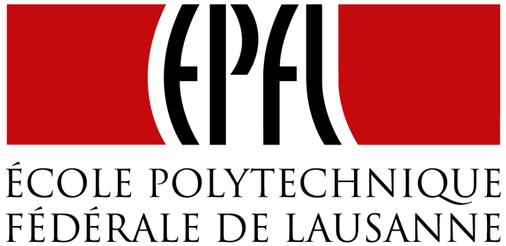
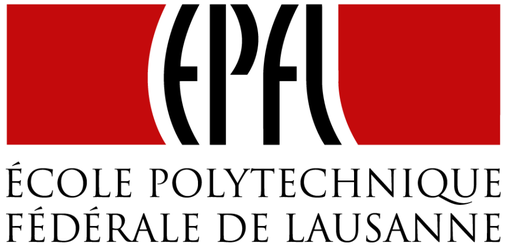


End of Composing Futures (1/2)

Principles of Reactive Programming

Erik Meijer



Composing Futures (2/2)

Principles of Reactive Programming

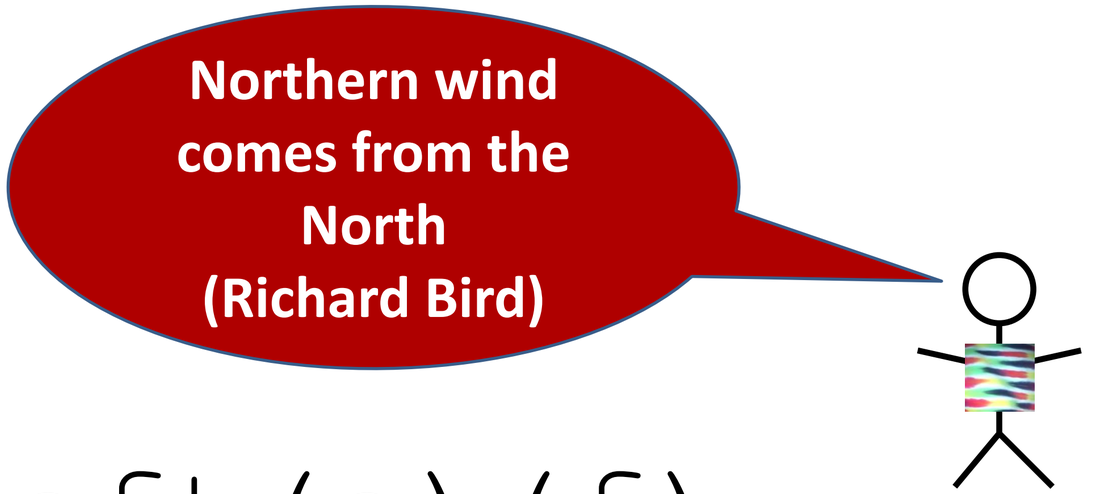
Erik Meijer

Avoid Recursion


```
foldRight
foldLeft
```

Folding lists

```
List(a,b,c).foldRight(e)(f)
=
f(a, f(b, f(c, e)))
```




```
List(a,b,c).foldLeft(e)(f)
=
f(f(f(e, a), b), c)
```

Retrying to send using foldLeft

```
def retry(noTimes: Int)(block: =>Future[T]):
Future[T] = {
  val ns = (1 to noTimes).toList
  val attempts = ns.map(_ => ()=>block)
  val failed = Future.failed(new Exception("boom"))
  val result = attempts.foldLeft(failed)
     ((a,block) => a recoverWith { block() })
  result
}
```

```
retry(3) { block }
= unfolds to
((failed recoverWith {block1()})
   recoverWith {block2()})
     recoverWith { block3 ()}
```

Retrying to send using foldLeft

```
def retry(noTimes: Int)(block: ⇒Future[T]):
Future[T] = {
  …
  val attempts = ns.map(_ => ()=>block)
  …
}

 ns =      List(1,          2,          …, noTimes)
```

Retrying to send using foldLeft

```
def retry(noTimes: Int)(block: ⇒Future[T]):
Future[T] = {
  …
  val attempts = ns.map(_ => ()=>block)
  …
}
```

```
ns =       List(1,          2,          …, noTimes)
attemps = List(()=>block, ()=>block, …, ()=>block)
```

Retrying to send using foldLeft

```
def retry(noTimes: Int)(block: ⇒Future[T]):
Future[T] = {
  …
  val result = attempts.foldLeft(failed)
      ((a,block) => a recoverWith { block() })
  result
}
```

ns = List(1, 2, …, noTimes)

attemps = List(()=>$block_1$, ()=>$block_2$, …, ()=>$block_{noTimes}$)

result = (…((failed recoverWith { $block_1$() }) …) recoverWith { $block_{noTimes}$() })

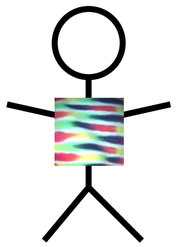
Retrying to send using foldRight

```
def retry(noTimes: Int)(block: =>Future[T])= {
  val ns = (1 to noTimes).toList
  val attempts: = ns.map(_ => () => block)
  val failed = Future.failed(new Exception)
  val result = attempts.foldRight(() =>failed)
    ((block, a) => () => { block() fallbackTo { a()
  result ()
}
```

```
retry(3) { block } ()
= unfolds to
block1 fallbackTo { block2 fallbackTo { block3 fallbackTo
{ failed }}}
```

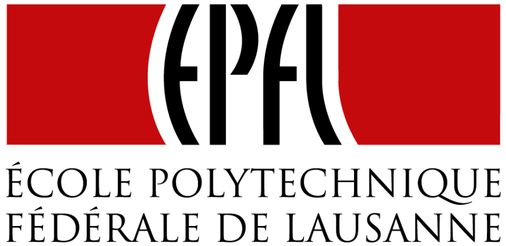
Use Recursion

Often, straight recursion is the way to go



And just leave the HO functions to the FP hipsters!

```
foldRight
foldLeft
```



End of Composing Futures (2/2)

Principles of Reactive Programming

Erik Meijer