
CS2030 Lecture 10

Functional Programming Concepts

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

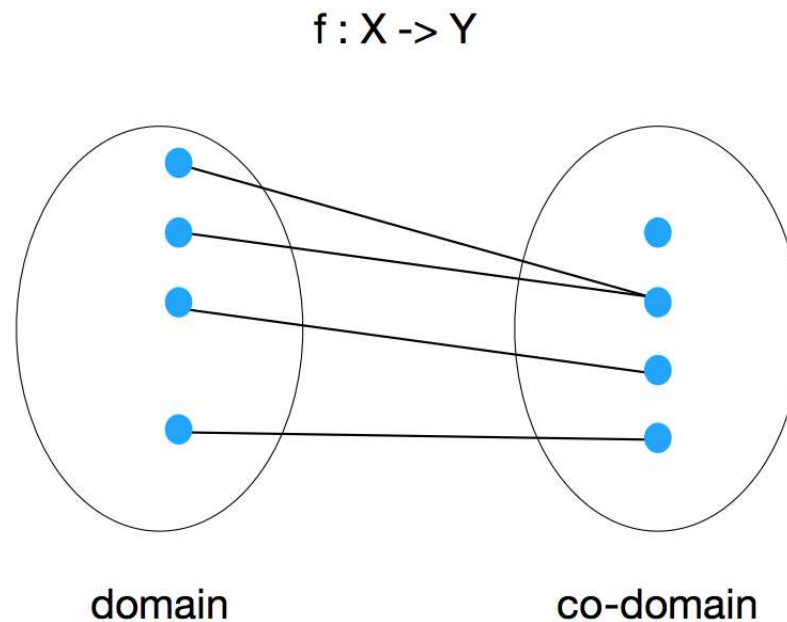
Semester 2 2022 / 2023

Lecture Outline and Learning Outcomes

- Understand the concepts of *referential transparency* and *no-side-effects* in **pure functions**
- Know how to perform **function composition**
- Appreciate how **currying** supports *partial evaluation*
- Understand how side effects can be handled within *contexts* represented as **functors** or **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 Y (co-domain), $f : X \rightarrow Y$.
 - 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
 - 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$16/0x000000008000b7840@5e3a8624
```

```
jshell> Function<Integer,Integer> g = x -> Math.abs(x) * 10  
g ==> $Lambda$17/0x000000008000b7c40@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$14/731395981@475530b9
```

```
jshell> Function<Integer, Circle> g = x -> new Circle(x)  
g ==> $Lambda$15/650023597@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$14/1268650975@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 $(x, y) \rightarrow x + y$ can be re-expressed as $x \rightarrow (y \rightarrow x + y)$ or simply, $x \rightarrow y \rightarrow x + y$

```
jshell> Function<Integer, Function<Integer,Integer>> f = x -> y -> x + y  
f ==> $Lambda$14/486898233@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$15/575593575@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 can be represented by **Functors** (with `map`) or **Monads** (with `flatMap`)
- In the following slides, assume `Functor<T>` and `Monad<T>` are generic interfaces with specific methods to be implemented

Functor

- Functor has a method:

$$\text{<R> Functor<R> map(Function<T,R> f)} \quad \boxed{c} \xRightarrow{x \rightarrow f(x)} \boxed{f(c)}$$

- A functor must obey the two *functor laws*:

- **Identity**: if mapping over an identity function $x \rightarrow x$, then resulting functor should be unchanged:

$$\text{functor.map}(x \rightarrow x) \equiv \text{functor} \quad \boxed{c} \xRightarrow{x \rightarrow x} \boxed{c}$$

- **Associative**: if mapping over $g \circ h$, then the resulting functor should be the same as mapping over h then g

$$\text{functor.map}(g \circ h) \equiv \text{functor.map}(f) \text{.map}(g)$$

$$\boxed{c} \xRightarrow{g \circ h} \boxed{g(h(c))} \equiv \boxed{c} \xRightarrow{h} \boxed{h(c)} \xRightarrow{g} \boxed{g(h(c))}$$

Functor

- Optional is a functor with map

```
jshell> Optional<String> opt1 = Optional.of("abc")  
opt1 ==> Optional[abc]
```

```
jshell> Optional<String> opt0 = Optional.empty()  
opt0 ==> Optional.empty
```

```
jshell> opt1.map(x -> x).equals(opt1) // identity  
$3 ==> true
```

```
jshell> opt0.map(x -> x).equals(opt0) // identity  
$4 ==> true
```

```
jshell> Function<String,Integer> h = x -> x.length()  
h ==> $Lambda$16/1282473384@224edc67
```

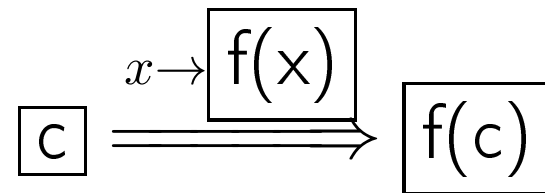
```
jshell> Function<Integer,Integer> g = x -> x * 10  
g ==> $Lambda$17/1188392295@d8355a8
```

```
jshell> opt1.map(g.compose(h)).equals(opt1.map(h).map(g)) // associative  
$.. ==> true
```

```
jshell> opt0.map(g.compose(h)).equals(opt0.map(h).map(g)) // associative  
$.. ==> true
```

Monad

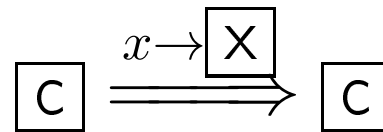
- Monad with the following methods:
 - `Monad<T> of(T value)`, that creates the Monad
 - `<R> Monad<R> flatMap(Function<T,Monad<R>> f)`



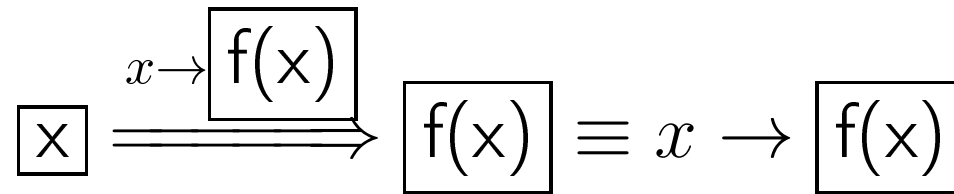
- Just like functor laws, there are monad laws
- In the following slide, suppose
 - `Monad.of(x)` gives \boxed{x} , i.e. wraps x within a context
 - monad is a constant represented by \boxed{c} , i.e. some fixed value wrapped within a context

Monad

- **Right identity:** `monad.flatMap(x -> Monad.of(x)) ≡ monad`

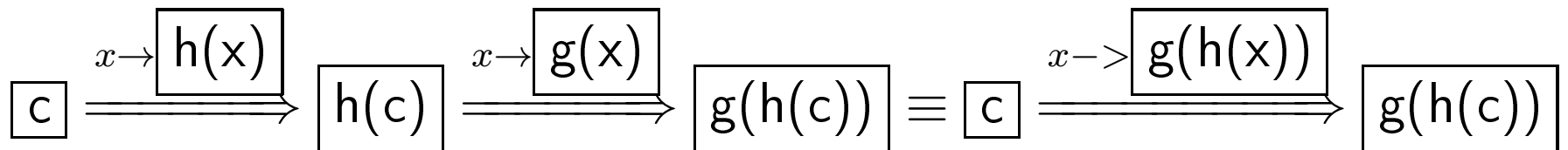


- **Left identity:** `Monad.of(x).flatMap(f) ≡ f.apply(x)`



- **Associative**

`monad.flatMap(h).flatMap(g) ≡ monad.flatMap(x -> h.apply(x).flatMap(g))`



Note the composition $g \circ h$ is expressed as `x -> h.apply(x).flatMap(g)`

Monad

□ Optional is also a Monad

```
jshell> Function<String,Optional<Integer>> f = x -> Optional.of(x.length())
f ==> $Lambda$19/1529306539@61832929

jshell> Optional.of("abc").flatMap(f).equals(f.apply("abc")) // left identity
$.. ==> true

jshell> Function<String,Optional<Integer>> e = x -> Optional.empty()
e ==> $Lambda$20/1582797472@26653222

jshell> Optional.of("abc").flatMap(e).equals(e.apply("abc")) // left identity
$.. ==> true

jshell> Optional<String> opt = Optional.of("monad")
opt ==> Optional[monad]

jshell> opt.flatMap(x -> Optional.of(x)).equals(opt) // right identity
$.. ==> true

jshell> opt = Optional.empty()
opt ==> Optional.empty

jshell> opt.flatMap(x -> Optional.of(x)).equals(opt) // right identity
$.. ==> true
```

OOP and FP Are Complementary

□ *continued...*

```
jshell> Function<Integer,Optional<Integer>> g = x -> Optional.of(x * 10)
g ==> $Lambda$22/670035812@6f7fd0e6

jshell> opt.flatMap(f).flatMap(g).equals(
...>     opt.flatMap(x -> f.apply(x).flatMap(g))) // associative
$.. ==> true
```

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

— Michael Feathers