CS2030 Lecture 8

Towards Functional and Declarative Programming

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

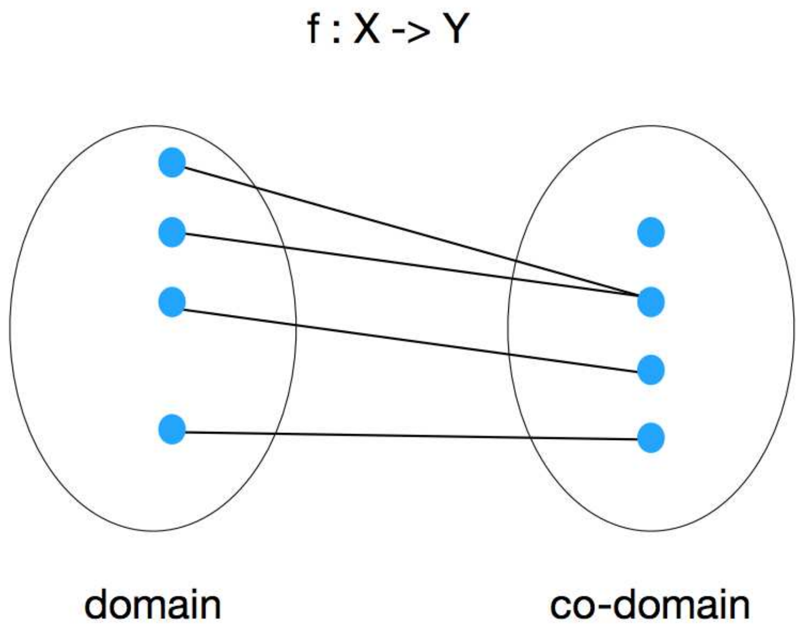
Semester 1 2019 / 2020

Lecture Outline

- Function
  - Pure function with no side effects
  - Functor and Monad
- Internal versus external iteration
- Stream concepts using `IntStream` and `Stream<T>`
  - Stream elements and pipelines
  - Intermediate and terminal operations
  - Stateless vs stateful operations
  - Lazy and eager evaluations
  - `map` and `flatMap`
  - Reduction
  - Infinite streams

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
like fog/ gof

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?

yes

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

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

no

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

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

Java's `Function` Functional Interface

- Why functions? e.g. `R apply(T t)` in `Function<T,R>`

```
jshell> Function<Integer,Integer> f = x -> x + 1
f ==> $Lambda$16/0x00000008000b7840@5e3a8624

jshell> Function<Integer,Integer> g = x -> Math.abs(x) * 10
g ==> $Lambda$17/0x00000008000b7c40@604ed9f0

jshell> f.apply(2)
$3 ==> 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 ==> 5

jshell> sumList(List.of(1, -2, 3), g)
$6 ==> 60
```

- *Abstraction principle*, as well as cross-barrier state manipulator
- Function composition, $(f \circ g)(x) = f(g(x))$

```
jshell> Function<Integer,Integer> fog = x -> f.apply(g.apply(x))
fog ==> $Lambda$18/0x00000008000b6040@5622fdf

jshell> fog.apply(2)
$8 ==> 21
```

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()
$5 ==> [3, 1, 2]
```

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

Functor

- `map` in both `Optional` and `Print` implements the following:

```
interface Functor<T> {
    public <R> Functor<R> f(Function<T,R> func);
}
```

- A functor must also obey the two functor laws
  - if `func` is an identity function `x -> x`, then it should not change the functor
  - if `func` is a composition $g \circ h$, then the resulting functor should be the same as calling f with h and then with g

```
jshell> list.get().equals(list.map(x -> x).get())
$6 ==> 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())
$9 ==> true
```

Monad

- `mapping` works with functors that encloses the same type
- Recall `Circle.getCircle(..).map(c -> c.getPoint())` which maps from `Optional<Circle>` to `Optional<Point>`
- `flatMap` in `Optional` implements the following:

```
interface Monad<T> {
    public Monad<T> of(T value);
    public <R> Monad<R> f(Function<T,Monda<R>> func);
}
```

- Just like functor laws, there are monad laws
  - Left identity: `Monad.of(x).flatMap(f)` $\equiv$ `f.apply(x)`
  - Right identity: `monad.flatMap(x -> Monad.of(x))` $\equiv$ `monad`
  - Associative: `monad.flatMap(f).flatMap(g)` $\equiv$ `monad.flatMap(x -> f.apply(x).flatMap(g))`

External Iteration

- Another example of handling context — that of looping
- An imperative loop specifies **how** to loop and sum

```
int sum = 0;
for (int x = 1; x <= 10; x++) {
    sum += x;
}
```

- Realize the variables `i` and `sum` *mutates* at each iteration
- Errors could be introduced when
  - `sum` is initialized wrongly before the loop
  - looping variable `x` is initialized wrongly
  - loop condition is wrong
  - increment of `x` is wrong
  - aggregation of `sum` is wrong

Internal Iteration

- A *declarative* approach that specifies **what** to do

```
int sum = IntStream
              .rangeClosed(1, 10)
              .sum();
```

- `sum` is assigned with the result of a **stream pipeline**
- Literal meaning "for the range 1 through 10, sum them"
- A **stream** is a sequence of elements on which tasks are performed; the stream pipeline moves the stream's elements through a sequence of tasks
- No need to specify how to iterate through elements or use any *mutatable* variables — no variable state, no problem 😄
- `IntStream` handles all the iteration details

Streams and Pipelines

- A stream pipeline starts with a **data source**
- Static method `IntStream.rangeClosed(1, 10)` creates an `IntStream` containing the ordered sequence $1, 2, \ldots, 9, 10$
  - `range(1, 10)` produces the ordered sequence $1, 2, \ldots, 8, 9$
- Instance method `sum` is the processing step, or **reduction**
  - it reduces the stream of values into a single value
  - Other reductions include `count`, `min`, `max`, `average`

```java
long count = IntStream
    .rangeClosed(1, 10)
    .count();
```

```java
OptionalInt max = IntStream
    .rangeClosed(1, 10)
    .max();
System.out.println(max.getAsInt())
```

- Reductions are **terminal operations** that initiate a stream pipeline's processing so as to produce a result

Mapping

- Most stream pipelines contain **intermediate operations** that specify tasks to perform on a stream's elements before a terminal operation produces a result
- `mapping` is a common intermediate operation
- Using `map`:

```
int sum = IntStream.rangeClosed(1, 10)
    .map(x -> x * 2)
    .sum();
```

- Using `flatMap`

```
int sum = IntStream.rangeClosed(1,5)
    .flatMap(x -> IntStream.rangeClosed(1,x))
    .sum()
```

- Notice the "flattening" effect in `flatMap`

Intermediate and Terminal Operations

- Intermediate operations (like `map`) use **lazy evaluation**
- Does not perform any operations on stream's elements until a terminal operation is called. Using filtering as an example,
  - Select elements that match a condition, or **predicate**

```java
int sum = 0;
for (int x = 1; x <= 10; x++) {
    if (x % 2 == 0) {
        sum += (2 * x);
    }
}
```

```java
int sum = IntStream
    .rangeClosed(1, 10)
    .filter(x -> x % 2 == 0)
    .map(x -> 2 * x)
    .sum();
```

  - `filter` receives a method that takes one parameter and returns a **`boolean`** result; if it is true the element is included in the resulting stream
- Terminal operation use **eager evaluation**, i.e. perform the requested operation when they are called

Stateless vs Stateful Operations

- Intermediate stream operations like `filter` and `map` are **stateless**, i.e. processing one stream element does not depend on other stream elements
- There are, however, **stateful** intermediate operations that depend on the current state
- E.g. stateful operations: `sorted`, `limit`, `distinct`, etc.

```
IntStream
    .of(7, 9, 5, 2, 8, 4, 1, 6, 10, 3)
    .sorted()
    .forEach(System.out::println);

IntStream
    .of(1, 1, 1, 0, 0, 0, 1, 0, 0, 1)
    .distinct()
    .forEach(System.out::println);
```

Method References

- A lambda that simply calls another method can be replaced with just that method's name, e.g. in the `forEach` terminal

```
IntStream
    .rangeClosed(1, 10)
    .forEach(x -> System.out.println(x));
```

- Using method reference

```
IntStream
    .rangeClosed(1, 10)
    .forEach(System.out::println);
```

- Types of method references:
  - reference to a static method
  - reference to an instance method
  - reference to a constructor

Boolean Terminal Operations

- Useful terminal operations that return a **`boolean`** result
  - `noneMatch` returns **`true`** if none of the elements pass the given predicate
  - `allMatch` returns **`true`** if every element passes the given predicate
  - `anyMatch` returns **`true`** if at least one element passes the given predicate
- Example: primality checking using external iteration

```
static boolean isPrime(int n) {
    for (x = 2; x < n; x++) {
        if (n % x == 0) {
            return false;
        }
    }
    return true;
}
```

```
static boolean isPrime(int n) {
    return IntStream
        .range(2, n)
        .noneMatch(x -> n % x == 0);
}
```

Stream Elements

- Each intermediate operation results in a new stream
- Each new stream is an object representing the processing steps that have been specified up to that point in the pipeline
  - Chaining intermediate operations adds to the set of processing steps to perform on each stream element
  - The last stream object contains all processing steps to perform on each stream element
- When *initiating a stream pipeline with a terminal operation*, the intermediate operations' processing steps are applied one stream element after another
- Stream elements within a stream can only be consumed once
  - Cannot iterate through a stream multiple times

Stream Elements

- The following illustrates the movement of stream elements

```
int sum = IntStream
    .rangeClosed(1, 10)
    .filter(
        x -> {
            System.out.println("filter: " + x);
            return x % 2 == 0;
        })
    .map(
        x -> {
            System.out.println("map: " + x);
            return 2 * x;
        })
    .sum();
System.out.println(sum);
```

```
filter: 1
filter: 2
map: 2
filter: 3
filter: 4
map: 4
filter: 5
filter: 6
map: 6
filter: 7
filter: 8
map: 8
filter: 9
filter: 10
map: 10
sum is 60
```

User-defined Reductions

- Using `IntStream`'s `reduce` method
- Terminal operations are specific implementations of `reduce`
- For example, using `reduce` in place of `sum`

```
IntStream
    .of(values)
    .reduce(0, (x, y) -> x + y)
```

  - First argument to `reduce` is the operation's identity value
  - Second argument is the lambda that receives two `int` values, adds them and returns the result; in the above
    - First calculation uses identity value 0 as left operand
    - Subsequent calculations uses the result of the prior calculation as the left operand
    - If stream is empty, the identity value is returned

Infinite Stream

- Lazy evaluation allows us to work with infinite streams that represent an infinite number of elements
- Since streams are lazy until a terminal operation is performed, intermediate operations can be used to restrict the total number of elements in the stream
- `iterate` generates an ordered sequence starting using the first argument as a seed value
- Example, to find the first 500 primes

```
IntStream
    .iterate(2, x -> x + 1)
    .filter(x -> isPrime(x))
    .limit(500)
    .forEach(System.out::println);
```

From IntStream to Stream<T>

- From IntStream to Stream

```
Stream<Circle> circles = IntStream
        .rangeClosed(1, 3)
        .mapToObj(Circle::new); // c -> new Circle(c)

circles.forEach(System.out::println);
```

- There are equivalent intermediate operations in Stream<T>
- Functional interfaces that stream operations take in:
  - Predicate<T> used in filter(Predicate <? super T> predicate)
  - Consumer<T> used in forEach(Consumer <? super T> consumer)
  - Supplier<T> used in generate(Supplier<? extends T> s)
  - Function<T,R> used in map(Function<? super T, ? extends R> mapper)
  - and more...

Stream<T>'s reduce Operator

- Stream<T>'s single-argument reduce method is declared as:

  `T reduce(T identity, BinaryOperator<T> accumulator)`

```
Circle[] circles = {new Circle(1), new Circle(2), new Circle(3)}
Circle newCircle = Stream
    .of(circles)
    .reduce(new Circle(0),
        (c1, c2) -> new Circle(c1.getRadius() + c2.getRadius()))
```

- Overloaded reduce method:

  `Optional<T> reduce(BinaryOperator<T> accumulator)`

```
Stream.of(circles)
    .reduce((c1, c2) -> new Circle(c1.getRadius()
                + c2.getRadius()))
    .ifPresent(System.out::println);
```

- reduce returns an Optional<T> which may have a value, or is empty (e.g. reduction on an empty stream)

Closure

- A lambda expression not only stores the function to invoke, but also data from the enclosing environment

```
class DelayedData {
    private int index;
    private Supplier<Integer> input;

    public DelayedData(int index, Supplier<Integer> input) {
        this.index = index;
        this.input = input;
    }

    public String toString() {
        return index + " : " + input.get();
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        DelayedData[] data = new DelayedData[5];
        for (int i = 0; i < data.length; i++) {
            data[i] = new DelayedData(i, () -> sc.nextInt());
        }
        Stream.of(data)
            .filter(x -> x.index % 2 == 0)
            .forEach(System.out::println);
    }
}
```

Lecture Summary

- Understand the use of functions as cross-barrier state manipulator, as well as facilitating the abstraction principle
- Appreciate the declarative style of programming
- Appreciate how lazy evaluations are used for intermediate operations, eager evaluation for terminal operations
- Know how to define reductions for use in a stream pipeline
- Appreciate how lazy evaluations support infinite streams
- 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*