Data-Parallel Operations II

Parallel Programming in Scala

Aleksandar Prokopec

Use-cases of the `fold` Operation

Implement the `sum` method:

```
def sum(xs: Array[Int]): Int = {
  xs.par.fold(0)(_ + _)
}
```

Use-cases of the `fold` Operation

Implement the `sum` method:

```
def sum(xs: Array[Int]): Int = {
  xs.par.fold(0)(_ + _)
}
```

Implement the `max` method:

```
def max(xs: Array[Int]): Int
```

Use-cases of the `fold` Operation

Implement the `sum` method:

```
def sum(xs: Array[Int]): Int = {
  xs.par.fold(0)(_ + _)
}
```

Implement the `max` method:

```
def max(xs: Array[Int]): Int = {
  xs.par.fold(Int.MinValue)(math.max)
}
```

Preconditions of the `fold` Operation

Given a list of "paper", "rock" and "scissors" strings, find out who won:

```
Array("paper", "rock", "paper", "scissors")
```

Preconditions of the `fold` Operation

Given a list of "paper", "rock" and "scissors" strings, find out who won:

```
Array("paper", "rock", "paper", "scissors")
  .par.fold("")(play)

def play(a: String, b: String): String = List(a, b).sorted match {
  case List("paper", "scissors") => "scissors"
  case List("paper", "rock")     => "paper"
  case List("rock", "scissors")  => "rock"
  case List(a, b) if a == b      => a
  case List("", b)               => b
}
```

Preconditions of the `fold` Operation

Given a list of "paper", "rock" and "scissors" strings, find out who won:

```
Array("paper", "rock", "paper", "scissors")
  .par.fold("")(play)
```

Preconditions of the `fold` Operation

Given a list of "paper", "rock" and "scissors" strings, find out who won:

```
Array("paper", "rock", "paper", "scissors")
  .par.fold("")(play)
```

```
play(play("paper", "rock"), play("paper", "scissors")) == "scissors"
```

Preconditions of the `fold` Operation

Given a list of "paper", "rock" and "scissors" strings, find out who won:

```
Array("paper", "rock", "paper", "scissors")
  .par.fold("")(play)
```

```
play(play("paper", "rock"), play("paper", "scissors")) == "scissors"
```

```
play("paper", play("rock", play("paper", "scissors"))) == "paper"
```

Why does this happen?

Preconditions of the `fold` Operation

Given a list of "paper", "rock" and "scissors" strings, find out who won:

```
Array("paper", "rock", "paper", "scissors")
  .par.fold("")(play)
```

```
play(play("paper", "rock"), play("paper", "scissors")) == "scissors"
```

```
play("paper", play("rock", play("paper", "scissors"))) == "paper"
```

Why does this happen?

The `play` operator is *commutative*, but not *associative*.

交换 结合

Preconditions of the `fold` Operation

In order for the `fold` operation to work correctly, the following relations must hold:

```
f(a, f(b, c)) == f(f(a, b), c)
f(z, a) == f(a, z) == a
```

We say that the neutral element `z` and the binary operator `f` must form a *monoid*.

Preconditions of the `fold` Operation

In order for the `fold` operation to work correctly, the following relations must hold: 结合率是必须的

```
f(a, f(b, c)) == f(f(a, b), c)
f(z, a) == f(a, z) == a
```

We say that the neutral element `z` and the binary operator `f` must form a *monoid*.

Commutativity does not matter for `fold` – the following relation is not necessary: 可交换不是必须的

```
f(a, b) == f(b, a)
```

Limitations of the `fold` Operation

Given an array of characters, use `fold` to return the vowel count:

Limitations of the `fold` Operation

Given an array of characters, use `fold` to return the vowel count:

```
Array(‘E‘, ‘P‘, ‘F‘, ‘L‘).par
  .fold(0)((count, c) => if (isVowel(c)) count + 1 else count)
```

Limitations of the `fold` Operation

Given an array of characters, use `fold` to return the vowel count:

```
Array('E', 'P', 'F', 'L').par
  .fold(0)((count, c) => if (isVowel(c)) count + 1 else count)
```

Question:

What does this snippet do?

- The program runs and returns the correct vowel count.
- The program is non-deterministic.
- The program returns incorrect vowel count.
- The program does not compile.

Limitations of the `fold` Operation

Given an array of characters, use `fold` to return the vowel count:

```
Array('E', 'P', 'F', 'L').par
  .fold(0)((count, c) => if (isVowel(c)) count + 1 else count)

// does not compile -- 0 is not a Char
```

The `fold` operation can only produce values of the same type as the collection that it is called on.

Limitations of the `fold` Operation

Given an array of characters, use `fold` to return the vowel count:

```
Array('E', 'P', 'F', 'L').par
  .fold(0)((count, c) => if (isVowel(c)) count + 1 else count)

// does not compile -- 0 is not a Char
```

The `fold` operation can only produce values of the same type as the collection that it is called on.

The `foldLeft` operation is *more expressive* than `fold`. Sanity check:

```
def fold(z: A)(op: (A, A) => A): A = foldLeft[A](z)(op)
```

The aggregate Operation

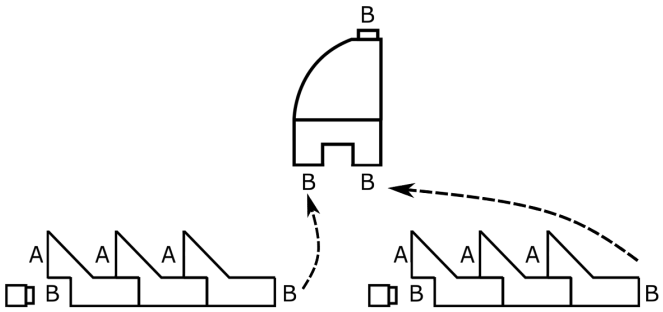
Let's examine the aggregate signature:

```
def aggregate[B](z: B)(f: (B, A) => B, g: (B, B) => B): B
```

The aggregate Operation

Let's examine the `aggregate` signature:

```
def aggregate[B](z: B)(f: (B, A) => B, g: (B, B) => B): B
```



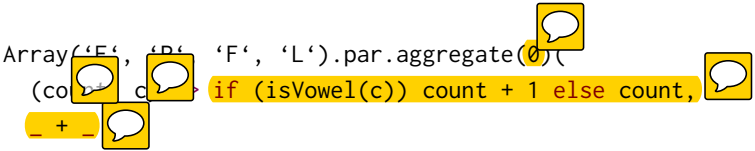
A combination of `foldLeft` and `fold`.

Using the aggregate Operation

Count the number of vowels in a character array:

Using the aggregate Operation

Count the number of vowels in a character array:



```
Array('E', 'R'  'F', 'L').par.aggregate(0)(
  (co   c  => if (isVowel(c)) count + 1 else count,
  _ + _
)
```

The Transformer Operations

So far, we saw the *accessor* combinators.

Transformer combinators, such as `map`, `filter`, `flatMap` and `groupBy`, do not return a single value, but instead return new collections as results.