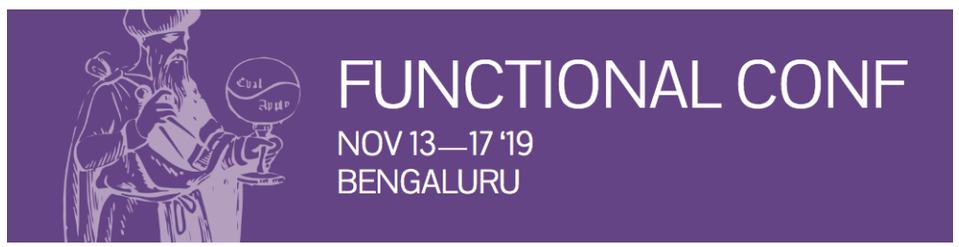
Making Your Program Spicy
Currying
and
Partial Function Application (PFA)

Dhaval Dalal
@softwareartisan
https://dhavaldalal.wordpress.com



14th Nov 2019

Currying

- Refers to the phenomena of rewriting a N-arg function to a nest of functions, each taking only 1-arg at a time

```
// (String, Integer) -> String
String merge(String x, Integer y) {
  return x + y.toString();
}
// String -> (Integer -> String)
Function<Integer, String> merge(String x) {
  return y -> x + y.toString();
}
// () -> (String -> (Integer -> String))
Function<String, Function<Integer, String>> merge() {
  return x -> y -> x + y.toString();
}
// (String -> (Integer -> String))
Function<String, Function<Integer, String>> merge = x -> y -> x + y.toString();
System.out.println(merge("Test#", 1));                  // Test#1
System.out.println(merge("Test#").apply(2));            // Test#2
System.out.println(merge().apply("Test#").apply(3));  // Test#3
System.out.println(merge.apply("Test#").apply(4));    // Test#4
```

Currying

- Curried function is a nested structure, just like Russian dolls, takes one arg at a time, instead of all the args at once.

For each arg, there is another nested function, that takes a arg and returns a function taking the subsequent arg, until all the args are exhausted.

$$f : (T, U, V) \mapsto R$$

$$f' : T \mapsto U \mapsto V \mapsto R$$

$$f = f'$$

Currying

- Function type associates towards right.

```
Function<String, Function<Integer, String>> merge =
  x ->
     y ->
        x + y.toString();
```

- In other words, arrow groups towards right.

```
x -> y -> x + y.toString();

// is same as
x -> (y -> x + y.toString());
```

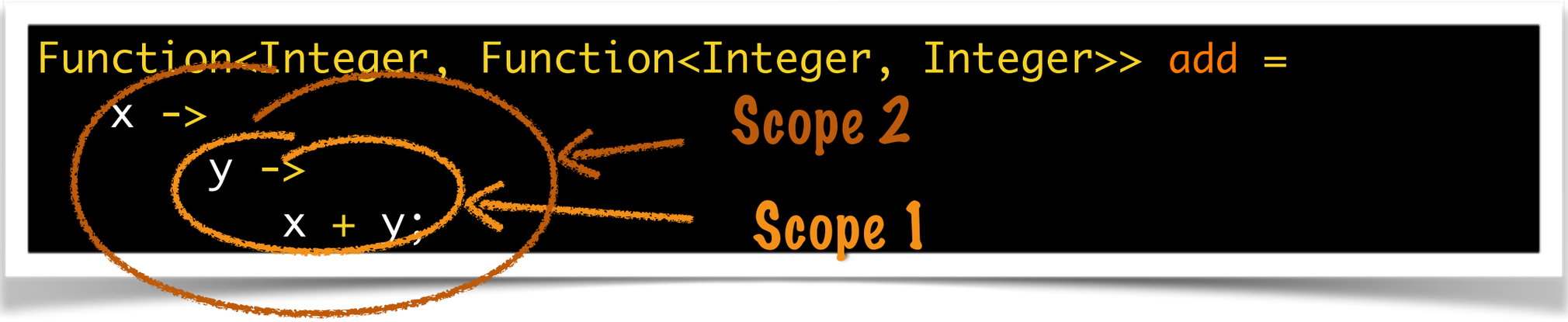
- Function application associates towards left. In other words, apply() groups towards left.

```
merge.apply("Test#").apply(1);

// is same as
(merge.apply("Test#")).apply(1);
```

Why Currying?

- Facilitates arguments to be applied from different scopes.



- Helps us reshape and re-purpose the original function by creating a partially applied function from it

```
Function<Integer, Function<Integer, Integer>> add =
  x ->
     y ->
        x + y;

Function<Integer, Integer> increment = add.apply(1);
increment.apply(2); // 3

Function<Integer, Integer> decrement = add.apply(-1);
decrement.apply(2); // 1
```

But, how does that make programming spicier?

- Let’s say we have a Customer repository.

```
class CustomerRepository {
  Optional<Customer> findById(Integer id) {
    return (id > 0) ? Optional.of(new Customer(id))
                    : Optional.empty();
  }
}
```

- Now, we want to allow authorised calls to repo. So, Let’s write an authorise function.

```
class Authoriser {
  public Optional<Customer> authorise(CustomerRepository repo, Request
request) {
    System.out.println("Authorizing Request#" + request.id);
    // Some auth code here which guards the request.
    // On success, the line below is executed.
    return repo.findById(Integer.parseInt(request.get("customer_id")));
  }
}
```

- We also have a Request.

```
class Request {
  public final int id;
  private Map<String, String> params = new HashMap<>();

  Request(int id) { this.id = id; }
  String set(String key, String value) { return params.put(key, value); }
  String get(String key) { return params.get(key); }
}
```

- Lets see them in action…

```
var authoriser = new Authoriser();
var repo = new CustomerRepository();

var request1 = new Request(1) {{ set("customer_id", "10"); }};

var customer1 = authoriser.authorise(repo, request1);
System.out.println(customer1);

var request2 = new Request(2) {{ set("customer_id", "30"); }};
var customer2 = authoriser.authorise(repo, request2);
System.out.println(customer2);
```

- Requests vary, however the CustomerRepository is same.

- Can we avoid repeated injection of the repo?

Solution 1

- Wrap the authorise function in another function (also called authorise)

```
Optional<Customer> authorise(Request req) {
  CustomerRepository repo = new CustomerRepository();
  return repo.findById(req.get());
}

Authoriser authoriser = new Authoriser();
Request req1 = new Request();
var customer1 = authoriser.authorise(req1);
Request req2 = new Request();
var customer2 = authoriser.authorise(req2);
```

- Requests vary, however the CustomerRepository is same.

- Can we avoid repeated injection of the repo?

Solution 1

- Wrap the authorise function in another function (also called authorise)

```
Optional<Customer> authorise(Request req) {
  CustomerRepository repo = new CustomerRepository();
  return repo.findById(req.get());
}

Authoriser authoriser = new Authoriser();
Request req1 = new Request();
var customer1 = authoriser.authorise(req1);
Request req2 = new Request();
var customer2 = authoriser.authorise(req2);
```

But newification locally like this is untestable!

Solution 2

- Re-shape authorise to accept only one fixed parameter - CustomerRepository

```
class Authoriser {
  public
  Function<Request, Optional<Customer>> authorise(CustomerRepository repo) {
   return req -> {
      //Some auth code here which guards the request.
      repo.findById(req.get());
    }
  }
}

var repo = new CustomerRepository();
Function<Request, Optional<Customer>> curriedAuthorise = authorise(repo);

Request req1 = new Request();
var customer1 = curriedAuthorise.apply(req1);
Request req2 = new Request();
var customer2 = curriedAuthorise.apply(req2);
```

Need some more spice?

Regular DI

```
interface Transaction { }
interface ApprovalStrategy {
  boolean approve(List<Transaction> ts);
  //…
}
class Clearing {
  private final ApprovalStrategy as;

  Clearing(ApprovalStrategy as) {
     this.as = as;
  }

  public boolean approve(List<Transaction> ts) {
    return as.approve(ts);
  }
}
//main
ApprovalStrategy singleMakerChecker = new SingleMakerChecker();
Clearing clearing = new Clearing(singleMakerChecker);
clearing.approve(ts);
```

Curried DI

```
interface Transaction { }
interface ApprovalStrategy {
  boolean approve(List<Transaction> ts);
  //…
}
class Clearing {

  public
  Function<List<Transaction>, Boolean> approve(ApprovalStrategy as) {
    return ts -> as.approve(ts);
  }
}
//main
Clearing clearing = new Clearing();
// ApprovalStrategy can now be injected from different contexts,
// one for production and a different one - say mock for testing,
// Just like in case of Regular DI.
clearing.approve(new SingleMakerChecker()).apply(ts);
```

Making Our Own Curry

Scala calls this curried

```
<T, U, R>
Function<T, Function<U, R>> curry(BiFunction<T, U, R> fn) {
  return t -> u -> fn.apply(t, u);
}

// Function with all args applied at the same time.
BiFunction<Integer, Integer, Integer> add = (x, y) -> x + y;

// Curried Function - one arg at a time.
Function<Integer, Function<Integer, Integer>> cAdd = curry(add);

Function<Integer, Integer> increment = cAdd.apply(1);
increment.apply(2); // 3

Function<Integer, Integer> decrement = cAdd.apply(-1);
decrement.apply(2); // 1
```

It would be nice if Java8 provided this out-of-box on BiFunction

Uncurry or Tuple it!

Scala calls this uncurried

It would be nice if Java8 provided this out-of-box on Function

```
<T, U, R>
BiFunction<T, U, R> uncurry(Function<T, Function<U, R>> fn) {
  return (t, u) -> fn.apply(t).apply(u);
}

// Curried Function - one arg at a time.
Function<Integer, Function<Integer, Integer>> add = x -> y -> x + y;

// Function with all args applied at the same time.
BiFunction<Integer, Integer, Integer> ucAdd = uncurry(add);

ucAdd.apply(2, 3); // 5
```

Currying in Scala

```
scala> def merge1(x: String, y: Int) = x + y.toString
merge1: (x: String, y: Int)String
scala> val merge1F = merge1 _
merge1F: (String, Int) => String

scala> def merge2(x: String)(y: Int) = x + y.toString
scala> val merge2F = merge2 _
merge2F: String => (Int => String)
scala> val mergeTest = merge2F("Test#")
mergeTest: Int => String
scala> mergeTest(1)
res1: Test#1
scala> mergeTest(2)
res2: Test#2
```

- Scala uses special syntax for currying - It does not use tupled arguments, instead uses multiple function call syntax in the definition to denote a curried function.

Curry ⟷ Uncurry

Scala

```
scala> def merge1(x: String, y: Int) = x + y.toString
merge1: (x: String, y: Int)String
scala> val merge1F = merge1 _
merge1F: (String, Int) => String
scala> val merge1FCurried = merge1F.curried
merge1FCurried: String => (Int => String)

scala> def merge2(x: String)(y: Int) = x + y.toString
scala> val merge2F = merge2 _
merge2F: String => (Int => String)
scala> val merge2FUncurried = Function.uncurried(merge2F)
merge2FUncurried: (String, Int) => String
```

PFA in Scala

```
// Partially Applied Function or Partial Function Application (PFA)
def greet(salutation: String, fname: String, lname: String)
  = s"$salutation. $fname $lname"

println(greet("Mr", "Dhaval", "Dalal"))  // Mr Dhaval Dalal

def new_ms = greet("Ms", _: String, _: String)
// (String, String) => String
println(new_ms("Ada", "Lovelace")) // Ms Ada Lovelace

def new_drew = greet(_: String, _: String, "Drew")
(String, String) => String
println(new_drew("Ms", "Nancy"))  // Ms Nancy Drew
```

- Supply a value to partially apply the function and to leave out the arguments unapplied, use underscore “_”

- **Note**: PFA is different from Partial Function

Re-shaping Third Party Library Functions

- In third-party libraries we may never be able to modify the code.
- Use Currying/PFA to simplify the callers interface.

```
def power(n: Double) = Math.pow(_: Double, n)

val square = power(2)
println(square(2)) // 4.0

val cube = power(3)
println(cube(2))  // 8.0
```

Currying in Haskell

```haskell
greet :: String -> String -> String -> String
greet salutation fname lname = salutation ++ ". " ++ fname ++ " " ++ lname

main :: IO ()
main = do
  print $ greet "Mr" "Dhaval" "Dalal"    -- Mr Dhaval Dalal
  let greet_mr = greet "Mr"
  print (greet_mr "Joe" "Shmo")  -- Mr Joe Shmo
  let greet_mr_joe = greet "Mr" "Joe"
  print (greet_mr_joe "Sober")   -- Mr Joe Sober
  print "Done!"
```

- In Haskell, all the functions are curried by default, you don’t need special syntax for that.

- It does not have Partially Applied Functions.

Currying in Haskell

- Even operators are curried by default.

```
Prelude> :type (*)
(*) :: Num a => a -> a -> a
Prelude> multiply2 = (*2)
Prelude> :type multiply2
multiplyBy2 :: Num a => a -> a
Prelude> multiply2 3
6
```

- Yet another example…

```
Prelude> :type filter
filter :: (a -> Bool) -> [a] -> [a]
Prelude> even = \x -> x `mod` 2 == 0
Prelude> filterEvens = filter even
Prelude> :type filterEvens
filterEvens :: Integral a => [a] -> [a]
Prelude> filterEvens [0..10]
[2,4,6,8,10]
Prelude> filterOdds = filter (not . even)
Prelude> filterOdds [0..10]
[1,3,5,7,9]
```

PFA in Clojure

```
(defn new-person [salutation f-name l-name]
  (println salutation f-name l-name))

(new-person "Mr." "Dhaval" "Dalal") ; Mr Dhaval Dalal
```

- Lets partially apply title to this function…

```
(def new-ms (partial new-person "Ms"))
(new-ms "Ada" "Lovelace") ; Ms Ada Lovelace
```

- Lets now partially apply title and first-name

```
(def new-ms-ada (partial new-person "Ms" "Ada"))
(new-ms-ada "Lovelace") ; Ms Ada Lovelace
```

PFA in Clojure

- What if we want to fix the third arg - last name instead of the 1st?

```
(def new-dalal (partial new-person salutation f-name "Dalal"))
(new-dalal "Mr" “Dhaval")
; The above code won't compile
```

- This is because we need placeholder for 1st and 2nd args, which we don’t want to apply., we can’t directly apply the 3rd argument.

- In Clojure, with partial function application the order of arguments matters. Its from left to right. This is just like currying.

- So, what do we do?

Lambda to the rescue

- We wrap this in a lambda!

```
(def new-dalal (fn [sal fname]
    (new-person sal fname "Dalal")))

(new-dalal "Mr." "Dhaval") ; Mr. Dhaval Dalal
```

- Doing this by hand every time is tedious!

- Fortunately, Clojure has a macro to synthesise this kind of "ad-hoc" partial function application

```
(def new-dalal #(new-person %1 %2 "Dalal")) ; Mr. Dhaval Dalal

(new-dalal "Mr." "Dhaval") ; Mr. Dhaval Dalal

(def new-ada #(new-person %1 "Ada" %2))
(new-ada "Ms." "Lovelace") ; Ms Ada Lovelace
(new-ada "Mrs." "???") ; Mrs. Ada ???
```

Currying in Groovy

- In Groovy, we need to explicitly call the curry method on closure!

```groovy
def filterList = { filter, list -> list.findAll(filter) }
def even = { it % 2 == 0 }
def evens = filterList.curry(even)
println evens(0..10) // [0, 2, 4, 6, 8, 10]

def not = { !it }
def odds = filterList.curry(even >> not)  // even andThen not
println odds(0..10) // [1, 3, 5, 7, 9]
```

- It curries from left to right

```groovy
def merge = { x, y, z ->
  println "arg0: x = $x"
  println "arg1: y = $y"
  println "arg2: z = $z"
  "$x$y$z"
}
println merge.curry(10)('Hello', new Date()) // 10Hello<Date>
println merge.curry(10, 'Hello')(new Date()) // 10Hello<Date>
```

Currying in Groovy

- And if you need to curry from right-to-left, use rcurry()

```
def merge = { x, y, z ->
  println "arg0: x = $x"
  println "arg1: y = $y"
  println "arg2: z = $z"
  "$x$y$z"
}
println merge.rcurry(10)('Hello', new Date()) // Hello<Date>10
println merge.rcurry(10, 'Hello')(new Date()) // <Date>10Hello
```

- And for currying at a index, use ncurry()

```
println merge.ncurry(1, new Date())(10, 'Hello')  //10<Date>Hello
println merge.ncurry(0, new Date(), 'Hello')(10)  //<Date>Hello10
```

Reflections

- You curry strictly from left-to-right.
- We **don’t have to provide all the arguments to the function at one go**! This is **partially applying the function**.
- In other words, **Currying enables Partial Function Application**, a.k.a - Partially Applied Function (PFA).
- **"ad-hoc PFA" is practically more powerful than currying in some ways**. With currying, you have to curry strictly from left to right, with PFA you can do ad-hoc application of args
- **In a language that deals well with PFA, you won’t really miss currying**, because you’ll use PFA where you would have used currying.

References

- Code Jugalbandi (http://codejugalbandi.org)
  - Exploring Functional Programming - Ryan Lemmer and Dhaval Dalal
    - https://github.com/CodeJugalbandi/FunctionalProgramming/tree/master/melodies/currying
- Healthy Code Magazine Code Jugalbandi Article
  - https://www.slideshare.net/DhavalDalal/3-code-jugalbandicurryingpfahealthycodemagazinemar2015

Thank - You!

https://github.com/DhavalDalal/FunctionalConference-2019-Currying-PFA-Demo