Amortized Conc-Tree Appends

Parallel Programming in Scala

Aleksandar Prokopec

Constant Time Appends in Conc-Trees

Let's use Conc-Trees to implement a `Combiner`.

How could we implement += method?

```
var xs: Conc[T] = Empty
def +=(elem: T) {
  xs = xs <> Single(elem)
}
```

Constant Time Appends in Conc-Trees

Let's use Conc-Trees to implement a `Combiner`.

How could we implement += method?

```
var xs: Conc[T] = Empty
def +=(elem: T) {
  xs = xs <> Single(elem)
}
```

This takes $O(\log n)$ time – can we do better than that?

Constant Time Appends in Conc-Trees

To achieve $O(1)$ appends with low constant factors, we need to extend the Conc-Tree data structure.

We will introduce a new `Append` node with different semantics:

```
case class Append[T](left: Conc[T], right: Conc[T]) extends Conc[T] {
  val level = 1 + math.max(left.level, right.level)
  val size = left.size + right.size
}
```

Constant Time Appends in Conc-Trees

One possible `appendLeaf` implementation:

```
def appendLeaf[T](xs: Conc[T], y: T): Conc[T] = Append(xs, new Single(y))
```

Constant Time Appends in Conc-Trees

One possible `appendLeaf` implementation:

```
def appendLeaf[T](xs: Conc[T], y: T): Conc[T] = Append(xs, new Single(y))
```

Can we still do $O(\log n)$ concatenation? I.e. can we eliminate `Append` nodes in $O(\log n)$ time?

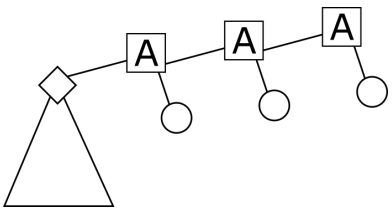
Constant Time Appends in Conc-Trees

One possible `appendLeaf` implementation:

```
def appendLeaf[T](xs: Conc[T], y: T): Conc[T] = Append(xs, new Single(y))
```

Can we still do $O(\log n)$ concatenation? I.e. can we eliminate `Append` nodes in $O(\log n)$ time?

This implementation breaks the $O(\log n)$ bound on the concatenation.



Counting in a Binary Number System

0

$W=2^0$

Counting in a Binary Number System

1

$W=2^0$

Counting in a Binary Number System

$$\begin{array}{c} 1 \\ W=2^0 \end{array} + 1$$

$$\begin{array}{cc} 1 & 0 \\ W=2^1 & W=2^0 \end{array}$$

Counting in a Binary Number System

$$\begin{array}{c} 1 \\ W=2^0 \end{array} + 1$$

$$\begin{array}{cc} 1 & 0 \\ W=2^1 & W=2^0 \end{array} + 1$$

$$\begin{array}{cc} 1 & 1 \\ W=2^1 & W=2^0 \end{array}$$

Counting in a Binary Number System

$$\begin{array}{cc} & 1 \\ + & 1 \\ \hline & \end{array}$$
$W=2^0$

$$\begin{array}{ccc} & 1 & 0 \\ + & & 1 \\ \hline & & \end{array}$$
$W=2^1$ $W=2^0$

$$\begin{array}{ccc} & 1 & 1 \\ + & & 1 \\ \hline & & \end{array}$$
$W=2^1$ $W=2^0$

$$\begin{array}{ccc} 1 & 0 & 0 \end{array}$$
$W=2^2$ $W=2^1$ $W=2^0$

Counting in a Binary Number System

$$\begin{array}{c} 1 \\ W=2^0 \end{array} + 1$$

$$\begin{array}{cc} 1 & 0 \\ W=2^1 & W=2^0 \end{array} + 1$$

$$\begin{array}{cc} 1 & 1 \\ W=2^1 & W=2^0 \end{array} + 1$$

$$\begin{array}{ccc} 1 & 0 & 0 \\ W=2^2 & W=2^1 & W=2^0 \end{array}$$

- To count up to n in the binary number system, we need $O(n)$ work.

Counting in a Binary Number System

$$\begin{array}{cc} & 1 \\ & W=2^0 \end{array} + 1$$

$$\begin{array}{cc} 1 & 0 \\ W=2^1 & W=2^0 \end{array} + 1$$

$$\begin{array}{cc} 1 & 1 \\ W=2^1 & W=2^0 \end{array} + 1$$

$$\begin{array}{ccc} 1 & 0 & 0 \\ W=2^2 & W=2^1 & W=2^0 \end{array}$$

- To count up to n in the binary number system, we need $O(n)$ work.
- A number n requires $O(\log n)$ digits.

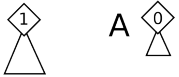
Counting in a Binary Number System

1 + 1

$W=2^0$

1 0 + 1

$W=2^1$ $W=2^0$



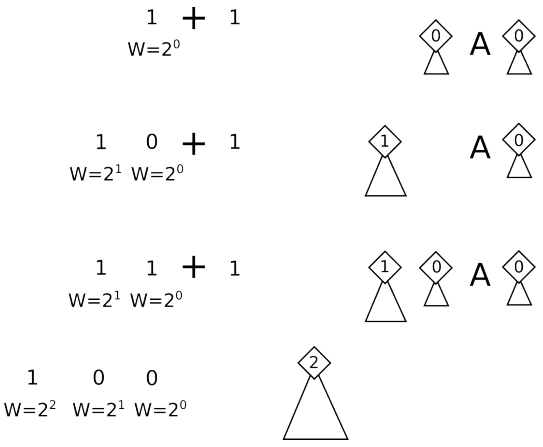
1 1 + 1

$W=2^1$ $W=2^0$

1 0 0

$W=2^2$ $W=2^1$ $W=2^0$

Counting in a Binary Number System



- To add n leaves to an Append list, we need $O(n)$ work.
- Storing n leaves requires $O(\log n)$ Append nodes.

Binary Number Representation

- 0 digit corresponds to a missing tree
- 1 digit corresponds to an existing tree

Constant Time Appends in Conc-Trees

```
def appendLeaf[T](xs: Conc[T], ys: Single[T]): Conc[T] = xs match {
  case Empty => ys
  case xs: Single[T] => new <>(xs, ys)
  case _ <> _ => new Append(xs, ys)
  case xs: Append[T] => append(xs, ys)
}
```

Constant Time Appends in Conc-Trees

```
@tailrec private def append[T](xs: Append[T], ys: Conc[T]): Conc[T] = {
  if (xs.right.level > ys.level) new Append(xs, ys)
  else {
    val zs = new <>(xs.right, ys)
    xs.left match {
      case ws @ Append(_, _) => append(ws, zs)
      case ws if ws.level <= zs.level => ws <> zs
      case ws => new Append(ws, zs)
    }
  }
}
```

Constant Time Appends in Conc-Trees

We have implemented an *immutable* data structure with:

- $O(1)$ appends
- $O(\log n)$ concatenation

Next, we will see if we can implement a more efficient, *mutable* data Conc-tree variant, which can implement a `Combiner`.