Splitters and Combiners

Parallel Programming and Data Analysis

Aleksandar Prokopec
Data-Parallel Abstractions

We will study the following abstractions:

▶ iterators
▶ splitters
▶ builders
▶ combiners
Iterator

The simplified *Iterator* trait is as follows:

```scala
trait Iterator[A] { 
    def next(): A
    def hasNext: Boolean
}

def iterator: Iterator[A] // on every collection
```
The simplified Iterator trait is as follows:

```scala
trait Iterator[A] {
    def next(): A
    def hasNext: Boolean
}
```

```scala
def iterator: Iterator[A] // on every collection
```

The *iterator contract*:

- `next` can be called only if `hasNext` returns `true`
- after `hasNext` returns `false`, it will always return `false`
Using Iterators

**Question:** How would you implement `foldLeft` on an iterator?

```scala
def foldLeft[B](z: B)(f: (B, A) => B): B
```
**Question:** How would you implement `foldLeft` on an iterator?

``` scala
def foldLeft[B](z: B)(f: (B, A) => B): B = {
  var s = z
  while (hasNext) s = f(s, next())
  s
}
```
The simplified `Splitter` trait is as follows:

```scala
trait Splitter[A] extends Iterator[A] {
  def split: Seq[Splitter[A]]
  def remaining: Int
}

def splitter: Splitter[A] // on every parallel collection
```
The simplified `Splitter` trait is as follows:

```scala
trait Splitter[A] extends Iterator[A] {
  def split: Seq[Splitter[A]]
  def remaining: Int
}
```

def splitter: Splitter[A] // on every parallel collection

The *splitter contract*:

- after calling `split`, the original splitter is left in an undefined state
- the resulting splitters traverse disjoint subsets of the original splitter
- `remaining` is an estimate on the number of remaining elements
- `split` is an efficient method – $O(\log n)$ or better
Using Splitters

*Question*: How would you implement `fold` on a splitter?

```python
def fold(z: A)(f: (A, A) => A): A
```
Question: How would you implement fold on a splitter?

```scala
def fold(z: A)(f: (A, A) => A): A = {
  if (remaining < threshold) foldLeft(z)(f)
```
Using Splitters

**Question:** How would you implement fold on a splitter?

```scala
def fold(z: A)(f: (A, A) => A): A = {
  if (remaining < threshold) foldLeft(z)(f)
  else {
    val children = for (child <- split) yield task { child.fold(z)(f) }
    children.map(_.join()).foldLeft(z)(f)
  }
}
```
The simplified Builder trait is as follows:

```scala
trait Builder[A, Repr] {
  def +=(elem: A): Builder[A, Repr]
  def result: Repr
}
```

def newBuilder: Builder[A, Repr] // on every collection

The *builder contract*:

- calling `result` returns a collection of type `Repr`, containing the elements that were previously added with `+=`
- calling `result` leaves the `Builder` in an undefined state
**Question:** How would you implement the `filter` method using `newBuilder`?

```scala
def filter(p: T => Boolean): Repr = {
  val b = newBuilder
  for (x <- this) if (p(x)) b += x
  b.result
}
```
Combiner

The simplified Combiner trait is as follows:

```scala
  def combine(that: Combiner[A, Repr]): Combiner[A, Repr]
}

def newCombiner: Combiner[T, Repr] // on every parallel collection
```

The **combiner contract**:

- calling combine returns a new combiner that contains elements of input combiners
- calling combine leaves both original Combiners in an undefined state
- combine is an efficient method – $O(P \cdot \log n)$ or better
Using Combiners

*Question:* How would you implement a parallel filter method using splitter and newCombiner?