Implementing Combiners

Parallel Programming in Scala

Aleksandar Prokopec

Builders

Builders are used in sequential collection methods:

Builders

Builders are used in sequential collection methods:

```
trait Builder[T, Repr] {
  def +=(elem: T): this.type
  def result: Repr
}
```

Combiners

```
trait Combiner[T, Repr] extends Builder[T, Repr] {
  def combine(that: Combiner[T, Repr]): Combiner[T, Repr]
}
```

Combiners

```
trait Combiner[T, Repr] extends Builder[T, Repr] {
  def combine(that: Combiner[T, Repr]): Combiner[T, Repr]
}
```

How can we implement the `combine` method *efficiently*?

Combiners

- when `Repr` is a set or a map, `combine` represents union

Combiners

- when `Repr` is a set or a map, `combine` represents union
- when `Repr` is a sequence, `combine` represents concatenation

Combiners

The `combine` operation must be efficient, i.e. execute in $O(\log n + \log m)$ time, where n and m are the sizes of two input combiners.

Combiners

The `combine` operation must be efficient, i.e. execute in $O(\log n + \log m)$ time, where n and m are the sizes of two input combiners.

Question: Is the method `combine` *efficient*?

```
def combine(xs: Array[Int], ys: Array[Int]): Array[Int] = {
  val r = new Array[Int](xs.length + ys.length)
  Array.copy(xs, 0, r, 0, xs.length)
  Array.copy(ys, 0, r, xs.length, ys.length)
  r
}
```

- Yes.
- No.

Array Concatenation

Arrays cannot be efficiently concatenated.

Sets

Typically, set data structures have efficient lookup, insertion and deletion.

Sets

Typically, set data structures have efficient lookup, insertion and deletion.

- hash tables – expected $O(1)$

Sets

Typically, set data structures have efficient lookup, insertion and deletion.

- hash tables – expected $O(1)$
- balanced trees – $O(\log n)$

Sets

Typically, set data structures have efficient lookup, insertion and deletion.

- hash tables – expected $O(1)$
- balanced trees – $O(\log n)$
- linked lists – $O(n)$

Sets

Typically, set data structures have efficient lookup, insertion and deletion.

- hash tables – expected $O(1)$
- balanced trees – $O(\log n)$
- linked lists – $O(n)$

Most set implementations do not have efficient union operation.

Sequences

Operation complexity for sequences can vary.

Sequences

Operation complexity for sequences can vary.

- mutable linked lists – $O(1)$ prepend and append, $O(n)$ insertion

Sequences

Operation complexity for sequences can vary.

- mutable linked lists – $O(1)$ prepend and append, $O(n)$ insertion
- functional (cons) lists – $O(1)$ prepend operations, everything else $O(n)$

Sequences

Operation complexity for sequences can vary.

- mutable linked lists – $O(1)$ prepend and append, $O(n)$ insertion
- functional (cons) lists – $O(1)$ prepend operations, everything else $O(n)$
- array lists – amortized $O(1)$ append, $O(1)$ random accesss, otherwise $O(n)$

Sequences

Operation complexity for sequences can vary.

- mutable linked lists – $O(1)$ prepend and append, $O(n)$ insertion
- functional (cons) lists – $O(1)$ prepend operations, everything else $O(n)$
- array lists – amortized $O(1)$ append, $O(1)$ random accesss, otherwise $O(n)$

Mutable linked list can have $O(1)$ concatenation, but for most sequences, concatenation is $O(n)$.