Verifying a Realistic Mutable Hash Table
Case Study (Short Paper)

Samuel Chassot[0009−0000−9751−9252] ✉ and Viktor Kunčak[0000−0001−7044−9522]

EPFL, Switzerland
`samuel.chassot@epfl.ch`, `viktor.kuncak@epfl.ch`

Abstract. In this work, we verify, using the Stainless program verifier, the mutable `LongMap` from the Scala standard library, a hash table using open addressing within a single array. As an executable specification, we write an immutable map based on a list of tuples and verify it against the mathematical definition of a map. We then show that `LongMap`'s operations correspond to operations of this association list. To express the resizing of the hash table array, we introduce a new reference-swapping construct in Stainless. This allows us to apply the decorator design pattern without introducing aliasing. Our verification effort led us to find and fix a bug in the original implementation that manifests for large hash tables. Our performance analysis shows the verified version to be within a 1.5 factor of the original data structure.

Keywords: Formal verification · Hash table · LongMap · Scala.

1 Introduction

With the improvements in effectiveness and expanding user base of proof assistants such as Isabelle/HOL [22] and Coq [27], we are witnessing systematic verification of many purely functional data structures. The verification of these data structures is highly effective using those tools. In the Scala language ecosystem, such verification efforts were carried out using the Stainless verifier [13] and its predecessor Leon [19]. However, verification of mutable data structures remains more challenging. As an example for hash table validation on the JVM platform, a recent attempt [8] provided a proof with interactive steps and an incomplete proof based on bounded model checking for one function. We consider such efforts very valuable. At the same time, our verification led us to discover a bug that bounded model checking would have likely missed due to the large arrays required. This illustrates the limitations of bounded checks and the need for complete formal verification.

In this work, we verify a data structure from the Scala standard library: the mutable `LongMap[V]`, a hash table with keys of type `Long` and values of a generic type `V`, implemented with open addressing (with all data stored in the arrays). We verify it using Stainless, a verification framework for a subset of Scala. To our knowledge, this is the first verified mutable map in Scala and the first verified hash table with open addressing and non-linear probing. Our implementation closely follows the existing implementation of the Scala library [26], which was implemented with efficiency in mind and withstood the test of usability. This is the fastest hash table implementation we know of in the Scala ecosystem. Our experience helped us further assess the use of Stainless for imperative code, following recent verification of the QOI compression algorithm [5] and file system components [12]. Our paper includes the following contributions:

1. As the key result, the adaptation and full formal verification of the mutable `LongMap` of the Scala standard library [26] using Stainless [13, 20]; this hash table can serve as a basis for other verified project; our code and the SMT queries generated during verification are available on Zenodo [6];
2. A reference implementation of a map verified against the mathematical definition of a map and lemmas for reasoning about such maps. This map is realized as a sorted list of tuples. We use it as an executable specification for `LongMap` and find that it supports automated and inductive reasoning better than the built-in maps of Stainless;
3. Introduction into Stainless of an operator for swapping references, which increases the expressive power of Stainless while preserving non-aliasing, allowing us to implement the resizing and balancing of the hash table;
4. An evaluation of the performance of both `LongMap` implementations (original and verified) and the mutable `HashMap` of the Scala standard library (unverified), showing that the performance of the verified implementation remains competitive despite the changes introduced to simplify verification.

1.1 Related Work

Hash tables have been of interest in verification from the early days of the field. Guttag [11] explores the use of algebraic specifications for reasoning about hash tables, though without formal connection to executable implementations. A hash table is one of the case studies [17] in the Jahob verification system [18, 29]. The version in Jahob does not use open addressing but separate chaining with linked list buckets. Furthermore, that case study uses, as an unverified assumption, the fact that the hash function is pure (total, without side effects, terminating, and deterministic). The Eiffel2 project offers a collection of verified data containers, impressive by its diversity [23]. They implemented and verified a hash table implementation using chaining. These containers are, however, simpler in their implementations than what appears in Scala and Java standard libraries. We could not explore this collection in more depth because the tools used are unavailable. De Boer et al. verified JDK's `IdentityHashTable`, based on open addressing and *linear* probing, in their case study [8]. The verification was done using KeY [1] and JJBMC [4], both accepting JML specification. They notably did not manage to provide a deductive proof for the `remove` method and one of its auxiliary methods, but instead used bounded model checking for a map of up to four elements. The KeY deductive proofs required interactive steps for the more complex methods, up to 1'655 for the `put` method. Hance et al. also proposed techniques to verify distributed systems interacting with an asynchronous environment, in particular file systems [14]. In this work, they developed and verified a hash table with open addressing and *linear* probing in Dafny. They implemented two versions of the hash table, one immutable and one mutable. This separates the functional correctness and correct heap manipulation proofs but requires implementing the hash table twice.

2 LongMap: from Scala Library to Stainless

A `LongMap[V]` (called `LongMap` in this work) is a data structure implementing a *map* behavior with keys of type `Long` (signed 64-bit machine integers) and values of generic type `V`. The mutable `LongMap` of the Scala standard library [26] is a hash table employing open addressing and non-linear probing.

We implement a subset of the original `LongMap` interface (outlined in Section 3). This subset corresponds to the functions implementing the map functionality (we omit functions specific to the Scala collections hierarchy). The `apply` function returns the value stored for a given key or a default value if absent. The `remove`, `update` (to add/update pairs), and `repack` (to resize/balance the map) functions return a Boolean value indicating the operation success.

The keys and values are stored in two arrays called `_keys` and `_values` respectively. Both are of size $N = 2^n$ for some $3 \leq n \leq 30$. The index of a given key is computed using a hash function. The corresponding value is stored in the second array at the same index as its key. We define `mask` $= N - 1$.

There are 2 special values in `_keys`: `0` and `Long.MinValue`. The value `0` indicates a free spot while `Long.MinValue` is a *tombstone* value, indicating that a key was removed at this spot.

We use open addressing with non-linear probing to resolve collisions. Following the original Scala implementation [7,26], the probing function is $\mathit{index}_{x+1} = (\mathit{index}_x + 2 * (x + 1) * x - 3)$ & mask, resulting in cubic index growth. Our verification is independent of the particular probing function but checks that the implementation is pure (i.e., deterministic, total, terminating, and without side effects), free of runtime errors, and returns an index within range.

All operations rely on two elementary ones: 1) looking for a key (`seekEntry`), and 2) looking for a key or an empty spot (`seekEntryOrOpen`). These two operations use non-linear probing, with the special values `0` and `Long.MinValue` in `_keys`. As an example, `update(k: Long, v: V)` starts out by computing `i = seekEntryOrOpen(k)`. If `k` is at index `i`, it writes `_values(i) = v`; if the function returns an open spot, it writes `_keys(i) = k` and `_values(i) = v`.

2.1 Adapting for verification

Next, we present the changes we made to the original code to comply with the supported subset of Scala, improve the SMT solver performance, make writing specifications easier, and simplify termination checking.

Tail recursion (to ease verification). We replace `while` loops with tail-recursive functions. Stainless can perform this transformation internally, but we have better control over specifications if we manually transform the source. For example, using a loop invariant makes having different pre- and post-conditions impossible. The Scala compiler transforms tail-recursive functions back to loops during compilation, so no performance is lost.

Loop counters (to prove termination). We introduce a counter and a condition that stops `while` loops (implemented as tail recursion) in `seekEntry` and `seekEntryOrOpen` after a fixed number of iterations. We need this counter as we do not know whether this probing function covers the space of all indices. Moreover, it allows the proof to be agnostic to the probing function. It has a negligible impact on performance, as shown in Section 4.

Data representation (for SMT performance). The original implementation uses the MSBs (Most Significant Bits) of the index returned by the seeking functions to indicate whether the index points to the key, a `0`, or a tombstone. We replace this with some ADT for better code readability and improved verification experience, as bitwise operations are often slow in SMT solvers.

Typing and initialization of arrays (to comply with subset). In the original implementation, the array storing values (`_values`) is an `Array[AnyRef]`, containing `null` by default, and using casts to store and access values. In our verified implementation, `_values` contains boxed values because Stainless does not support `null`s, and the `Array.fill` function (used to instantiate new arrays) does not support generically typed arrays. The boxing is implemented using case classes (i.e., ADTs).

Refactoring (to ease verification). We split the implementation into two classes, following the decorator design pattern (DP), as detailed in Section 3.1.

3 Specification and Verification

We first implement `ListLongMap`, an immutable map backed by a strictly ordered list of pairs `(Long, V)`, and verify it against the mathematical specification of a map. It serves as the executable specification of the mutable `LongMap`. We thus specify the mutable `LongMap` as behaving as the corresponding `ListLongMap`. A ghost method `map()` (not executed at runtime) of `LongMap` returns an instance of `ListLongMap` with the same content and is used in contracts. For example, `update` is specified as follows: `old(this).map() + (k, v) == this.map()`. Figure 1 shows the `LongMap` interface and specification. Postconditions, expressed using `ensuring` calls, are lambda functions taking the return value as parameter (i.e., `res`). The method `valid` is the data structure representation invariant stating, among other things, that the inserted elements can be found when searching subsequently using the same probing function. Table 1 shows the lines of code for the program, specification, and proofs for both maps.

3.1 Decorator Design Pattern for Modular Proofs

Following the decorator DP, we split the `LongMap` implementation into two classes to better modularize the proof. First, `LongMapFixedSize` implements

```
def contains(key: Long): Boolean = { ...
} ensuring (res ⇒ valid && (res == map.contains(key)))
def apply(key: Long): V = { ...
} ensuring (res ⇒ valid &&
                        (if (contains(key)) res == map.get(key).get
                         else res == underlying.v.defaultEntry(key)))
def update(key: Long, v: V): Boolean = { ...
} ensuring (res ⇒ valid &&
    (if (res) map.contains(key) && (map == old(this).map + (key, v))
     else map == old(this).map))
def remove(key: Long): Boolean = { ...
} ensuring (res ⇒ valid && (if (res) map == old(this).map − key
                                     else map == old(this).map))
def repack(): Boolean = { ... } ensuring (res ⇒ !res || map == old(this).map)
```

Fig. 1. Mutable LongMap interface and specification (note that we omit preconditions in this figure which are only checks of the invariant (*valid*), if any).

the `LongMap` specification depicted in Figure 1 without resizing (with arrays of a given fixed length). Then, we implement the `LongMap` class as a decorator of `LongMapFixedSize`. It implements the same interface and adds the resizing operation (`repack` function). Being a decorator, this class forwards all operations to an underlying instance of `LongMapFixedSize` except for the `repack` function. A key observation about the original implementation of `repack` is that it works very similarly to the `update` function to insert all pairs. Only some checks are omitted because the array is assumed to be fresh (containing no tombstone values and, initially, no keys). This observation allows us to use `update` to implement `repack` without significantly impacting the performance, while simplifying the proof.

3.2 Swap Operation for More Expressive Unique Reference

As discussed in Section 3.1, the `LongMap` class relies on an underlying instance of the `LongMapFixedSize` class. The underlying instance must be replaced by a new one during calls to `repack`. The repacking process first computes the new array size, then creates a new instance of `LongMapFixedSize` with this size, inserts all pairs, and finally replaces the current underlying instance with this new one. Aliasing appears during this replacement, yet Stainless disallows it. We can, however, observe that there is no need for aliasing because the reference to the newly created instance is not accessed after the replacement. We thus introduced a *swap* operation [15] into the Stainless verifier. In addition to array element swap [12], Stainless now offers a `Cell` class that encapsulates a mutable variable and offers a `swap` operation to swap the content of two cells. This construct enlarges the expressiveness of Stainless without the need for aliasing and enables the implementation of a resizable data structure on top of a fixed-size one.

Class	Program LOC	Proof+spec. LOC	Total LOC
ListLongMap	156	678	834
MutableLongMap	409	7'358	7'767

Table 1. Lines of code for program, as well as specification and proof. We use many ghost functions to express induction proofs. When a function has many arguments, we typically typeset each argument on a separate line, contributing to line counts.

3.3 Finding and Confirming a Bug in The Original Implementation

During the verification, we discovered that the `repack` function does not satisfy the specification stated in its documentation. The documentation states that the map can accommodate up to 2^{30} values (preferably not more than 2^{29}) [25]. However, a number of keys greater than or equal to 2^{28} makes `repack` loop forever. The bug arises in the computation of the new `mask` and is an integer overflow: a mask candidate is reduced while it is `> _size * 8` (where `_size` is the number of keys stored in the array). However, if `_size * 8` overflows, i.e., `_size` $\geq 2^{28}$, the mask candidate is reduced below `_size`. The new array then cannot accommodate all the keys. We fix the bug by modifying the loop condition and then prove that the function always returns a large enough and valid mask. Despite the small scope hypothesis [16] and claims in [8], we do not expect that bounded model checking would have discovered this bug, given that it occurs only after inserting so many key-value pairs.

3.4 SMT queries

Stainless generates verification conditions (VCs) which are solved by Inox [28] using SMT solvers (here, CVC4 [3], cvc5 [2], and Z3 [9]) and incremental function unrolling. So, a sequence of calls to SMT solvers happens for each VC and solvers run in parallel in a race. Generated SMT queries [6] use algebraic datatypes and sets. They do not contain *set-logic* directive. Only the query corresponding to the winning solver is recorded for each VC, as the others might be incomplete. Stainless uses generalized arrays [21] with non-constant default values to encode generic arrays among other things. This feature is unavailable on CVC4 and not implemented in Stainless for cvc5. Hence, VCs using it can be solved only by Z3. This partially explains why Z3 is solving most queries (Figure 2 (right)).

4 Evaluation

We run the benchmarks on an Ubuntu 20.04.6 LTS server with an Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz, 64GB RAM.

Verification takes around 400 seconds when running from scratch (around 100 seconds when re-running with a populated verification-condition cache [10]). Figure 2 shows the VCs solving time (with cache completely disabled). Most VCs are solved quickly, with a mean and a median of around 0.16 and 0.1 seconds,

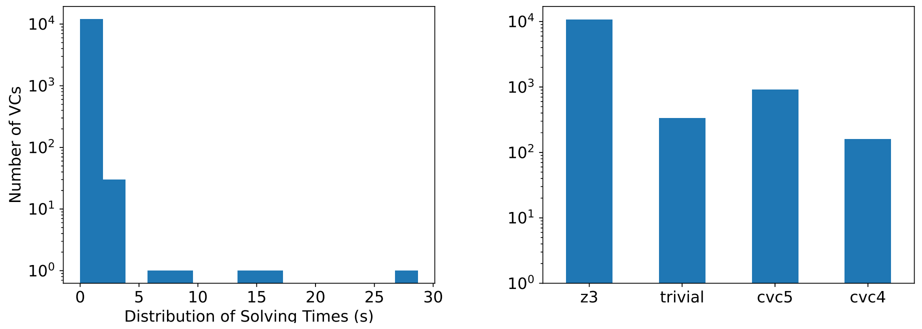

Fig. 2. Left: VC solving time distribution with Stainless cache disabled. Right: number of queries solved by each solvers. Both use a logarithmic scale.

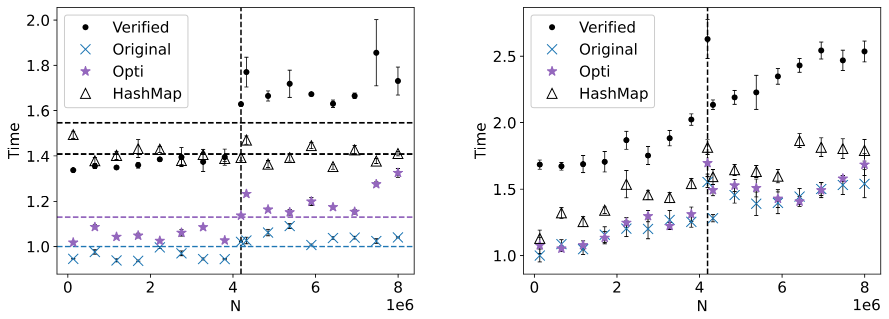

Fig. 3. Lookup of N keys in a map prepopulated with 2^{22} pairs (left) (time normalized per operation) and insertion of 2^{22} pairs (initial capacity of 16) followed by lookup of N keys (right). Horizontal lines show the average. The black vertical lines show 2^{22}. The error bars show the 95% CI. The time on the y-axis is normalized with respect to the first data point of the original map.

respectively. The cumulative solving time is 1'937 seconds. Only 3 VCs need more than 10 seconds in Stainless, with one VC capping at 29 seconds out of 12'122 VCs. When calling the solver directly on the generated SMT-LIB files, the cumulative solving time falls to 407 seconds. This likely shows the overhead of the unrolling in Inox [28], which is especially visible for fast VCs. Figure 2 also shows the distribution of VCs solved by each solver.

We compare the performance of our verified implementation to the original [26], the general `HashMap` of the Scala standard library [24], and an optimized version (denoted `Opti`) that changes the verified implementation to use `Array[AnyRef]` for `_values`. We use `Long` as the type of stored values. We consider three scenarios: looking up keys in a pre-populated map, populating the map first, then looking up keys, and populating the map with all pairs, removing half of the keys, and inserting all pairs again before looking up keys. Results are in Figure 3 and Figure 4. Our verified `LongMap` is around 1.5× slower than the

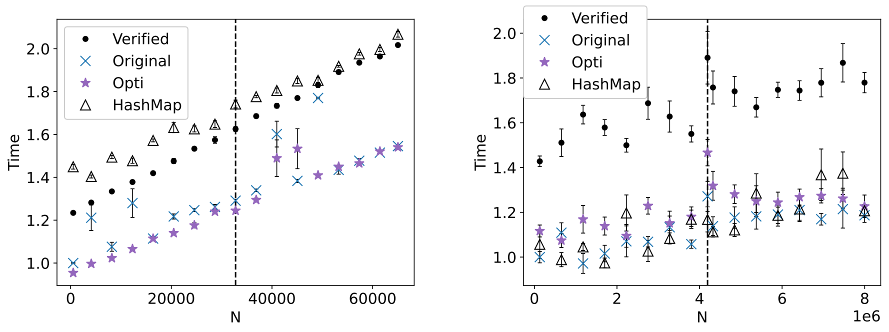

Fig. 4. Total time to lookup N keys and: (**left**) insert 2^{15} pairs with initial capacity 2^{17}, or (**right**) insert 2^{22} pairs, remove 2^{21}, and insert 2^{22} again, with initial capacity of 16. The black vertical line shows 2^{15} (left) and 2^{22} (right). The error bars show the 95% CI. The time on the y-axis is normalized with respect to the original map.

original implementation for lookups only, see Figure 3 (left). The performance gap is similar when taking the population process into account (Figure 3 (right)). We argue that this is acceptable. Indeed, the `LongMap` is the fastest map we know in the Scala ecosystem. As shown by Figure 4, the performance of our verified implementation is comparable to the Scala `HashMap` (better in some scenarios).

Consequences of Adapting for Verifiability. To understand the impact of pointer indirection in the `_values` array (Section 2.1), we modified our verified implementation to use `Array[AnyRef]` like the original (abandoning the proof). The results are shown as `Opti` in the figures, with performance close to the original one, indicating that this indirection was indeed responsible for the overhead. Similarly, in our version, creating `_values` and `_keys` arrays relies on `Array.fill`, which writes all values and is slower than constructing an array of `nulls` in the original implementation. Therefore, the verified `repack` operation is slower than the original, see Figure 4 (right). As shown by Figure 4 (left), without resizing, the performance is similar to `HashMap`, suggesting the impact of `Array.fill`. Calls to `repack` are infrequent, so this performance loss is limited. Finally, as witnessed by the `Opti` implementation performance being close to the original, there is limited performance impact of the way seek functions pass information to the caller, and of counter checks for loop termination (Section 2.1).

5 Conclusion

We verified `LongMap` from the Scala standard library, a mutable hash table with `Long` keys, employing open addressing and non-linear probing. This led us to identify and fix a bug in the original library implementation. The performance evaluation of our verified implementation against the original shows a slowdown of around 1.5. The changes we needed to perform for verifiability point to directions for further improving verification support for efficient Scala constructs.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification – The KeY Book, Lecture Notes in Computer Science, vol. 10001. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-49812-6, http://link.springer.com/10.1007/978-3-319-49812-6
2. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength smt solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 415–442. Springer International Publishing, Cham (2022)
3. Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 171–177. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_14, https://doi.org/10.1007/978-3-642-22110-1_14
4. Beckert, B., Kirsten, M., Klamroth, J., Ulbrich, M.: Modular Verification of JML Contracts Using Bounded Model Checking. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles, vol. 12476, pp. 60–80. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-61362-4_4, http://link.springer.com/10.1007/978-3-030-61362-4_4, series Title: Lecture Notes in Computer Science
5. Bucev, M., Kunčak, V.: Formally verified Quite OK Image format. In: Formal Methods in Computer-Aided Design (FMCAD) (2022)
6. Chassot, S., Kunčak, V.: Verifying a Realistic Mutable Hash Table Case Study (Short Paper) (Artifact) (Apr 2024). https://doi.org/10.5281/zenodo.11079220, https://doi.org/10.5281/zenodo.11079220
7. Commit: New mutable hash map with long keys, https://github.com/scala/scala/commit/05aedd936e7e7bcf0fa2443abd58b39732f173a9
8. De Boer, M., De Gouw, S., Klamroth, J., Jung, C., Ulbrich, M., Weigl, A.: Formal Specification and Verification of JDK's Identity Hash Map Implementation. Formal Aspects of Computing **35**(3), 18:1–18:26 (Sep 2023). https://doi.org/10.1145/3594729, https://dl.acm.org/doi/10.1145/3594729
9. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. p. 337–340. TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008), https://doi.org/10.1007/978-3-540-78800-3_24
10. Guilloud, S., Bucev, M., Milovančević, D., Kunčak, V.: Formula normalizations in verification. In: Computer-Aided Verification (CAV) (2023)
11. Guttag, J.V.: Abstract data type and the development of data structures. Commun. ACM **20**(6), 396–404 (1977). https://doi.org/10.1145/359605.359618
12. Hamza, J., Felix, S., Kunčak, V., Nussbaumer, I., Schramka, F.: From verified Scala to STIX file system embedded code using Stainless. In: NASA Formal Methods (NFM). p. 18 (2022), http://infoscience.epfl.ch/record/292424

13. Hamza, J., Voirol, N., Kunčak, V.: System fr: formalized foundations for the stainless verifier. Proc. ACM Program. Lang. **3**(OOPSLA) (oct 2019). https://doi.org/10.1145/3360592, https://doi.org/10.1145/3360592
14. Hance, T., Lattuada, A., Hawblitzel, C., Howell, J., Johnson, R., Parno, B.: Storage Systems are Distributed Systems (So Verify Them That Way!). In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2020)
15. Harms, D.E., Weide, B.W.: Copying and swapping: Influences on the design of reusable software components. IEEE Trans. Software Eng. **17**(5), 424–435 (1991). https://doi.org/10.1109/32.90445
16. Jackson, D., Damon, C.A.: Elements of style: analyzing a software design feature with a counterexample detector. ACM SIGSOFT Software Engineering Notes **21**(3), 239–249 (May 1996). https://doi.org/10.1145/226295.226322, https://dl.acm.org/doi/10.1145/226295.226322
17. Jahob Hashtables Codebase, https://github.com/epfl-lara/jahob/tree/master/examples/containers/hashtable
18. Kuncak, V.: Modular Data Structure Verification. Ph.D. thesis, EECS Department, Massachusetts Institute of Technology (February 2007), http://hdl.handle.net/1721.1/38533
19. Madhavan, R., Kulal, S., Kuncak, V.: Contract-based resource verification for higher-order functions with memoization. ACM SIGPLAN Notices **52**(1), 330–343 (Jan 2017). https://doi.org/10.1145/3093333.3009874, https://dl.acm.org/doi/10.1145/3093333.3009874
20. Milovančević, D., Kunčak, V.: Proving and disproving equivalence of functional programming assignments. In: ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI) (2023)
21. de Moura, L.M., Bjørner, N.S.: Generalized, efficient array decision procedures. In: Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA. pp. 45–52. IEEE (2009). https://doi.org/10.1109/FMCAD.2009.5351142
22. Nipkow, T., Wenzel, M., Paulson, L.C., Goos, G., Hartmanis, J., Van Leeuwen, J. (eds.): Isabelle/HOL, Lecture Notes in Computer Science, vol. 2283. Springer, Berlin, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9, http://link.springer.com/10.1007/3-540-45949-9
23. Polikarpova, N., Tschannen, J., Furia, C.A.: A fully verified container library. Formal Aspects of Computing **30**(5), 495–523 (Sep 2018). https://doi.org/10.1007/s00165-017-0435-1, https://doi.org/10.1007/s00165-017-0435-1
24. HashMap Scala Standard Library, https://scala-lang.org/api/3.3.1/scala/collection/mutable/HashMap.html
25. LongMap Specification, https://github.com/scala/scala/blob/263e5bd60d9c3947d8d17b7d8769a4b94f6865c7/src/library/scala/collection/mutable/LongMap.scala#L36
26. LongMap Implementation - Standard Library, https://github.com/scala/scala/blob/948ed0b60466803f404d5f24a3afc0a89c06ffc1/src/library/scala/collection/mutable/LongMap.scala
27. Team, T.C.D.: The Coq Proof Assistant (Jun 2023). https://doi.org/10.5281/ZENODO.1003420, https://zenodo.org/record/1003420, language: en
28. Voirol, N.C.Y.: Verified functional programming p. 229 (2019). https://doi.org/https://doi.org/10.5075/epfl-thesis-9479
29. Zee, K., Kuncak, V., Rinard, M.: Full functional verification of linked data structures. In: ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI) (2008)