SAV – project Report

Stefanos Skalistis

EPFL, 6 June 2015

Binomial Heap

In this project, we tried to develop and verify Binomial Heaps within Leon. Binomial heaps are an important data structure that is used for priorities queues. A nice feature of binomial heaps is that it uses structures, trees and forests, that are defined recursively. Briefly, a binomial heap is a binomial forest (of trees) on which elements are added/deleted. A binomial tree, on the other hand, holds a value and a rank-ordered binomial forest as children.

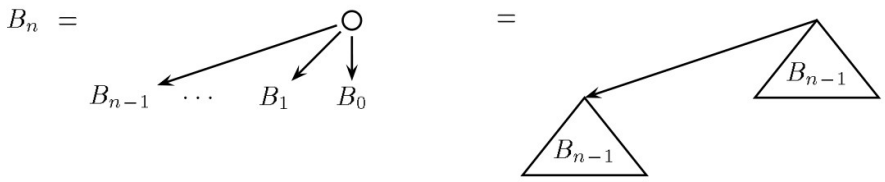


Figure 1. Different representations of BT of rank n.

More specifically, for a binomial tree (BT) its rank is defined recursively; the zero-rank BT holds only a value. A BT of rank *n*, has as children a forest of trees with decreasing (or increasing) ranks, that is from rank *n-1* to *0*. This is shown diagrammatically in Figure 1; the first four BT are also shown in Figure 2.

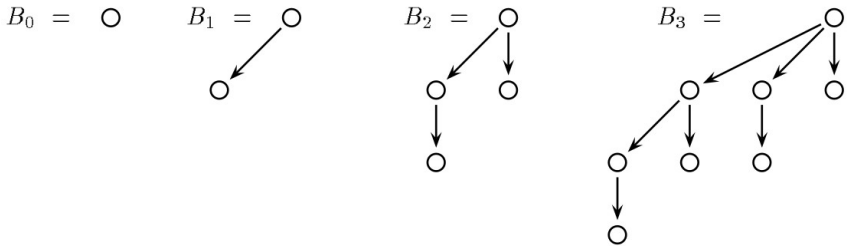


Figure 2. The concrete BT of rank n.

Binomial Trees & Binomial Forests in Leon

It is apparent that this cyclic two-party definition of BT (trees are defined in terms of forests and forests in terms of trees) is going to present challenges for Leon. While forests can be defined as a traditional *List*, we refrained from using lists as there are properties of forests that are not easy to check. This will become clear further on when we discuss the additional required properties. Instead, we define forests in terms of *Cons* but with the extension of functions that allow the check of such properties.

A binomial tree is defined as:

```
case class BinomialTree(forest: BinomialForest, value: BigInt)
```

The *value* field holds the a single value of the tree; when used in the binomial heap this should less than of all the values in the *forest.* It is sufficient to ensure that the *value* is less than the values of the roots of the trees in the forest, due to the recursive definition. The forest of a BT has another important property that should be validated; the forest of rank *n* BT should have trees of decreasing ranks (from *n-1* to 0) and thus the size of the forest should equal to rank *n.*

These are expressed using the *isFull* property of the BinomialForest and the tree lemma:

```
def treeLemma(a :BinomialTree) = {
  require(a.isBT)
  (a.rank >= 0) &&
  (a.rank == a.forest.size) &&
  a.forest.isFull
}.holds
```

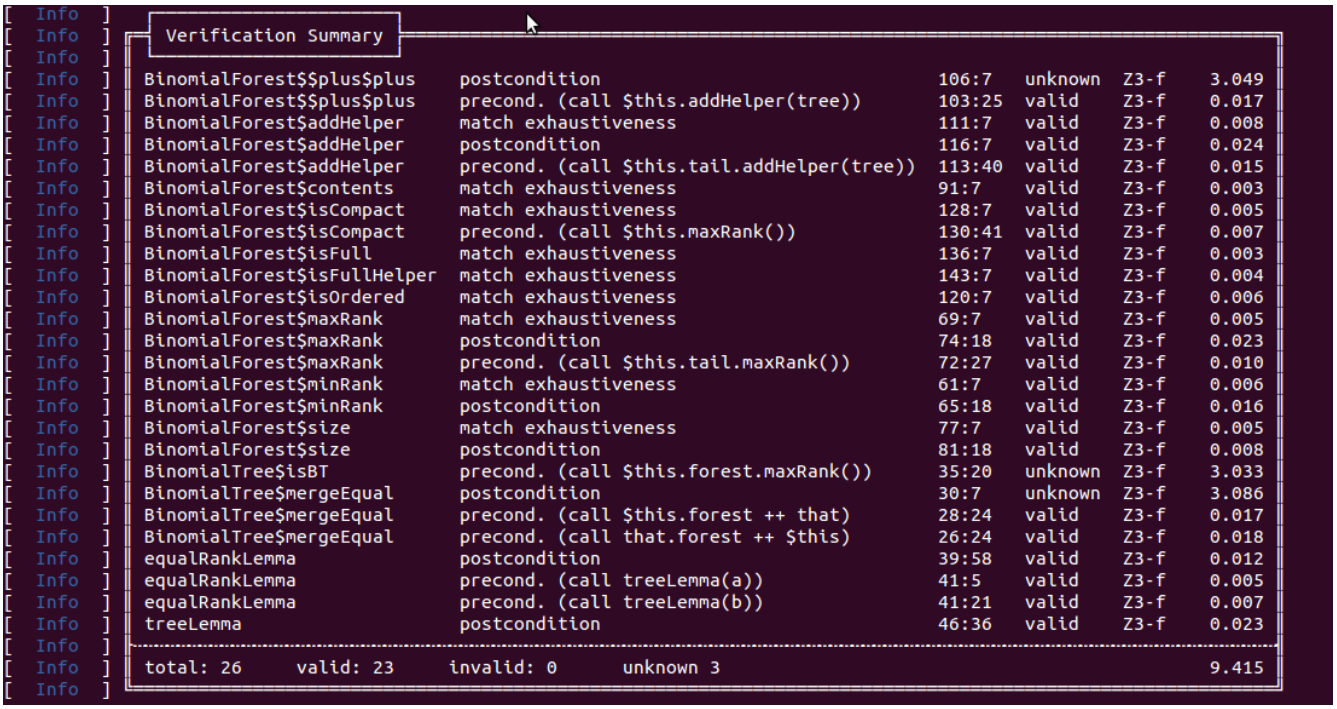
As unusual it may seem, BT do not support addition of values; instead the trivial zero-rank tree is created and merged with others. Merging of BTs is only defined for equal ranked trees, and is performed simply by choosing one of roots (typically the minimum) to be the root of the new high-ranked tree with its forest being composed by the two trees. This is better illustrated in Figure 1 (far-right part).

Forests of BTs require to be always “full” as mentioned earlier; thus only adding a tree with rank exactly one higher than the ones existing is supported. This contrary, to the heap where there is no such restriction. This is enforced by the following properties which are added to verification conditions.

```
def isFull: Boolean = {
  this match {
    case Nil() => true
    case Cons(h, t ) => h.rank == 0 && isFullHelper(h.rank,t)
  }
}
def isFullHelper(rank:BigInt, tail: BinomialForest): Boolean = {
  tail match{
    case Nil() => true
    case Cons(h,t) => h.rank == rank + 1 && isFullHelper(h.rank,t)
  }
}
```

Results & Summary

For the time being only some properties can not verified (but not disproven either with 20 sec timeouts). It is estimated that some lemmas are required to be successfully verified. Given enough time and those properties proven the Binomial heap can be developed. We will proceed in that direction. In the figure below the verification results are displayed (for 3 sec timeout)



References

[1] http://www.cs.cornell.edu/courses/cs312/2005fa/hw/binomial-queues.pdf

[2] *Vuillemin, Jean. "A data structure for manipulating priority queues." Communications of the ACM 21, no. 4 (1978): 309-315.*