CS2030 Lecture 10

Functional Programming Concepts

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

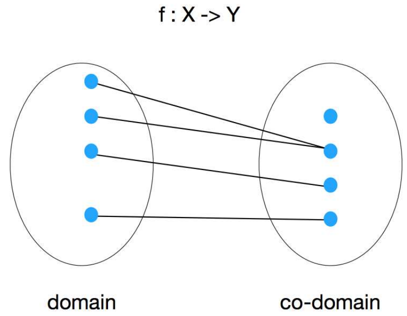
Semester 1 2023 / 2024

Lecture Outline and Learning Outcomes

- Understand the concepts of *effect-free pure functions* and *referential transparency*
- Know how to perform **function composition**
- Appreciate how **currying** supports *partial evaluation*
- Understand how side effects can be handled within *contexts* represented as functors and monads
  - Awareness of the **laws** of functors and monads
- Appreciate that *object-oriented programming* and *functional programming* are *complementary* techniques

Function

- A *function* is a mapping from a set of inputs X (domain) to a set of outputs (range) within a co-domain Y, $f : X \rightarrow Y$.
  - 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



Pure Function

- A *pure function* is a function that
  - takes in arguments and returns a deterministic value
  - is effect-free, i.e. has no other *side effects*
- Examples of side effects:
  - Modifying external state
  - Program input and output
  - Throwing exceptions
- The absence of side-effects is a necessary condition for *referential transparency*
  - any expression can be replaced by its resulting value, without changing the property of the program

Pure Function

- Exercise:
  - Are the following functions pure?

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

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

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

int s(int i) {
    return this.x + i;
}
```


Higher Order Functions

- Functions are first-class citizens
  - Higher-order functions can take in other functions

```
jshell> Function<Integer,Integer> f = x -> x + 1
f ==> $Lambda$../0x00000008000b7840@5e3a8624

jshell> Function<Integer,Integer> g = x -> Math.abs(x) * 10
g ==> $Lambda$../0x00000008000b7c40@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
```


Function Composition

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

```
jshell> Function<String, Integer> f = str -> str.length()
f ==> $Lambda$../731395981@475530b9

jshell> Function<Integer, Circle> g = x -> new Circle(x)
g ==> $Lambda$../650023597@4c70fda8
```

- `Function<T,R>` has a default `andThen` method:

```
default <V> Function<T,V> andThen(
        Function<? super R, ? extends V> after)

jshell> f.andThen(g).apply("abc")
$.. ==> Circle with radius: 3.0
```

- `Function<T,R>` has an alternative default `compose` method:

```
default <V> Function<V,R> compose(
        Function<? super V, ? extends T> before)

jshell> g.compose(f).apply("abc")
$.. ==> Circle with radius: 3.0
```


Function With Multiple Arguments

- Consider the following:

```
jshell> BinaryOperator<Integer> f = (x,y) -> x + y
f ==> $Lambda$../1268650975@2b98378d

jshell> f.apply(1, 2)
$.. ==> 3
```

- We can achieve the same with just `Function<T,R>`

```
jshell> Function<Integer, Function<Integer,Integer>> f = new Function<>() {
   ...>    @Override
   ...>    public Function<Integer,Integer> apply(Integer x) {
   ...>       return new Function<Integer,Integer>() {
   ...>          @Override
   ...>          public Integer apply(Integer y) {
   ...>             return x + y;
   ...>          }
   ...>       };
   ...>    }
   ...> }
f ==> 1@2b98378d

jshell> f.apply(1).apply(2)
$.. ==> 3
```

Currying

- The lambda expression `(x, y) -> x + y` can be re-expressed as `x -> (y -> x + y)` or simply, `x -> y -> x + y`

```
jshell> Function<Integer, Function<Integer,Integer>> f = x -> y -> x + y
f ==> $Lambda$../486898233@26be92ad

jshell> f.apply(1).apply(2)
$.. ==> 3
```

- This is known as **currying**, and it gives us a way to handle lambdas of an arbitrary number of arguments
- Currying supports *partial evaluation*
  - E.g. partially evaluating `f` for increment:

```
jshell> Function<Integer,Integer> inc = f.apply(1)
inc ==> $Lambda$../575593575@46d56d67

jshell> inc.apply(10)
$.. ==> 11
```


Pure Functions.. or *Pure Fantasy?*

- Side-effects are a necessary evil
- Handle side-effects within a *context*, e.g.
  - `Maybe`/`Optional` handles the context of missing values
  - `ImList` handles the context of list processing
  - `Stream` handles the context of loops (and parallel) processing
  - *etc.*
- Values wrapped within contexts that provide the services of
  - `map`: a functor contract, and
  - `flatMap`: a monad contract
- Need to obey the laws of the functor and monad

Exercise: Logging Context

- Define a logging context to log program computations
  - useful for debugging

```
class Log<T> {
    private final T value;
    private final String log;

    private Log(T value, String log) {
        this.value = value;
        this.log = log;
    }

    static <T> Log<T> of(T value) {
        return new Log<T>(value, "");
    }

    static <T> Log<T> of(T value, String log) {
        return new Log<T>(value, log);
    }

    @Override
    public String toString() {
        return "Log[" + this.value + "]: " + this.log;
    }
}
```

```
jshell> Log<Integer> five = Log.<Integer>of(5)
five ==> Log[5]:

jshell> five = Log.<Integer>of(5, "five")
five ==> Log[5]: five
```


`Log` is a Functor with `map`

- `map` method applies the function but does not change the log

```
<R> Log<R> map(Function<? super T, ? extends R> mapper) {
    return new Log<R>(mapper.apply(this.value), this.log);
}
```

- `five.map(x -> x)` $\stackrel{?}{\equiv}$ `five`
- `five.map(f).map(g)` $\stackrel{?}{\equiv}$ `five.map(g.compose(f))`

```
jshell> five.map(x -> x)
$.. ==> Log[5]: five

jshell> Function<Integer,Integer> addOne = x -> x + 1
addOne ==> $Lambda$../0x00007fa01400c208@50040f0c

jshell> Function<Integer,Integer> mulTwo = x -> x * 2
mulTwo ==> $Lambda$../0x00007fa01400c650@4783da3f

jshell> five.map(addOne).map(mulTwo)
$.. ==> Log[12]: five

jshell> five.map(mulTwo.compose(addOne)) // or five.map(x -> mulTwo.apply(addOne.apply(x)))
$.. ==> Log[12]: five
```

- `map` should obey the identity and associativity laws of the Functor

`Log` is a Monad with `flatMap`

- Define the `flatMap` function that combines logs

```java
<R> Log<R> flatMap(Function<? super T, ? extends Log<? extends R>> mapper) {
    Log<? extends R> result = mapper.apply(this.value);
    return Log.<R>of(result.value, this.log + ", " + result.log);
}
```

- Identity function for Monads

```
jshell> Function<Integer, Log<Integer>> identity = x -> Log.of(x)
identity ==> $Lambda$../0x00007fd30000f940@433c675d
```

- Applying the identity function

```
jshell> Log<Integer> five = Log.<Integer>of(5, "5")
five ==> Log[5]: 5

jshell> five.flatMap(identity) // should be the same as above?
$.. ==> Log[5]: 5,

jshell> Function<Integer,Log<Integer>> addOneLog = x -> Log.of(x, "add 1").map(y -> y + 1)
addOneLog ==> $Lambda$../0x00007fd30000a410@70177ecd

jshell> addOneLog.apply(5)
$.. ==> Log[6]: add 1

jshell> identity.apply(5).flatMap(addOneLog) // should be the same as above?
$.. ==> Log[6]: , add 1
```


Identity Laws of the Monad

- Redefine `flatMap` to cater to empty logs

```java
<R> Log<R> flatMap(Function<? super T, ? extends Log<? extends R>> mapper) {
    Log<? extends R> result = mapper.apply(this.value);
    String resultLog = (this.log.isEmpty() || result.log.isEmpty()) ?
        this.log + result.log : this.log + ", " + result.log;
    return Log.<R>of(result.value, resultLog);
}
```

- Right identity law:

```
jshell> Log<Integer> five = Log.<Integer>of(5, "5")
five ==> Log[5]: 5

jshell> five.flatMap(identity) // same as above!
$.. ==> Log[5]: 5
```

- Left identity law:

```
jshell> Function<Integer,Log<Integer>> addOneLog = x -> Log.of(x, "add 1").map(y -> y + 1)
addOneLog ==> $Lambda$../0x00007f3fe800b890@65b3120a

jshell> addOneLog.apply(5)
$.. ==> Log[6]: add 1

jshell> identity.apply(5).flatMap(addOneLog) // same as above!
$.. ==> Log[6]: add 1
```


Associativity Law of the Monad

- Just like the associativity law of the Functor for `map`, there is the associativity law of the Monad for `flatMap`

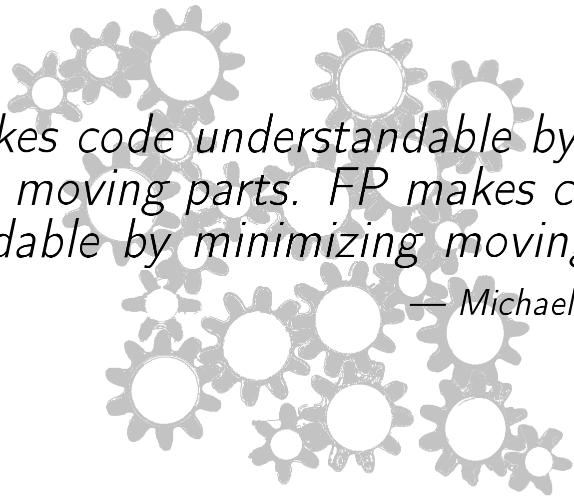
```
jshell> Function<Integer,Log<Integer>> mulTwoLog = x -> Log.of(x, "mul 2").map(y -> y * 2)
mulTwoLog ==> $Lambda$../0x00007fd30000b208@4769b07b

jshell> five.flatMap(addOneLog).flatMap(mulTwoLog)
$.. ==> Log[12]: 5, add 1, mul 2
```

- Above should be equivalent to `five.flatMap(mulTwoLog.compose(addOneLog))`
  - `mulTwoLog.compose(addOneLog)` or `addOneLog.andThen(mulTwoLog)` is a compilation error, since
    - output type of `addOneLog` (a `Log<Integer>`) is not the same as the input type of `mulTwoLog` (an `Integer`)
    - use `x -> addOneLog.apply(x).flatMap(mulTwoLog)` instead
- Notice the equivalence below:

```
jshell> five.flatMap(x -> addOneLog.apply(x).flatMap(mulTwoLog))
$.. ==> Log[12]: 5, add 1, mul 2
```


OOP and FP Are Complementary



OO makes code understandable by encapsulating moving parts. FP makes code understandable by minimizing moving parts.

— Michael Feathers