Purely Functional Implementation of Optimal Priority Queues in Stainless

Arthur Passuello
EPFL

François Quellec
EPFL

Michaël Spierer
EPFL

Aurélien Debbas
EPFL

Abstract

This report present an attempt at proving the correctness of a Stainless implementation of the Optimal Purely Functional Priority Queues introduced by Chris Okasaki and Gerth Stølting Brodal in Standard-ML. Their model for Priority Queues guarantees optimal asymptotic complexity on their four basic operations: *findMin* in $O(1)$, *insert* in $O(1)$, *meld* in $O(1)$ and *deleteMin* in $O(log(n))$. This report shows how it was possible to translate this model in `Scala` using the *Stainless* library, to prove its correctness by defining invariant conditions on the four operations and helping *Stainless* verify them.

2012 ACM Subject Classification Optimal Purely Functionnal Priority Queues and their implementation in Scala and validation in Stainless

Keywords and phrases Formal Verification, Stainless, Priority Queues, Scala, Binomial Queues, Skew Binomial Queues, Rooted Priority Queues, Bootstrap Priority Queue

1 Introduction

This section first introduces the theoretical background behind this project and the tools that were used. Second, we introduce the paper on which the project is based and finally the approach we took to reach our goal.

1.1 Priority Queue

A *Priority Queue* is an abstract data type which is a special type of Queue. In regular Queues, emplaced elements are served according to some pre-defined policy according to which all elements are treated equally. In a Priority Queue, emplaced elements are associated with a priority and are served according to this priority. Elements with the priority considered the highest are served first and elements sharing the same priority are served as if they belonged to the same regular Queue.[5]

Priority Queues must support basic operations to be used properly (e.g. inserting a new element and retrieving an element). In this paper, we will focus on 4 commonly needed operations[4] :

- *insert*(Q, e) : insert the element e in the Priority Queue Q and returns the new queue
- *findMin*(Q) : returns the minimum element in the queue Q
- *meld*(Q_1, Q_2) : merges the Priority Queues Q_1 and Q_2 and returns the new queue
- *deleteMin*(Q) : removes the minimum element from Q and return the new queue.

1.2 Binomial Trees and Forest

A particularly useful data structure, when implementing Priority Queues, are *Binomial Forest*, which are a forest of *Binomial Trees* as introduced by *Vuillemin*[7].

Binomial trees are composed of *nodes* and *leaves*, each node has a rank $p > 0$ and leaves have a rank $= 0$. Binomial trees are defined inductively and, for a tree B of rank p, the following properties hold:

- B_p has 2^p nodes and there are $\begin{pmatrix} p \\ k \end{pmatrix}$ nodes at depth k in B_p
- in B_p, only the root node has p children and for $0 \leq k < p$, there are 2^{p-k-1} nodes with k children
- the number of children of a node is equal to the number of '1's following the last 0 in its binary numbering
- the children of a node are in decreasing order of rank

Thus, Binomial Trees always contain a number of element that is a power of 2. In order to represent sets of elements of any size, we decompose the set into groups of sizes that are a power of 2 – equivalent to a binary representation of the size of the set. Every '1' in this binary number represents a tree of rank equal to the corresponding power of 2 (e.g. if $|s| = 13$, we have $13 =$ '1101'b, and we will store the set s in 3 trees of rank 3, 2 and 0, containing respectively 8, 4 and 1 elements).

For a Binomial Forest B with n elements and composed of k trees, we have the following :

- every tree in B has different rank, which corresponds to a power of 2 in the binary decomposition of n (and has k '1's)
- the trees are stored in increasing number of rank
- when uniting B with a Binomial Forest S of m elements ($m > 0$), trees of similar rank are merged. The tree with the smallest root becomes the parent of the other. If the result is of the same rank as another tree, the two are merged as well, etc..
- every tree preserves heap order with the values stored in it (i.e. the root is the smallest elements and its children the smallest element of their sub-tree).

This data structure provides a good base for a functional implementation of Priority Queues [2], the next section will introduce their concrete implementation and worst-case complexity regarding the operations introduced in the previous section.

The next sub-section introduces the programming framework of this project.

1.3 Formal Verification & Stainless

"Formal verification methods, which are primarily based on theoretical computer science fundamentals like logic calculi, automata theory and strongly type systems, fulfill these requirements.The main principle behind formal analysis of a system is to construct a computer based mathematical model of the given system and formally verify, within a computer, that this model meets rigorous specifications of intended behavior. Due to the mathematical nature of the analysis, 100% accuracy can be guaranteed."[3]

An existing - and renowned - tool for `Scala` software formal verification is `Stainless`. *Stainless* is able to verify that a *Scala* program is correct for all possible inputs and otherwise

reports inputs for which the program fails. In order to define what "correct" and "fail" mean, specific syntax allows the programmer to specify constraints on the input (e.g. no negative number for a program computing the real root of a real number) and correctness conditions on the output (e.g. the result of a program computing the square of a real number should be positive).

Stainless starts by elaborating a formal definition of the program to be verified, as well as its input's (*pre*) and output (*post*) *conditions*. In order to be able to complete the proof in a finite time, tools are also provided to assist Stainless and guide the process of proving the correctness of the model. More details will be provided as needed in the `Implementation` section.

1.4 Prior work

This project is mainly based on the academic paper *"Optimal Purely Functional Priority Queues"*, by Brodal and Okasaki [2]. This paper introduces a base model for Priority Queues implemented exclusively with binomial queues and offering $O(log(n))$ worst-case complexity on the 4 operations introduced in Section 1.1.

In order to improve these complexities, 3 optimizations are proposed (the details of which will be further discussed in the `Implementation` section) :

1. replacing binomial heaps with *skew* binomial heaps. Making use of the *skew binary* notation to reduce the worst-case complexity of *meld* to $O(1)$.
2. adding a global root stored aside from the rest of the queue to reduce the complexity of *findMin* to $O(1)$.
3. replacing the primitive priority of type T (i.e. a Rooted Skew Binomial Heap of elements T) by a bootstrapped data-structure : a Rooted Priority Queues whose elements are in-turn Rooted Priority Queues of type T.

For all the proposed models and improvements, the paper provides a *Standard-ML* implementation which defines the 4 function's algorithms and that can be quite directly translated into Scala. However, in order to formally verify the correctness of these algorithms in Stainless, one must adapt the underlying data-structures so that Stainless may be able to elaborate a formal model of the queue and the functions. Furthermore, one must also define conditions that faithfully describes the constraints and invariant of the 4 functions' inputs and result.

Thus, this paper aims to propose a Stainless-compatible Scala implementation and verification of the model presented in the aforementioned paper. Our approach is defined in the next paragraph.

Our approach

Roughly, the project can be divided in two phases.

▶ **Step 1.** *Translate Standard-ML code from the paper in* `Scala`*, compatible with Stainless.*

Stainless uses a specific version of Scala, which must be taken into account when translating the algorithms from the reference paper. Furthermore, the data-structure defined to store the queue's elements and represent the Binomial Forest must be defined so that Stainless

may be able to elaborate a model of the functions that use it. Finally, Stainless provides a library with its own data-structures (e.g. *List*) that must be used when available, instead of the standard Scala library. Indeed, the structures in the Stainless library have built-in formal properties that are used to properly elaborate the formal proof.

▶ **Step 2.** *Define the function's pre- and post-conditions and elaborate the proofs using Stainless' library.*

Once the data-structures and functions are properly defined in the Stainless framework, the actual verification may begin. In order to do that, one must first define the minimal conditions on the functions' inputs (e.g. in *insert*(Q, e), Q must be a well defined queue) to ensure proper behavior of the function and the output conditions (e.g. the result from *findMin*(Q) should be the minimal element of Q). Then, if the model is correct, Stainless may still timeout trying to verify it. In these case, "guidance" towards the correct formal proof must be provided through some intermediary lemmas and properties.

2 Implementation

In this section, we will introduce every part of our implementation. First, the data structures and utility functions used to organize and represent the element in the queues. Then, we will introduce every implementation matching the models found in the reference paper. For every implementation, we will start, if need be, with a short introduction of the new theoretical concepts involved. The complete code is available in this repository.

2.1 *Ordering* Trait

Ordering a generic type

Priority Queues implemented in this paper aim to be generic. That is, the type of the elements stored in the queue, used to determine their priority, is generic, we will use the letter T to refer to this type.
Regardless of what this type could concretely be, it needs to provide a way to *order* elements between themselves. In order to provide a generic and verified way to order those elements, we need to define a *Trait* of the same type T that would contain the necessary functions to do so.

Implementation

As the *Ordering Trait* in *Scala* is not compatible with *Stainless* and especially not verified, we had to implement our own based on the paper *Verified Functional Programming* written by *Voirol* [6]. First of all, we start by defining the Trait which consists only of a *compare* function that allows to compare two elements of the same type. Then, in order to prove the correctness of the ordering we define the constraints that this method must satisfy in order for the Ordering trait to be considered valid:

- **inverse:** $\forall$ x, y. sign(compare(x,y)) == -sign(compare(y,x)),
- **transitive:** $\forall$ x, y, z. compare(x,y) $>$ 0 $\wedge$ compare(y,z) $>$ 0 $\Rightarrow$ compare(x,z) $>$ 0,
- **consistent:** $\forall$ x, y, z. compare(x,y) == 0 $\Rightarrow$ sign(compare(x,z)) == sign(compare(y,z)).

Here, the sign (x) is either -1, 0 or 1 depending on the sign of x. Once our law has been defined, Stainless will automatically check that any concrete implementation of Ordering follows these constraints and we can therefore rely on them to verify the code based on Ordering. Below you will find the implementation of the Ordering trait in Scala/Stainless.

```
trait Ordering[T]{
   def compare(x:T,y:T) : Int

   @law def inverse(x:T,y:T) : Boolean =
     sign(compare(x,y)) == -sign(compare(y,x))
   @law def transitive(x:T,y:T,z:T) : Boolean =
     (compare(x,y) > 0 && compare(y,z) > 0) ==> (compare(x,z) > 0)
   @law def consistent(x:T,y:T,z:T) : Boolean =(
     compare(x,y)==0)==>(sign(compare(x,z))==sign(compare(y,z)))

   private final def sign(i:Int) : Int =
   if (i>0) 1 else (if(i<0) -1 else 0)
}
```

Now in order to verify sortedness, we need to define a function to check if a list is sorted. In order to do so we trivially compare each pair of element recursively as demonstrated below.

```
def isSorted[T](l:List[T])(implicit comparator:Ordering[T]):Boolean = {
   decreases(l)
   l match {
     case x::(xs@(y::ys)) => comparator.compare(x,y) <= 0 &&
         isSorted(xs)
     case _ => true
   }
}
```

Note that we need to add the *decreases(l)* line to help stainless understand how the function terminates, indeed in some cases it couldn't infer the measure.

2.2 *Heap* and *Node* structures

Justification

Every queue implemented in this paper stores its element in a Forest of Binomial Trees. In the *Standard-ML* code provided, a rather straightforward definition of the tree is proposed : a tree and every node in it is a tuple *Tree(value, rank, children)*. The *value* is the generic parameter of type T, and the *children* can be viewed themselves as a forest of Binomial Trees. Finally, to represent an empty queue and the absence of children in the leaf nodes, there needs to be a way to represent an empty forest.
The Stainless library contains a generic *List* type which, at a first glance, seems like a good candidate to represent the forests of trees:

```
sealed abstract class List[Tree[T]]
case class Cons[T](h: Tree[T], t: List[Tree[T]]) extends List[Tree[T]]
case class Nil[Tree[T]]() extends List[Tree[T]]
```

However, it became soon apparent that lists were too complex for *Stainless* to formally verify the conditions in which they were involved. Indeed, there is no upper bound on the size of the list, nor on the size of the trees whose complexity grows exponentially w.r.t. their rank. To help Stainless we took great inspiration from the code above and opted to use a structure tailored to our needs, introduced in the next paragraph.

Implementation

A "verified" function in stainless is guaranteed to never crash, however, it can still lead to an infinite evaluation. In order to help stainless determine whether our functions are finite, in particular for the *toList* function we describe in the next section, we decided to use a Heap structure to represent our Binomial Trees as follow.

```
sealed abstract class Heap[+T]
private case class Nodes[+T](head:Node[T], tail:Heap[T]) extends Heap[T]
private case object Empty extends Heap[Nothing]
```

In doing so, a node of a Binomial Tree takes a T-type *Heap* instead of a T-type *List* as a parameter. This structure makes it easy to prove a large number of functions by induction.

```
case class Node[+T](value:T, rank:BigInt = 0, childrens:Heap[T]){
   require(childrens match {
     case Empty => rank == 0
     case Nodes(n, ns) => rank == n.rank+1
   })
}
```

2.3 Serializing Tools

Justification

A formal verification of a system should be independent of its internal behavior. Thus, to properly verify that a Priority Queue's 4 functions correctly operate on its elements, there is a need for a way to represent said content through an external, simple, data structure. Furthermore, two Priority Queues containing the same elements should provide identical results for similar calls on a given operator, and thus should be considered equivalent from outside the structure.
As a result, it was decided to create a set of function that would serialize a Priority Queue of type T into a *sorted* list of type T. This allows to express expected behavior and construct expected results based solely on the stored elements, with no consideration of their organization inside the queue.

Implementation

First, in order to convert our *Priority Queues* bases on *Heap* to *Lists* we had to implement the following two methods for *Node* and *Heap*.

```
def heapToList[T](h : Heap[T]) : List[T] = {
   decreases(size(h))
   h match {
     case Empty => Nil()
     case Nodes(n, ns) => nodeToList(n) ++ heapToList(ns)
   }
}

def nodeToList[T](n : Node[T]) : List[T] = {
   decreases(size(n))
   n match {
     case Node(v, _, h) => List(v) ++ heapToList(h)
```

```
    }
}
```

In some cases *Stainless* could not prove the termination of these two methods, so we helped it by indicating that the size of the Heap decreased with each recursive call, to do this we use simple induction.

```
def size[T](@induct h: Heap[T]): BigInt = {
    h match {
      case Empty => 0
      case Nodes(n, ns) => size(n) + size(ns)
    }
}

def size[T](@induct n: Node[T]): BigInt = {
    1 + size(n.childrens)
}
```

Finally, to be able to compare the results of each methods defined in a Priority Queue, we had to implement a function to sort the generic *Lists*.

```
def SInsert[T](x: T, l: List[T])(implicit comparator: Ordering[T]) :
    List[T] = {
    require(isSorted(l))
    decreases(l)
    l match{
      case Nil() => x::Nil()
      case y::ys if comparator.compare(x,y) <= 0 => x::l
      case y::ys => comparator.inverse(x, y); y::SInsert(x,ys)
    }
 }.ensuring(isSorted(_))

def sort[T](l: List[T])(implicit comparator: Ordering[T]) : List[T] = {
    decreases(l)
    l match{
      case Nil() => Nil[T]()
      case x::xs => SInsert(x,sort(xs))
    }
}.ensuring(isSorted(_))
```

Note that the system needs to know that the *SInsert* function preserves sortedness in order to prove it's correctness. To do that we require that the *List* is already sorted before insertion and use the inverse lemma of *Ordering Trait* to prove that the recursive concatenation is correct.

Implementation

2.4 *Priority Queue* Interface

Reference

The reference paper defines a signature describing a Priority Queue's type, elements and operators' signatures (*isEmpty*, *insert*, *findMin*, *meld* and *deleteMin*). This signature, provide in Figure 1, has been translated into an *abstract class* in the Scala implementation.

```
signature ORDERED =
sig
    type T                          (* type of ordered elements *)
    val leq : T × T → bool          (* total ordering relation *)
end

signature PRIORITY_QUEUE =
sig
    structure Elem : ORDERED

    type T                          (* type of priority queues *)

    val empty     : T
    val isEmpty   : T → bool

    val insert    : Elem.T × T → T
    val meld      : T × T → T

    exception EMPTY

    val findMin   : T → Elem.T      (* raises EMPTY if queue is empty *)
    val deleteMin : T → T           (* raises EMPTY if queue is empty *)
end
```

Figure 1 Signature for Priority Queues.

Furthermore, as any implementation of Priority Queue should show the same behavior from the outside, the operators' *post-conditions* (i.e. expected properties of the operators' results based on their input and the elements present in the queue) are defined in the abstract class. Note that the provided code defines an *Exception* that should be raised when trying to retrieve or remove an element from an empty queue. However, Stainless does not yet have support for exceptions. Thus, it was decided instead to require, as a *pre-condition*, the queue to be non-empty calling such an operator.

Implementation

For the definition of the lemmas we want to prove, we have decided to focus only on the correctness of the implementation and not on the complexity. With the tools we have previously designed, it has become trivial to define these rules by simply comparing **sorted** *Lists*.

```
sealed abstract class PriorityQueue[T]{
   def comp: Ordering[T]

   def isEmpty = (??? : Boolean)
     .ensuring(res => res == this.toList.isEmpty)

   def findMin = {
     require(!this.toList.isEmpty)
     (??? : T)
   }.ensuring(res => res == this.toList.head)

   def insert(x: T) = (??? : PriorityQueue[T])
     .ensuring(res => res.toList == sort(x::this.toList)(this.comp))
```

```
  def meld(that: PriorityQueue[T]) = (??? : PriorityQueue[T])
    .ensuring(res => res.toList == sort(this.toList ++
        that.toList)(this.comp))

  def deleteMin = {
    require(!this.toList.isEmpty)
    (??? : PriorityQueue[T])
  }.ensuring(res => !res.toList.contains(this.findMin) &&
      this.toList.size - 1 == (this.toList & res.toList).size)

  def toList = (??? : List[T])
    .ensuring(res => isSorted(res)(this.comp))
}
```

2.5 Binomial Queue

Reference

As a baseline for their optimized model, Brodal and Okasaki first proposed a Priority Queue based on Binomial Trees, as introduced in Section 1.2. Their algorithms of the four operators they propose, shown here, have been quite directly translated into Scala, with some modifications required for *Stainless* to successfully build the *ADT* Tree.

Implementation

When define Binomial Queues as

```
case class BinomialQueue[T](elems: Heap[T], comparator: Ordering[T])
    extends PriorityQueue[T]
```

Note that we use a Heap[T] instead of a List[T] in order to help stainless (and us) to prove termination and find measures. We then redefine each method from PriorityQueue[T] like in the paper. All our methods implementation are available on our github page.

We find it a little hard to understand at first that *Stainless* oblige us to redefine the *ensuring* notations in the concrete class in addition to the abstract *Priority Queue* class. In other words, it accepted all implementations regardless of the correctness. But once understood it was pretty straightforward.

2.6 Skew Binomial Queue

Skew-Binary

"Skew-binary is a positional system in which the $n-th$ digit has weight $2^n - 1$, using digits 0, 1 and 2. In canonical form only the least significant nonzero digit is allowed to be 2. [...]"[1]

In the context of Binomial Trees, that means the rules have changed. In the standard binary system, a tree of rank p has exactly p children of rank $p-1, p-2, ..., 0$. In the skew-binary system, a tree of rank p may be composed only of two trees of rank $p-1$ or two trees of rank $p-1$ and a tree of rank 0, which may be the root, or not. Figure **??** illustrates this new concept.

Reference

"... Incrementing and decrementing numbers in this system can be done in O(1) since an increment will affect at most the two least significant nonzero digits and not carry through the entire number. "[1]
In their paper, Brodal and Okasaki make use of the skew-binary system to reduce the worst-case complexity of *meld* to a constant. In Section 1.2, we explained that a Forest of Binomial Tree could be seen as the binary number of the number of stored elements. Thus, inserting a new element in a queue (i.e. adding a tree of rank 0 in a forest of trees) could be seen as adding 1 to this binary number. Every induced carry would represent a *link* between two trees and, in the worst case, from a single link in a queue of n elements, there could be $log(n)$ subsequent links induced.
As declared in the above citation, this linking-cascade is no longer possible when using the skew-binary system. Thus, to reduce the worst-case complexity of *inserting* an new element in the Priority Queue, Brodal and Okasaki define a model using the skew-binary system. Note that this modification only concerns the operators used to link two trees, by adding *skewLink*, and the rest of the implementation remains identical to the previous version.

Implementation

We define Skew Binomial Priority Queues similarly to Binomial Queues:

```
case class SkewBinomialQueue[T](elems: Heap[T], comparator:
    Ordering[T]) extends PriorityQueue[T]
```

Since sometimes linking two Nodes is different in Skew Binomial Priority Queues, we had to define a method to do so

```
 def skewLink(q0: Node[T], q1: Node[T], q2: Node[T]): Node[T] = {
  if (comparator.compare(q0.value, q1.value) > 0 &&
      comparator.compare(q2.value, q1.value) > 0)
    Node(q1.value, q1.rank + 1, Nodes(q0, Nodes(q2, q1.childrens)))
  else if (comparator.compare(q0.value, q2.value) > 0 &&
      comparator.compare(q1.value, q2.value) > 0)
    Node(q2.value, q2.rank + 1, Nodes(q0, Nodes(q1, q2.childrens)))
  else
    Node(q0.value, q1.rank + 1, Nodes(q1, Nodes(q2, Empty)))
 }
```

Again, all the SkewBPQ implementation are available here. But as an example, here is the implementation of insert:

```
 def insert(x: T): SkewBinomialQueue[T] = {
  this.elems match {
     case Nodes(q1, Nodes(q2, qs)) if q1.rank == q2.rank =>
         SkewBinomialQueue(Nodes(skewLink(Node(x, 0, Empty), q1, q2),
         qs), this.comp)
     case Nodes(q1, Empty) => SkewBinomialQueue(Nodes(Node(x, 0,
         Empty), Nodes(q1, Empty)), this.comp)
     case Empty => SkewBinomialQueue(Nodes(Node(x, 0, Empty),
         Empty), this.comp)
   }
```

```
    }.ensuring(res => res.toList == sort(x::this.toList)(this.comp))


  def insert_(y: Node[T], nodes: Heap[T]): Heap[T] = {
    decreases(nodes)
    nodes match {
      case Empty => Nodes(y, Empty)
      case Nodes(n, ns) => {
        if (y.rank < n.rank) Nodes(y, Nodes(n, ns))
        else {
        insert_(link(n, y), ns)
        }
      }
    }
  }
```

2.7 Rooted Priority Queue

Reference

So far, Priority Queues are represented as a Forest of Heap ordered Binomial Trees. That is, it is guaranteed that the root of every tree is the minimum of said tree and thus, that the minimum of the queue is one of the roots. However, that implies that, to find this minimum, it requires to scan $log(n)$ elements. In order to reduce this logarithmic complexity, Brodal and Okasaki propose a model that would store the root of the queue (i.e. its minimum) aside from the rest of the queue. With this global root, retrieving the minimum value of the queues trivially amount to returning the value of the root. However, deleting the minimum element now requires to remove the new minimum in the rest of the queue and set it as the global root. This requires $log(n)$ steps to find it (*findMin* from the previous implementation) and then $log(n)$ steps to *meld* its children back into the rest of the queue.

This addition also brings a new issue regarding the type of the queue. Formally written, we had the following so far [2]:

$$\begin{array}{lcl} \text{type } \mathit{Tree} & = & \mathit{Node} \text{ of type } \alpha \times \mathit{Rank} \times \mathit{Tree} \text{ list (children)} \\ P_\alpha & = & \mathit{Tree} \text{ list} \end{array}$$

Thus, an empty queue would be represented by an empty list of trees. With the global root, we have the following :

$$RP_\alpha \quad = \quad \{empty\} \mid (\alpha \times P_\alpha)$$

To represent an empty queue, there is now a need for a distinct type as there is not a proper way to represent an empty value of the generic type T. Fortunately, the *Heap* structure presented in Section 2.2 already has the *Empty* extension that can be exploited in to solve this matter.

Implementation

The Rooted Priority Queue's constructor signature was defined as follows

```
case class RootedPriorityQueue[T](elems: Heap[T], comparator: Ordering[T])
                                  extends PriorityQueue[T]
```

Note that the `Heap` abstract class can be extended 2 ways :

```
private case class  Nodes[+T](head : Node[T], tail : Heap[T])
                              extends Heap[T]
private case object Empty extends Heap[Nothing]
```

Thus, if *elems* is `Empty`, we have an empty queue and, upon insertion of an element or melding with another queue, the result is trivial. If *elems* is a `Nodes`, then the *head* is considered as the root of the queue. On the other hand, *tail* should be a Priority Queue. Thus, at every operation on the Rooted Priority Queue's queue, *tail* is first inserted in a `SkewBinomialQueue` before applying the necessary operation. Then, the elements of the returned *SkewBinomialQueue* are placed in the *tail* variable. An example is provided with the function meld below.

```
def meld(that: PriorityQueue[T]): PriorityQueue[T] = {
    this.elems match{
    case Empty => that
    case Nodes(x1,q1) => that match {
        case RootedPriorityQueue(elems_,_) => elems_ match {
          case Empty => this
          case Nodes(x2,q2) => {
            val pq1 = SkewBinomialQueue(q1,this.comparator)
            val pq2 = SkewBinomialQueue(q2,this.comparator)
            if(comparator.compare(x1.value,x2.value) <= 0){
              RootedPriorityQueue(Nodes(x1,
              pq1.meld(pq2).insert(x2.value).elems),this.comparator)
            }else{
              RootedPriorityQueue(Nodes(x2,
              pq1.meld(pq2).insert(x1.value).elems),this.comparator)
            } } }
        case SkewBinomialQueue(el, _) => {
        el match {
          case Empty => this
          case Nodes(n, ns) => {
            val temp = RootedPriorityQueue(Nodes(Node(that.findMin, 0,
            Empty), that.deleteMin.elems), this.comparator)
            meld(temp)
          } } }
      } }
  }
```

Finally, note that in this implementation, there were several possible candidates with the underlying type of `PriorityQueue` (whereas in the previous implementation, there was only one). Thus, as melding two queues require them to be of the same type, the *meld* function of both `SkewBinomialQueue` and `RootedPriorityQueue` convert *that* to their respective type if necessary.

2.8 Bootstrapped Priority Queue

Reference

In Section 2.6, we introduced a way to insert elements in a Priority Queue in constant time. Based on this improvement, Brodal and Okasaki propose a way to reduce melding two Priority Queues to a simple insertion of one into the other, thus allowing melding in constant

time as well. To do so, the authors introduce the concept of *data-structural bootstrapping*. That is, instead of having a primitive Priority Queue P, α being the type of the stored elements, to construct a Bootstrapped Priority Queue BP_α as a primitive Priority Queue whose elements are other Bootstrapped Priority Queues BP_α :

$$BP = P_{BP_\alpha}$$

However, this configuration does not allow to store simple elements α. Furthermore, it should also be possible to represent an empty queue. As a final configuration, the authors propose :

$$BP_\alpha = \{\text{empty}\} \mid (\alpha \times R_\alpha) \text{ with } R_\alpha = \alpha \times P_{R\alpha}$$

Implementation

A first attempts was to simply define the Bootstrapped Priority Queues' signature as follow :

```
case class BoostrappedSBQ[T](root: Node[T],
                             queue: PriorityQueue[BoostrappedSBQ[T]],
                             comparator: Ordering[T])
                             extends PriorityQueue[T]
case class EmptyBSBQ[T](comparator: Ordering[T]) extends PriorityQueue[T]
```

The functions were then directly translated from the Standard-ML code in the paper. Note that, in order to provide an ordering on elements of type `BootstrappedSBQ[T]]`, we had to concretely implement *compare* method from the trait `Ordering[BootstrappedSBQ[T]]` by using *comparator* to compare the *root* element. As a result, it seemed that stainless was not able to handle recursive relation between *BootstrappedSBQ* and its parameter *queue*. Indeed, when trying to infer the measures of the functions expected to deal with the queue (i.e. the $P_{R\alpha}$ queue), Stainless remained stuck in a supposed infinite loop. A dead-end, then.

Luckily, due to the limitations of the Standard-ML language regarding recursive structures, the authors give a hint on how to "reduce the recursion on structures to recursion on data-types"[2].

As a first attempt to get around this issue, we defined the following abstract class for R_α :

```
sealed abstract class BSBQ[T] extends PriorityQueue[T]{
    val root: Node[T]
    val queue: PriorityQueue[BSBQ[T]]
}
```

Then, we had the following for the queues:

```
case class EmptyBSBQ[T](comparator: Ordering[T],
                        comp2: Ordering[BSBQ[T]])extends PriorityQueue[T]

case class BoostrappedSBQ[T](root: Node[T],
                             queue: PriorityQueue[BSBQ[T]],
                             comparator: Ordering[T],
                             comp2: Ordering[BSBQ[T]]) extends BSBQ[T]
```

Note that this time, the class constructors ask for `Ordering` instances of both `T` and `BSBQ[T]`. After adaption of the queues' body's functions, Stainless seemed to be able to go through some functions' measures. However, in the process of inferring those measures, Stainless suddenly crashed with the cryptic message.

```
"[ Error ] There was an error during the watch cycle"
```

Another dead-end.

Our last attempt took inspiration of the work done in the *Rooted* Priority Queue implementation as well as the hint from the authors and tried to fully displace the recursion on the data-type. We defined the following type :

```
  private case class BootstrappNodes[+T](root: Node[T],
                                         queue: Heap[BootstrappNodes[T]])
                                         extends Heap[T]
```

This structure, used in lieu of R_α, could be defined as $R'_\alpha = (\alpha \times Tree_{R\alpha}\ \text{list})$. Using this, we defined the queue's signature as follows :

```
case class BoostrappedSBQ[T](elems: Heap[T], comparator: Ordering[T])
                            extends PriorityQueue[T]
```

Note that by using the type `Heap`, we can trivially define the empty queue using `Empty` for the *elems* parameter. To obtain the correct type (i.e. R_α) in the queue's function, the parameter *queue* had to first be placed in a Priority Queue - whose type would then not be α but R'_α - before being used. After modifying or accessing its content through a Priority Queue, we would then keep only the `Heap` by selecting only its *elems* parameter. Finally, the ordering for the `BootstrappSBQ` was explicitely defined in the class.

Unfortunately, using a `case class` as the type of a Priority Queue was not supported by Stainless and the compilation returned the error :

```
"[ Error ] Stainless does not support type Product with
                            Serializable"
```

```
def comp2: Ordering[BoostrappedSBQ[T]] = new Ordering[BoostrappedSBQ[T]]{
        def compare(x:BoostrappedSBQ[T],y:BoostrappedSBQ[T]) : Int =
            (x.elems, y.elems) match {
              case (_, Empty) => 1
              case (BootstrappNodes(r1, _), BootstrappNodes(r2, _)) =>
                comparator.compare(r1.value, r2.value)
        }
    }

def meld(that: PriorityQueue[T]): BoostrappedSBQ[T] = that match {
      case b@BoostrappedSBQ(e, c) => {
        (this.elems, e) match {
          case (_ , Empty) => this
          case (BootstrappNodes(r1, q1), BootstrappNodes(r2, q2)) =>
            if(comp.compare(r1.value, r2.value) <= 0)
              BoostrappedSBQ(
                BootstrappNodes(
                  r1,
                  SkewBinomialQueue(q1, this.comp2)
                  .insert(b)
                  .elems),
                comp)
            else
```

```
            BoostrappedSBQ(
              BootstrappNodes(
                r2,
                SkewBinomialQueue(q2, this.comp2)
                .insert(this)
                .elems),
              comp)
        }
      }
    }
```

A final and bitter dead-end, then.

3 Results

The final result of all these implementations is as follows :

PriorityQueue : Although the post conditions declared in the here had to be appended to their respective functions in the concrete implementation of this class, they are the same for every implementations, as initially intended.

BinomialQueue : this implementation runs correctly and has been fully verified

SkewBinomialQueue : this implementation runs correctly and has been fully verified

RootedPriorityQueue : this implementation runs correctly has been fully verified

BootstrappedBQ : no implementation runs without error. No information on the model validity aside from this.

To summarize, all initial goals have been reached. With the exception of the bootstrapped implementation.

4 Discussion

In this section, we will discuss some of the points mentioned in results Section in more details.

4.1 PriorityQueue

This abstract class was defined with the intention to provide a general "type" that all Priority Queues would extend. As a result, there were constraints on the type of the 4 operators defined in it (i.e. *PriorityQueue*). This forced us to carefully match the type of the Priority Queues passed as arguments to and return values by the queue's operators. Finally, when several extension of this class were present (e.g. in *RootedSkewBinomialQueue*), it forced us to handle cases where operators would be applied to different types of Priority Queues.

4.2 BinomialQueue

This first implementation represented the largest amount of work, in retrospect. Indeed, the work to come up with a baseline, and thus facing all the unexpected issue arising from the translation for Standard-ML code as well as from Stainless, made us come up with the `Ordering` trait as well as the `Heap`s classes as solutions. However, once we reached a valid and verified model, we only had to adapt it for the other implementations. For example, *SkewBinomialQueue* only necessitated to modify a couple lines and add some new functions but no deep modifications were required and we were able to obtain a valid model quite fast.

4.3 RootedPriorityQueue

To implement this model, we started with a Priority Queue whose parameter would be a root (i.e. `Node`) and a queue (i.e. `PriorityQUeue`). However, it became quite tedious to find how define an empty queue (due to the fact that we could not define an empty `Node`). At first, we tried to define it as a separate class, similarly to what we did in *BootstrappedSkewBinomialQueue* but it then became too complex for Stainless to be able to verify the model. Our final solution, albeit a sort a "trick" more than anything else, enabled us to come up with a more simple structure regarding the `RootedPriorityQueue` class, despite increasing the complexity and error-proneness of the code.

4.4 BootstrappedPrioritQueue

This implementation was by far the biggest challenge of this project. So big indeed, that despite our extended efforts, we were not able to come up with a solution that would either compile within the Stainless framework, or be handled by Stainless at runtime. Several quite different approaches were explored and each of ended without satisfying results. However, we were still able to understand a couple things.
First, it seems that Stainless does not handle well the verification of a class that has itself in its parameters. Indeed, this approach implements the recursion on the queue structure itself and, when Stainless tried to measure the functions from `SkewBinomialQueue` (which had the type `BootstrappedPriorityQueue` instead of `T`), it entered in an infinite a noisy loop. It remains possible, however, that our way of expressing this structural recursion was ill-defined
Second, it seems that Stainless does not accept `case class`' as primitive generic types. Indeed, as detailed in the last attempt of Section 2.8, the promising approach of displacing the recursion on the data-type of the queues instead of the queues themselves was simply not compatible with Stainless.

5 Future work

As mentioned in the last attempt described in Section 2.8, Stainless came up with an error that prevented us from moving forward as it doesn't support certain types of classes such as the one we created for the bootstrapped queues. To be more precise, the error message was 'Stainless does not support type Product with Serializable' so it does not support case class as a generic type of a case class and this is exactly what we had to do for Bootstrap Queues.

Something that could be said more generally is that Stainless is a very complex tool and in order to find some optimal solutions or to understand it better, it would be necessary to take a deep look at Stainless code. Unfortunately, in this project, we didn't have the time to go and look at the Stainless core code in order to fully understand it.

6 Conclusions

This project is nothing but a modest introduction to the formal verification of complex, purely functional data structures such as the ones proposed by Brodal and Okasaki. The process through which our thinking had to go from the proposed model to our solution, passing through many interpretations and attempts at recreating our references has however been quite insightful regarding our approach to both functional data-structures, and formal verification.

Finally, it is true to say that this project undoubtedly ended with a bitter failure, due to the current limitations of Stainless and our understanding of it. However, this could be interpreted as a suggestion to look around for other formal verification tools, such as *Coq*, and try our luck with them.

References

1 A169683. URL: https://oeis.org/A169683.

2 Gerth Stølting Brodal and Chris Okasaki. Optimal Purely Functional Priority Queues. *Basic Research in Computer Science*, RS-96-37, 1996. doi:https://doi.org/10.7146/brics.v3i37.20019.

3 Osman Hasan and Sofiène Tahar. Formal verification methods. In *Encyclopedia of Information Science and Technology, Third Edition*, pages 7162–7170. IGI Global, 2015. doi:10.4018/978-1-4666-5888-2.ch705.

4 K. Mehlhorn and A. Tsakalidis. Handbook of theoretical computer science, volume a: Algorithms and complexity. 1990.

5 Rupert G. Miller. Priority queues. *The Annals of Mathematical Statistics*, 31(1):86–103, 1960. URL: http://www.jstor.org/stable/2237496.

6 Nicolas Charles Yves Voirol. *Verified Functional Programming*. PhD thesis, Lausanne, EPFL, February 2019.

7 Jean Vuillemin. A data structure for manipulating priority queues. *Communication of the ACM*, 21(4), 1978.