Formal verification of purely skiplists with arbitrary many levels

Tanguy Rocher
tanguy.rocher@epfl.ch

Hugo Lepeytre
hugo.lepeytre@epfl.ch

Paul Juillard
paul.juillard@epfl.ch

January 7, 2022

Abstract

The Stainless framework is applied to formally verify the correctness of a minimal functional implementation in Scala of a skiplist object. This implementation is not production worthy, but can rather be seen as a Stainless use-case. Our results and method are compared to previous work on imperative implementations. Difficulties, limits and take-aways from this exercise are explored. All resources are public on github.

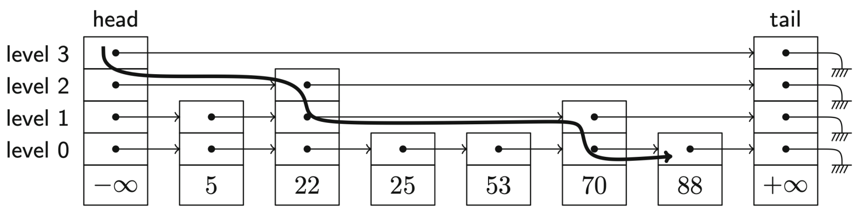


Figure 1: A skiplist with 4 levels and the traversal searching 88 [4]

1 Introduction

A Skiplist is a relatively simple data-structure to store ordered sets. It is of importance because they allow for an easier implementation of sets than balanced trees while having comparable performance. Their main uses leverage their logarithmic time search in combination with their locality properties, enabling efficient concurrent modifications. Formal verification has been achieved for certain implementations, in particular in [4] which presents a Presburger Arithmetic language for imperative skiplist implementations. Notably, it allows verification of skiplists of arbitrary height, whereas the previous verification algorithms did not scale beyond 3 levels. The construction of this verification framework and algorithm needs well-thought formal definitions of signatures and other Presburger Arithmetic constructs that are then translated to Coq code and verified.

In the following, two alterations are made to the aforementioned process. The first is to move from an imperative implementation to a functional implementation. Formal verification for functional skiplists has not been published, this work explores its feasibility. Secondly, we use Stainless, a verification framework for Scala code, in order to verify this implementation. This moves away from the weight of verbose formal definitions and equations of the mentioned and more widespread method, towards a code-oriented approach. This allows to explore the benefits and drawbacks of using Stainless or likewise frameworks compared to the Coq-translated formal expressions method. After preliminary definitions, the proposed implementation for a functional skiplist is presented, followed by its correctness verification and discussion on the process.

2 Preliminaries

2.1 Skiplists

A skiplist is a data structure consisting of singly-linked ordered lists organised as levels : The list at level 0 contains all elements in the set while every level $i > 0$ contains a subset of the elements at level $i - 1$. Every element at

level i links to the next element on level i as well as to itself on level $i-1$. With the structure in mind, 3 main operations are provided : insert, search and remove. The detail of these operations are implementation dependent. As an example, the data structure and search operation are represented in Figure 2.1.
Properties for skiplists are the following:

- elements of a skiplist are ordered
- at all levels, the first element is the special *head* element
- the list of elements at level i is a subset of the list of elements at level $i-1$
- the search operation has logarithmic complexity in the number of elements in the list

Note that as in [4], there is no complexity requirements for insert and remove operations. The following implementation is thus happy to trade operation complexity for simplicity of verification.

2.2 Stainless

Stainless is a verification framework for a subset of the Scala programming language. Programmers can embed the verification of operations along their definitions. An example use for a functional implementation of the factorial function is reproduced from the documentation [1]:

```scala
def factorial(n: BigInt): BigInt = {
  require(n >= 0)
  if(n == 0) {
    BigInt(1)
  } else {
    n * factorial(n - 1)
  }
} ensuring(res => res >= 0)}
```

The core of the function is written in usual Scala code. In addition, Stainless functions `require` and `ensuring` show expected preconditions and post-conditions for this function, namely that the factorial operation is defined for natural integers, and that it's result is positive respectively. Running Stainless on this piece of code verifies that under the stated preconditions, the post-condition holds, and that the operation terminates.
Compared to other methods, the use of Stainless for code verification already shows two advantages for tentative provers such as us:

- desired properties for Scala code are also written as Scala code

- the code verification is embedded in the code, which reduces the back-and-forth between a formal verification model and the implementation's code

Concretely, Stainless can be used by the programmer to formally verify their code, whilst keeping their programmer-cap on, without the need to wear the logician-cap. This does not imply that the process is more efficient in any way yet, but rather that it may seem more approachable for programmers.

2.3 Previous works

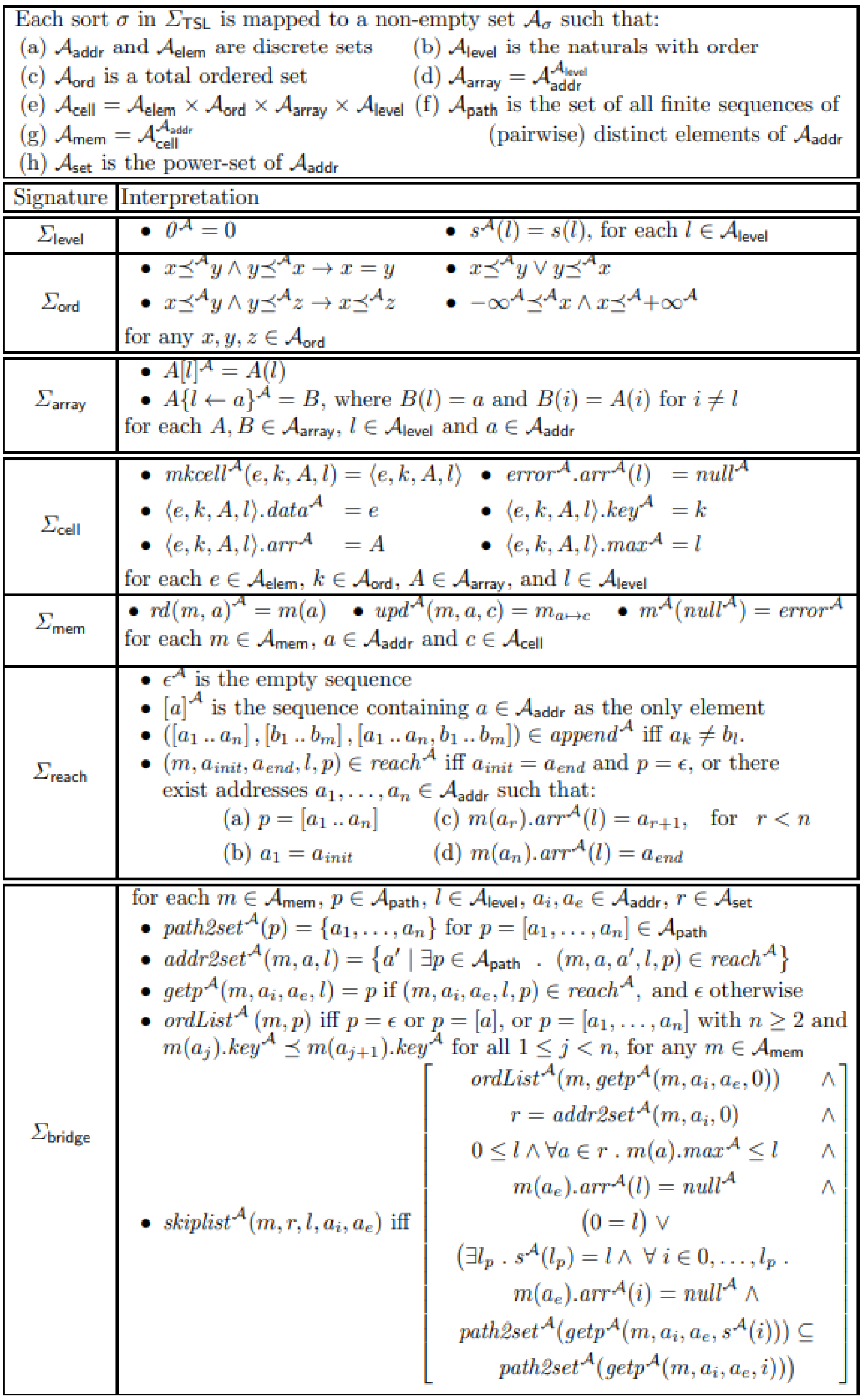
In [4], the authors complete a scalable verification framework for imperative implementation of skiplists of arbitrary levels, extending their previous works for skiplists with a fixed number of levels. Their work and our insights are reported in [3]. For an insight of the complexity, we reproduce here the table of their TSL-interpretation in Fig 2, namely their Theory for SkipLists in an imperative scenery. The details are not to be understood by the reader, only the mindset of the approach and the emerged complexity are relevant for our further comparisons.

3 Implementation

This section describes the proposed functional implementation for a skiplist. In order to make use of Stainless, the natural choice of programming language was Scala. As Stainless applies to only a subset of Scala, we crafted an implementation from scratch. There are many possible implementations for skiplist operations and basic properties. We wished to follow the KISS principle and use:

- the least dependencies, in particular only basic stainless-compatible Scala libraries were used, and no previous verification work on lists and other such constructs is leveraged
- the simplest "axioms", which is what we call the properties our objects need to verify to be considered valid Skiplists. Using only the most general properties adds the necessity to derive all other properties from them. For example, the property stating that each *level* i is a subset of the *level* $i-1$ leaves to be proven that $\forall\ 0 \leq j < i$ *level* i is a subset of *level* j.
- the most general properties on skiplist operations to be verified.

As will be shown further, some of these design choices may have led to unnecessary complexity.



Each sort σ in Σ_{TSL} is mapped to a non-empty set $\mathcal{A}_\sigma$ such that:

(a) $\mathcal{A}_{\mathsf{addr}}$ and $\mathcal{A}_{\mathsf{elem}}$ are discrete sets (b) $\mathcal{A}_{\mathsf{level}}$ is the naturals with order

(c) $\mathcal{A}_{\mathsf{ord}}$ is a total ordered set (d) $\mathcal{A}_{\mathsf{array}} = \mathcal{A}_{\mathsf{addr}}^{\mathcal{A}_{\mathsf{level}}}$

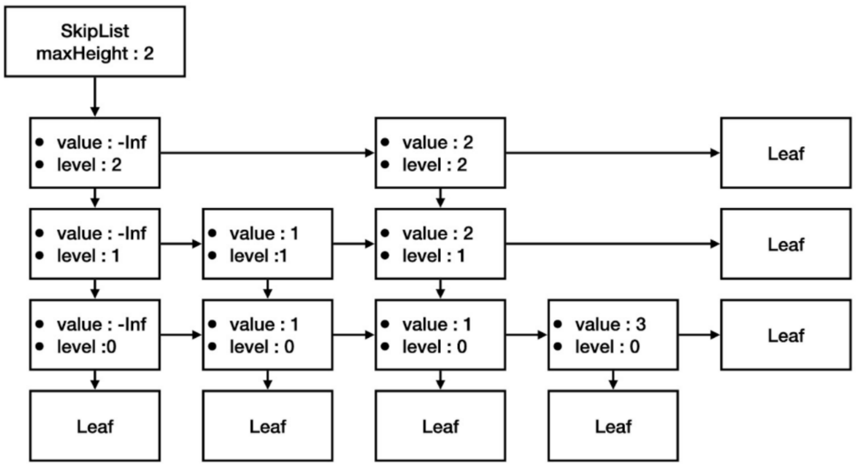
(e) $\mathcal{A}_{\mathsf{cell}} = \mathcal{A}_{\mathsf{elem}} \times \mathcal{A}_{\mathsf{ord}} \times \mathcal{A}_{\mathsf{array}} \times \mathcal{A}_{\mathsf{level}}$ (f) $\mathcal{A}_{\mathsf{path}}$ is the set of all finite sequences of (pairwise) distinct elements of $\mathcal{A}_{\mathsf{addr}}$

(g) $\mathcal{A}_{\mathsf{mem}} = \mathcal{A}_{\mathsf{cell}}^{\mathcal{A}_{\mathsf{addr}}}$

(h) $\mathcal{A}_{\mathsf{set}}$ is the power-set of $\mathcal{A}_{\mathsf{addr}}$

Signature	Interpretation
Σ_{level}	• $0^{\mathcal{A}} = 0$ • $s^{\mathcal{A}}(l) = s(l)$, for each $l \in \mathcal{A}_{\mathsf{level}}$
Σ_{ord}	• $x \preceq^{\mathcal{A}} y \wedge y \preceq^{\mathcal{A}} x \to x = y$ • $x \preceq^{\mathcal{A}} y \vee y \preceq^{\mathcal{A}} x$ • $x \preceq^{\mathcal{A}} y \wedge y \preceq^{\mathcal{A}} z \to x \preceq^{\mathcal{A}} z$ • $-\infty^{\mathcal{A}} \preceq^{\mathcal{A}} x \wedge x \preceq^{\mathcal{A}} +\infty^{\mathcal{A}}$ for any $x, y, z \in \mathcal{A}_{\mathsf{ord}}$
Σ_{array}	• $A[l]^{\mathcal{A}} = A(l)$ • $A\{l \leftarrow a\}^{\mathcal{A}} = B$, where $B(l) = a$ and $B(i) = A(i)$ for $i \neq l$ for each $A, B \in \mathcal{A}_{\mathsf{array}}$, $l \in \mathcal{A}_{\mathsf{level}}$ and $a \in \mathcal{A}_{\mathsf{addr}}$
Σ_{cell}	• $mkcell^{\mathcal{A}}(e,k,A,l) = \langle e,k,A,l \rangle$ • $error^{\mathcal{A}}.arr^{\mathcal{A}}(l) = null^{\mathcal{A}}$ • $\langle e,k,A,l \rangle.data^{\mathcal{A}} = e$ • $\langle e,k,A,l \rangle.key^{\mathcal{A}} = k$ • $\langle e,k,A,l \rangle.arr^{\mathcal{A}} = A$ • $\langle e,k,A,l \rangle.max^{\mathcal{A}} = l$ for each $e \in \mathcal{A}_{\mathsf{elem}}$, $k \in \mathcal{A}_{\mathsf{ord}}$, $A \in \mathcal{A}_{\mathsf{array}}$, and $l \in \mathcal{A}_{\mathsf{level}}$
Σ_{mem}	• $rd(m,a)^{\mathcal{A}} = m(a)$ • $upd^{\mathcal{A}}(m,a,c) = m_{a \mapsto c}$ • $m^{\mathcal{A}}(null^{\mathcal{A}}) = error^{\mathcal{A}}$ for each $m \in \mathcal{A}_{\mathsf{mem}}$, $a \in \mathcal{A}_{\mathsf{addr}}$ and $c \in \mathcal{A}_{\mathsf{cell}}$
Σ_{reach}	• $\epsilon^{\mathcal{A}}$ is the empty sequence • $[a]^{\mathcal{A}}$ is the sequence containing $a \in \mathcal{A}_{\mathsf{addr}}$ as the only element • $([a_1 \,..\, a_n], [b_1 \,..\, b_m], [a_1 \,..\, a_n, b_1 \,..\, b_m]) \in append^{\mathcal{A}}$ iff $a_k \neq b_l$. • $(m, a_{init}, a_{end}, l, p) \in reach^{\mathcal{A}}$ iff $a_{init} = a_{end}$ and $p = \epsilon$, or there exist addresses $a_1, \ldots, a_n \in \mathcal{A}_{\mathsf{addr}}$ such that: (a) $p = [a_1 \,..\, a_n]$ (c) $m(a_r).arr^{\mathcal{A}}(l) = a_{r+1}$, for $r < n$ (b) $a_1 = a_{init}$ (d) $m(a_n).arr^{\mathcal{A}}(l) = a_{end}$
Σ_{bridge}	for each $m \in \mathcal{A}_{\mathsf{mem}}$, $p \in \mathcal{A}_{\mathsf{path}}$, $l \in \mathcal{A}_{\mathsf{level}}$, $a_i, a_e \in \mathcal{A}_{\mathsf{addr}}$, $r \in \mathcal{A}_{\mathsf{set}}$ • $path2set^{\mathcal{A}}(p) = \{a_1, \ldots, a_n\}$ for $p = [a_1, \ldots, a_n] \in \mathcal{A}_{\mathsf{path}}$ • $addr2set^{\mathcal{A}}(m,a,l) = \left\{a' \mid \exists p \in \mathcal{A}_{\mathsf{path}} \;.\; (m,a,a',l,p) \in reach^{\mathcal{A}}\right\}$ • $getp^{\mathcal{A}}(m,a_i,a_e,l) = p$ if $(m,a_i,a_e,l,p) \in reach^{\mathcal{A}}$, and ϵ otherwise • $ordList^{\mathcal{A}}(m,p)$ iff $p = \epsilon$ or $p = [a]$, or $p = [a_1, \ldots, a_n]$ with $n \geq 2$ and $m(a_j).key^{\mathcal{A}} \preceq m(a_{j+1}).key^{\mathcal{A}}$ for all $1 \leq j < n$, for any $m \in \mathcal{A}_{\mathsf{mem}}$ • $skiplist^{\mathcal{A}}(m,r,l,a_i,a_e)$ iff $\left[\begin{array}{rl} ordList^{\mathcal{A}}(m, getp^{\mathcal{A}}(m,a_i,a_e,0)) & \wedge \\ r = addr2set^{\mathcal{A}}(m,a_i,0) & \wedge \\ 0 \leq l \wedge \forall a \in r \,.\, m(a).max^{\mathcal{A}} \leq l & \wedge \\ m(a_e).arr^{\mathcal{A}}(l) = null^{\mathcal{A}} & \wedge \\ (0 = l) \vee & \\ (\exists l_p \,.\, s^{\mathcal{A}}(l_p) = l \wedge \forall\, i \in 0, \ldots, l_p \,. & \\ m(a_e).arr^{\mathcal{A}}(i) = null^{\mathcal{A}} \wedge & \\ path2set^{\mathcal{A}}(getp^{\mathcal{A}}(m,a_i,a_e,s^{\mathcal{A}}(i))) \subseteq & \\ path2set^{\mathcal{A}}(getp^{\mathcal{A}}(m,a_i,a_e,i))) & \end{array}\right]$

Figure 2: Characterization of TSL-interpretation from [4]



H

Figure 3: Correct skiplist construct of height 2 for the set {1, 2, 3}

3.1 Objects

3.1.1 Nodes

Our skiplists are composed of `Node` objects, that can be either `SkipNode`s or `Leaf`s. `SkipNode` objects contain their value, and their relationship information to the rest of the structure. This information can be contained by the two progressive links to the `Node` downwards, which contains the same value at the lower level or is a Leaf, and the `Node` to the right, which contains the next value at the same level or is a Leaf. To distinguish between nodes at different levels, `SkipNode`s also maintain their height. The corresponding Scala code is shown below:

```scala
abstract class Node
case class SkipNode(value: Int,
                    down: Node,
                    right: Node,
                    height: BigInt)
                    extends Node
case object Leaf extends Node
```

3.1.2 SkipList

From `Node` objects, a complete skiplist can be contained as a whole from the top-left-most node of the list, which we call the head, along with the current complete height of the list.

```scala
class SkipList(head: Node, maxHeight: BigInt)
```

At this point, our skiplists and node objects can be combined into anything that is not a correct skiplist. For example, a `Node` of height 3 can point rightwards to a `Node` of height 9, which is not desired. To ensure our objects are correctly constructed skiplists, we defined the following structural properties:

- the head element is the special $-inf$ element
- the node heights are positive and strictly decrease downwards
- the node values increase rightwards
- each level is a subset of the underneath level recursively

```
def isASkipList(sl: SkipList): Boolean = {
    headIsMinInt(sl) &&
    hasNonNegativeHeight(sl.head) &&
    heightDecreasesDown(sl.head) &&
    increasesToTheRight(sl.head) &&
    levelsAxiom(sl.head)
}
```

Figure 3 illustrates a skiplist object defined as above.

3.2 Search

From skiplist properties, the search method must have logarithmic complexity for "well-formed" skiplists. We do not delve in the details of the structure as it does not impact implementation or verification much. A recrusive traversal search, akin to searching a balanced tree is satisfactory and described in the following pseudo-code fragment:

```
def search(value: int, sl: SkipList){ return search(v, sl.head) }

def search(value: int, n: Node){
    if(n is Leaf) return Not Found
    else if(n.value == value) return n
    else if(n.value < value) {
        if(n.right.value < value) return search(v, n.right)
        else return search(v, n.down)
    }
}
```

3.3 Insert

There are many ways to insert into a skiplist. To this point, we have purposefully avoided a touchy subject and made use of deceptions: our `Nodes` don't contain actual pointers to other nodes, but the `Nodes` themselves! This means that if we modify or insert somewhere at value x and height y, all south-east nodes must be made part of this newly created node. This adds a lot of complexity to the inserting process. It also highly encourages a bottom-up approach to rebuild with already updated nodes, whereas an imperative implementation using links would prefer a top-down approach, as searching is fast in this direction. Hence was chosen a bottom-up, level per level approach which was seemingly easier to verify at the cost of operation complexity as inserting in a level is a simple task. Additionally, we wished to remain true to our KISS principle and have not investigated the use of pointers. The pseudo-code follows:

```
//returns the -inf element of level level
def leftmost(head: Node, level: int)

// create a SkipNode of value value at the right position without a down node
def insertRight(value: int, n: Node)

// recursively update downward links of n to newly created node of the lower level
def stitch(n: Node, lower: Node)

//recursive level per level insert from bottom to top
def insertUpwards(value: int, desiredHeight: int, originalHead: Node,
                  currentHeight: int,
                  levelLeftmost: Node, newLowerLeftmost: Node): {
    if(desiredHeight >= currentHeight){
        //insertRight in this level
        //stitch this level with the lowerLevel which was updated prior
        //recurse up
    }
    else if(desiredHeight + 1 == currentHeight) {
        // stitch and recurse up
    }
    else if(currentHeight < originalHead.height) {
        //recurse to the top
    }
    else return levelLeftmost //the new head with insert value
}

def insert(value: int, height: int, sl: SkipList){
    if(height > sl.maxHeight) sl = increaseHeightOfHead(height)
    bottomLeft = leftmost(sl.head, 0)
    return insertUpwards(value, height, sl.head, 0, bottomLeft, Leaf)
}
```

3.4 Remove

As discussed later, correctness and properties on the remove operation were not achieved in this project. We hence do not reproduce the pseudo-code and details but the operation can be made with similar constructs as for inserting.

3.5 Remarks

One might ask: Weren't skiplist operations supposed to be simple? Isn't the complexity and memory load here far from optimal, and sufficient to disqualify this implementation from usage? The backbone problem is not having pointers, and the headstrong authors did not wish to come back on their word of using no dependencies. This choice heavily impacts the both design and time complexity of the operations. Another bias towards more complexity is the goal for verification. As stated previously, because no time complexity is required for the insert operation as in [4], the downstream verification procedure was taken into account for the choices regarding various aspects of the implementation. Although this may look bad, what programmer can swear that they did not modify some function's structure after not being able to resolve testing errors? These similar modifications only apply earlier in the design, and in the hope for a better reward, as verification is much stronger then passing a set of tests.

4 Formal Verification

Stainless verifies termination properties of operations by default. It leaves to verify stability and correctness properties, which are respectively properties stating that the operations do not have undesired side-effects on the given skiplist, and that operations result in the correct modification or yield the correct result.

4.1 Properties

In the following, it is assumed that sl is a valid skiplist and k, a, b, are distinct integers.

4.1.1 Search properties

- after search(k, sl), sl is still a valid skiplist
- if(k is in sl) search(k, sl) returns k
- if(k is not in sl) search(k, sl) returns Not Found
- b is in sl $\iff$ after search(k, sl) b is still in sl

4.1.2 Insert properties

- after insert(k, h, sl), sl is still a valid skiplist
- after insert(k, h, sl), k is in the returned skiplist
- b is in sl $\iff$ after insert(k, h, sl) b is still in sl

4.1.3 Remove properties

Similar to insert.

4.2 Results

After a lot more effort than envisioned, properties for *search* and all but last properties for *insert* were proven. Nothing points to an impossibility to prove the remaining properties. The verification job validates with above 2000 Stainless assertions with timeout under three seconds in about three minutes on a 4 core 2.8 GHz intel processor with 16G or Ram. The complete implementation and verification code is available on github[2].

4.3 Complication or complexity?

It is important to distinguish between complexity and complication: which part of the hardness of our work was inherent, and which induced?
Along proving all our properties, a few example of lemmas about relationships we had to prove are:

- relationship to the `Node` right
- relationship rightwards
- relationship to the `Node` down
- relationship downwards
- relationship south-east-wards
- subset and superset relationships between levels

For many of these, an additional axiom could have removed the necessity for a proof. Also, as discussed previously, the absence of pointers increased non-trivially the complexity of the design of our operations. This impacted the verification procedure as well. Both are complications that ripple from the choice to follow, maybe too strictly, the KISS principle. Although, it is not clear what complexity remains after cutting on the above. The authors believe that the backbone of the work would remain similar given more axioms, as this is not where the complexity lied: the proofs would need similar lemmas as those above. Further, if there is no functional pointer library that has been formally verified, it would move the complexity of the task to the verification of functional pointer logic. This line of work has not been investigated into.

4.4 Method comparison

Although the approach is very different to [4], it is hard to assess if the work done here was easier or more pleasant. No good conclusion on workload can be taken from the single effort of 3 students and without inner knowledge about the difficulty to obtain the results presented in [4].
Yet, the previous argument for approachability stand: coding a proof next to its code is much less daunting -even though it might be deceiving- than writing formal expressions. Another remark we can safely conclude on is that compared to the relatively few and complex steps of the traditional method, using Stainless as a proof assistant modifies the structure of the workload to many (many) simple and self-contained steps. This also plays in favor of the approachability of the method. From this segmentation of the workload, workers more frequently benefit from achievements although partial, and the work can be shared efficiently.

5 Conclusion

The proposed implementation for functional skiplists in Scala is described, and design choices are discussed to reflect their relationship to the verification process. The verification process using Stainless, a formal verification framework for Scala code, is presented along with its shortcomings. Namely, properties of correctness and stability are verified for the search operation and partially for the insert operation, but verification is left unattempted for the remove operation. A reflection about the added complexity induced by the design choices hints at both inherent complexity and superfluous complication. Further, the use of Stainless as a tool for formal verification is compared to previous works' approach and shows that although no conclusion on the effect to workload can be taken, this method might be more accessible to programmers. All-in-all, the authors were surprised by the work required and regret not being able to complete the verification process.

References

[1] Stainless documentation. `https://epfl-lara.github.io/stainless/intro.html`.

[2] Tanguy Rocher, Hugo Lepeytre, and Paul Juillard. Functional skiplist verification using stainless. `https://github.com/PaulJuillard/CS550-skiplist`.

[3] Tanguy Rocher, Hugo Lepeytre, and Paul Juillard. Review: Formal verification of skiplists with arbitrary many levels. CS550 paper review.

[4] Alejandro Sánchez and César Sánchez. Formal verification of skiplists with arbitrary many levels. In Franck Cassez and Jean-François Raskin, editors, *Automated Technology for Verification and Analysis*, pages 314–329, Cham, 2014. Springer International Publishing.