acaea5

jshell> f.apply(Double.valueOf(5)) // left identity
$.. ==> -0.9589242746631385: called sin

jshell> DoubleString.of(5).flatMap(f) // left identity
$.. ==> -0.9589242746631385: called sin

jshell> DoubleString c = DoubleString.of(5).flatMap(f) // right identity
c ==> -0.9589242746631385: called sin

jshell> c.flatMap(x -> DoubleString.of(x)) // right identity
$.. ==> -0.9589242746631385: called sin

jshell> c.flatMap(f).flatMap(g) // associative
$.. ==> -0.548496762646804: called sin called sin called cube

jshell> c.flatMap(x -> f.apply(x).flatMap(g)) // associative
$.. ==> -0.548496762646804: called sin called sin called cube
```

Lecture Summary

- Understand the use of functions as cross-barrier state manipulator, as well as facilitating the abstraction principle
- Appreciate how OO and FP complement each other

> *OO makes code understandable by encapsulating moving parts. FP makes code understandable by minimizing moving parts.*
>
> — *Michael Feathers*