Verified Mutable Data Structures

Samuel Chassot
EPFL
samuel.chassot@epfl.ch

Supervisor
Viktor Kunčak
EPFL / LARA

1. Introduction

Malfunctions in software like airplane control systems or nuclear plant control systems can have catastrophic consequences. Formal verification is the only form of sofware testing that can guarantee the absence of bugs. Formally verified software gives a mathematical proof that the specification is correctly implemented and that no bugs would induce unwanted behaviour. This has a high development cost and having an entirely verified program takes time and effort. However, having verified components already has great benefits.

We implement in Scala and formally verify with Stainless[1] a hash map that can then be reused and act as a basis on which to rely. The implementation we propose is based on the `LongMap` of the Scala standard library[2] with some minor adaptations. This map is implemented with mutable arrays. We give the specification with respect to an implementation of a map based on a list of tuples, that we implement and formally verify as well.

Our main contributions are:

- the implementation and the verification of a `ListMap` with `Long` keys and generic type value.
- the adaptation and verification of a HashMap based on the implementation of the Scala standard library, that uses a mutable array.

The code is available on Github: `https://github.com/epfl-lara/bolts/tree/master/data-structures/maps`

2. Background - Hash maps

We present here the principle and basic implementations of hash maps (or hash tables).

Hash maps are specific implementations of maps, which are data-structures that organize the data in form of key to value pairs. Depending on the implementation, the key and the value can be of various types, not necessarily identical.

The basic interface of maps is:

- `update(key, value)`: inserts a new key-value mapping in the table (erasing a possible previous one involving the same key).
- `contains(key)`: returns a boolean indicating whether a mapping for this key exists in the map or not.
- `apply(key)`: returns the value of the mapping corresponding to the given key, or a default value (or `NIL` depending on the implementation).
- `remove(key)`: removes a mapping corresponding to that key (does nothing if no such mapping exists in the map).
- various states of the table: `size`, `isEmpty`, ...

Hash maps use a hash function to distribute keys over an array of memory. To compute the index of a key, we then compute:

[1]`https://github.com/epfl-lara/stainless`
[2]`https://github.com/scala/scala/blob/2.13.x/src/library/scala/collection/mutable/LongMap.scala`

```
index = f(key) mod N
```

where `N` is the length of the array and `f` is a hash function of the form:

```
f: K -> Int
```

The hash function is chosen to minimize the number of collisions. It is however impossible to avoid them completely. The array usually cannot contain the entire range of keys and even if it can, the probability of collision is still non-zero. There are then multiple ways of dealing with collisions and we present some of them.

2.1. Collisions resolution

2.1.1 Separate chaining - linked lists

One way to resolve the collisions problem is to use linked lists. The idea consists of using each slot of the array as the starting block of a linked list. Each block contains a key-value pair and potentially a pointer to the next block.

To update a pair `(k, v)` in the map, the process is the following:

1. Compute the index for `k` using the hash function.
2. If the pointed slot is free (no mapping exists for this key yet), store the pair in it.
3. If not, go through the linked list:
   (a) If one of the block contains `k`, replace the current value by `v`.
   (b) If not (no mapping exists for this key yet), allocate a new block, store `(k, v)` in it and add it to the end of the linked (by storing the pointer to the new block in the last block of the list).

To retrieve a value for a key `k`, the process is the following:

1. Compute the index for `k` using the hash function.
2. If the slot is empty, it means that no mapping exists for `k`.
3. If not, go through the linked list:
   (a) If one of the block contains `k`, return `v`.
   (b) If not, it means that no mapping exists for `k`.

This method is a simple way to deal with collisions. The main issues with this type of mechanisms is that it does not behave nicely with CPU caches and induces a lot of misses and it requires new memory allocations during execution. There exist other ways of implementing separate chaining that help. For example, we can decide to have X (e.g., 3) slots in the array to store the X first collisions for each hash without having to allocate new memory.

2.1.2 Open addressing

With open addressing, all key-value pairs are stored in the initial array (i.e., no new space is allocated during execution for non-variable length maps). The hash function gives an index for the pair in the array, if the pointed slot is free, the pair is stored in it. If the slot is occupied, the array is probed until a free slot is found (or not if the map is full). There exist multiple probing techniques, the simplest is the linear probing that uses a fixed size step (generally 1). The quadratic probing uses a step of which the length is given by the value of a quadratic polynomial and therefore changes at each step, becoming larger.

Let's define the *occupancy factor* of a map as following:

$$occupancy\ factor = m/n$$

where m is the number of pairs that the map contains at a given point of the execution and n is its capacity.

2.1.3 Discussion

Both of the techniques have advantages and drawbacks.

Separate chaining pros and cons:

+ the number of elements that can be stored in the table is not upper bounded by the initial size of the array.
+ the hashing function only needs to minimize the number of collisions (i.e., values that hash to the same array slot).
+ the implementation is easy and the performance is good for retrieving values (if there is few collisions).
− it needs to allocate new memory when a collision occurs (i.e., the system on which it runs must have a memory allocator).
− insertion is therefore slower.
− it does not behave nicely with CPU caches.

Open addressing pros and cons:

+ there is no need for new memory allocation which means it can be implemented on systems with no memory allocator.
+ the overhead while inserting a new key-value pair is reduced (for a sufficiently low occupancy factor).
+ uses less memory for small key and values types (not needing any pointer in each block).
− the number of pairs that can be stored in the array is determined by the size of the array.
− the performance is dropping dramatically when the occupation factor increases (above around 0.7 [2]) as the number of hops needed during the probing phase increases.
− if the occupancy factor is low, memory is wasted because the array is always allocated entirely.

Concerning probing techniques for open addressing, the quadratic probing tends to perform better than the linear one [1]. One interesting observation is that, if two index from 2 different probing sequences are equal $x_i = y_i$, it is less probable that $x_{i+1} = y_{i+1}$ with quadratic probing than with linear probing, for which it is always the case. Quadratic probing becomes slower as the occupancy factor increases [1]. Linear probing is easier to implement but also suffers drastically when the array becomes full [1].

3. Background - base implementation

We base our verified implementation on the Scala standard library `LongMap[V]`[3]. This is a `Map` with keys of type `Long` and generically typed values. This map implements open addressing with quadratic probing. The maximum capacity is 2^{30} pairs but the documentation states that the performance decreases after reaching 2^{29}, which corresponds to an occupancy factor of 0.5.

This implementation stores keys and values in 2 separate arrays of the same size called `_keys` and `_values`. The key and the corresponding value of a pair are stored at the same index in their respective arrays. An integer `mask` conditions the size of the arrays which is `mask + 1` and is used to perform a cheap modulo operation (through a bitwise *and*). Both arrays are initialized to all `0`'s.

The map takes as a parameter a function called `defaultEntry` that maps `Long` to `V` (i.e., the type of the values) that represents the default entry for every key.

The map keeps a field variable `_size` that represents the number of pairs *in the arrays*.

The map offers a `repack` operation that computes a new `mask` given the occupancy factor of the map and copies all the key-value pairs in newly allocated arrays. This is used when the occupancy factor goes above or below a given threshold. The map also keeps track of the number of vacant slots (i.e., slots freed by removing a key) and this is also taken into account to determine when to repack. Repacking can also be initiated by the user of the map to improve performance after a massive

[3]`https://github.com/scala/scala/blob/2.13.x/src/library/scala/collection/mutable/LongMap.scala`

filling phase and before a read intensive phase as specified by the documentation [4]. We do not detail the repacking process as we do not implement it in our verified version.

The quadratic probing uses the following (recursively defined) polynomial:

$$index_{x+1} = (index_x + 2 * (x + 1) * x - 3) \& mask$$

where `x` is the iteration number during the probing.

The hash function used by this implementation is the following:

```
def toIndex(k: Long): Int = {
  val h = ((k ^ (k >>> 32)) & 0xFFFFFFFFL).toInt
  val x = (h ^ (h >>> 16)) * 0x85EBCA6B
  (x ^ (x >>> 13)) & mask
}
```

It already returns a valid index (i.e., modulo the size of the array) by performing a bitwise `and` with the `mask` just before returning.

3.1. Retrieve process

We present how a key (different from `0` and `Long.MinValue`, we detail later how they are treated and why) is retrieved from the map. This procedure finds the index of a pair:

1. Compute $index_0 = toIndex(key)$.
2. Iteratively probe the array starting at $index_0$ using the quadratic probing polynomial:
3. For each iteration `x`:
   (a) Read the element in `_keys` at $index_x$:
      - If it is equal to the wanted key: return $index_x$.
      - If it is equal to 0: the key is not in the map, return and inform that the key is not found.
      - Else: continue probing.

This is performed by the following function:

```
def seekEntry(k: Long): Int = {
    var e = toIndex(k)
    var x = 0
    var q = 0L
    while ({ q = _keys(e); if (q==k) return e; q != 0}) {
        x += 1; e = (e + 2*(x+1)*x - 3) & mask
    }
    e | MissingBit
}
```

This implementation uses MSBs of the index to store information. As the maximum size of the arrays is 2^{30}, there is indeed at least 3 bits free at the MSBs of any valid index. Here, `MissingBit` is equal to `0x80000000` (i.e., bit 31 set) and it indicates that the key is not found in the array and that the function encounters a 0.

The `apply(key)` function simply calls `seekEntry(key)` and, depending on the MSBs of the returned index, returns the value in the `_values` array at this index if the key was found or `defaultValue(key)` otherwise.

[4] https://github.com/scala/scala/blob/e7def0744253c00c5cba5e020c9187e83f9b6dc8/src/library/scala/collection/mutable/LongMap.scala#L271

3.2. Insertion process

We now present the process for inserting key-value pairs (again for keys different from `0` and `Long.MinValue`). To insert a key, we need to either find its index (to update the value) or find a free space in which to insert it. Note that if a free space is returned, the key must not be elsewhere in the array, as it would otherwise create a bug because the same key would map to possibly two different values. The process is the following for a key `k`:

1. Compute $index_0 = toIndex(\mathtt{k})$.
2. Iteratively probe the array starting at $index_0$ using the quadratic probing polynomial:
3. For each iteration `x`:
   (a) Read the element in `_keys` at $index_x$:
   - If it is equal to `k`, return $index_x$.
   - If it is equal to `Long.MinValue`, it means that the slot is free (it can contain a new mapping). However, this flag means that a key stood in this slot and was removed. Therefore, there is possibly other more recent keys later in the array. It therefore needs to keep this index (as it is the best free slot to return) and continue probing as `k` can be found later in the array.
   - If it is equal to 0, it means that `k` is not in the map, so return the free slot index. This index is either the one stored if we encountered a `Long.MinValue` earlier, or $index_x$ if not.
   - Else, continue probing.

This process is implemented in the function `seekEntryOrOpen`:

```
private def seekEntryOrOpen(k: Long): Int = {
    var e = toIndex(k)
    var x = 0
    var q = 0L
    while ({ q = _keys(e); if (q==k) return e; q+q != 0}) {
      x += 1
      e = (e + 2*(x+1)*x - 3) & mask
    }
    if (q == 0) return e | MissingBit
    val o = e | MissVacant
    while ({ q = _keys(e); if (q==k) return e; q != 0}) {
      x += 1
      e = (e + 2*(x+1)*x - 3) & mask
    }
    o
  }
```

Again, MSBs of the index are used to indicate the nature of the returned index. `MissingBit` again indicates that the key is not in the map and that the returned index points to a 0 slot. `MissVacant` (which is equal to `0xC0000000`, i.e., the bits 31 and 32 set) indicates also that the key is not in the map but that the returned index points to a `Long.MinValue` slot. If all the MSBs are 0, it means the index points to the wanted key.

The `update(key, value)` function calls `seekEntryOrOpens(key)` and update the arrays according to the return value: it simply updates the `_values` array if the key was found or update `_keys` and `_values` arrays if a free slot was returned.

3.3. Special case: 0 and Long.MinValue

The `0` and `Long.MinValue` values are used in the `_keys` array to indicate different types of free slots. The mappings corresponding to those 2 values can then not be stored in the arrays. An integer field of the class, `extraKeys`, indicates if mappings exist for these keys with its 2 LSBs: the bit 0 is set if the key `0` is defined and the bit 1 is set if the key `Long.MinValue` is defined. The values of these mappings are stored in 2 other fields of the class (`zeroValue` and `minValue`).

The processes for inserting and deleting as well as retrieving are easy for these 2. Each function first checks if the key passed as parameter is `0` or `Long.MinValue` (using the fact that these are the only 2 `Long`s that satisfy `x == -x`). If it is the case, it uses bitwise operation to check whether the key is present in the map or not and to insert/delete it if needed. Updating the values are simply performed by changing the value of the corresponding field of the class.

3.4. Code for apply and update

Here is the code for the two methods `apply` and `update`:

```
override def apply(key: Long): V = {
    if (key == -key) {
      if ((((key>>>63).toInt+1) & extraKeys) == 0) defaultEntry(key)
      else if (key == 0) zeroValue.asInstanceOf[V]
      else minValue.asInstanceOf[V]
    }
    else {
      val i = seekEntry(key)
      if (i < 0) defaultEntry(key) else _values(i).asInstanceOf[V]
    }
  }

override def update(key: Long, value: V): Unit = {
    if (key == -key) {
      if (key == 0) {
        zeroValue = value.asInstanceOf[AnyRef]
        extraKeys |= 1
      }
      else {
        minValue = value.asInstanceOf[AnyRef]
        extraKeys |= 2
      }
    }
    else {
      val i = seekEntryOrOpen(key)
      if (i < 0) {
        val j = i & IndexMask
        _keys(j) = key
        _values(j) = value.asInstanceOf[AnyRef]
        _size += 1
        if ((i & VacantBit) != 0) _vacant -= 1
        else if (imbalanced) repack()
      }
      else {
        _keys(i) = key
        _values(i) = value.asInstanceOf[AnyRef]
      }
    }
  }
```

3.5. Other functions

The class implements other functions like `getOrElse`, `getOrElseNull`, `getOrElse`, `get`, `put`, ... We do not detail them here because their implementations are very similar to `apply` and `update` and we do not implement them in our verified version.

4. Verified MutableLongMap

In this section we present the adaptations that we do to the original code and the proof strategy.
We call our verified version `MutableLongMap`.

4.1. Choices

4.1.1 Interface

We decide to implement a subset of the functions provided by the original code. We argue that the basic ones that we provide are sufficient to implement all the others in a straight forward manner. Our `MutableLongMap` provides the following public interface:

```
def size: Int
def isEmpty: Boolean
def contains(key: Long): Boolean
def apply(key: Long): Long
def update(key: Long, v: Long): Boolean
def remove(key: Long): Boolean
```

4.1.2 Long type for values

We decide to have keys and values of type `Long`. It helps a lot during the proof process and our implementation can be adapted to store generically typed values relatively easily after the verification is done. We also argue that the type `Long` for values is useful in itself.

4.1.3 Fixed size map

The repack process in the original code is a complex one: it computes a new `mask` based on the current occupancy factor of the map and the number of vacant slots (i.e., freed ones), initializes two new arrays for `_keys` and `_values`. It then transfers all the key-value pairs in these new arrays and replaces the old arrays by those. Proving the preservation of an invariant during this process is difficult. The copy process of the key-value pairs is analog to the ”normal” insertion process for each pair, the only difference is that we know in advance that the key is not in the array when being inserted. Therefore, having a map with `mask` fixed when instantiating is sufficient to implement this behaviour. One can indeed implement this by applying the decorator design pattern: a new class has a fixed-length map as field, it redirects all operations to the underlying map. Once the underlying map is full (or other criteria based on occupancy factor), it instantiates a new one with a bigger/smaller mask and inserts all key-value pairs from the old one using the `update` function.

Moreover, a fixed-length map is sufficient in some applications where the number of pairs can be approximated in advance.

4.2. Invariant

To achieve proof of correctness of such structures we need an *invariant*: a boolean expression that stays true between each function call and encapsulates the state of the data-structure. It represents all valid states of the data-structure and will be part of all pre- and postconditions of functions that make changes to the map. We indeed want that every function call preserves the validity of the state.

We now detail the invariant of our MutableLongMap:

```
private def simpleValid: Boolean = {
    validMask(mask) &&
    _values.length == mask + 1 &&
    _keys.length == _values.length &&
    _size >= 0 &&
    _size <= mask + 1 &&
    size >= _size &&
    size ==  _size + (extraKeys + 1) / 2 &&
    extraKeys >= 0 &&
    extraKeys <= 3
}

private def valid: Boolean = {
    //class invariant
    simpleValid &&
    arrayCountValidKeysTailRec(_keys, 0, _keys.length) == _size &&
    arrayForallSeekEntryOrOpenFound(0)(_keys, mask) &&
    arrayNoDuplicates(_keys, 0)
}
```

The invariant itself is the valid boolean. It is split in two different functions so that the simpleValid part can be used in pre- and postconditions of functions that are part of the valid part.

The simpleValid part ensures basic conditions:

- mask is valid (i.e., $== 2^N - 1$ for some $0 \leq N \leq 30$).
- the length of the arrays _keys and _values is mask 1+.
- _size is valid.
- size (call to the function of the public interface, i.e., the real number of pairs including 0 and Long.MinValue) is valid.
- extraKeys is in a valid state i.e., only bits 0 and 1 can be set so between 0 and 3.

The valid part ensures simpleValid plus more complex conditions:

- arrayCountValidKeysTailRec is defined recursively on Array[Long] and computes the number of *valid keys* in this array from a starting index to the end of the array (a valid key is a Long different from 0 and Long.MinValue). The number of valid keys in the whole _keys array must always be equals to _size.
- arrayForallSeekEntryOrOpenFound is a recursive function defined on arrays of Long as follows:

```
def arrayForallSeekEntryOrOpenFound(i: Int)
            (implicit _keys: Array[Long], mask: Int): Boolean = {
    //require(...)

    if (i >= _keys.length) true
    else if (validKeyInArray(_keys(i))) {
        seekEntryOrOpenDecoupled(_keys(i))(_keys, mask) == Found(i) &&
        arrayForallSeekEntryOrOpenFound(i + 1)
    } else arrayForallSeekEntryOrOpenFound(i + 1)
}
```

It is `true` if and only if, for each valid key in the array from the index `i` to the end of the array, the function `seekEntryOrOpen` finds the correct index. It means that, if a key is in the array `_keys`, `seekEntryOrOpen` finds its index and, by contraposition, if `seekEntryOrOpen` returns a free slot's index for a given key, this key is not in the array. This is a crucial property to ensure that keys can be found either to read the value or update it.

- The last condition ensures that the array `_keys` does not contain any duplicated *valid* keys. `arrayNoDuplicates` is also defined recursively on an array starting at a given index as follows:

```
def arrayNoDuplicates(a: Array[Long],
                      from: Int,
                      acc: List[Long] = Nil[Long]()): Boolean = {
    //require(...)

    if (from >= a.length) true
    else if (validKeyInArray(a(from)) && acc.contains(a(from))) false
    else if (validKeyInArray(a(from)))
                  arrayNoDuplicates(a, from + 1, Cons(a(from), acc))
    else arrayNoDuplicates(a, from + 1, acc)
}
```

4.3. Adaptations

We try to minimize the changes to the original code but we have to make some for practical reasons. We present them in this section.

4.3.1 While loops to tail recursion

The Stainless framework only uses recursion. The compiler then automatically transform `while` loops in tail recursive function calls. We decide to change it manually in the code to make it closer to what Stainless uses and thus make reasoning easier. So `seekEntry` and `seekEntryOrOpen` call helper tail recursive functions instead of having `while` loops. The performance of both of these constructs is similar in our context so this change has no significant impact for the user.

4.3.2 Stop conditions for ”seek” functions

In the original code, the ”seek” functions (`seekEntry`, `seekEntryOrOpen` and alike) could loop indefinitely if the array `_keys` does not contain the wanted key nor `0` and `Long.MinValue`. There is surely an intuitive or empirical argument to justify that it would not happen in a normal execution (especially with the repack process). This is however obviously not enough for formal verification. We hence modify the `seekEntry`, `seekEntryOrOpen` functions to return an *undefined* state if they reach a maximum number of iterations (set arbitrarily to 2048) without loss of generality.

4.3.3 Return type of ”seek” functions

The original code of the ”seek” functions uses MSBs of the returned integer index to encode information about its nature. Bitwise operations make however the proofs using our framework complicated and slow. Bitwise operations are indeed difficult to translate into verification conditions and the resulting ones tend to be too large and complicated to be solved by a SAT solver. We then decide to change the implementation so that `seekEntry` and `seekEntryOrOpen` (the only 2 ”seek” functions we use) return an instance of a case class instead. We therefore design the following type:

```
abstract sealed class SeekEntryResult
  case class Found(index: Int) extends SeekEntryResult
  case class MissingZero(index: Int) extends SeekEntryResult
  case class MissingVacant(index: Int) extends SeekEntryResult
  case class Intermediate(undefined: Boolean, index: Int, x: Int)
                                            extends SeekEntryResult
  case class Undefined() extends SeekEntryResult
```

- `Found` is returned when the index points to the wanted key.
- `MissingZero` and `MissingVacant` indicates that the key was not found and that the index points to a `0` respectively a `Long.MinValue`.
- `Intermediate` is used by one helper function of `seekEntryOrOpen` as it needs to return its iterations count `x` along with the index.
- `Undefined` is returned when the maximum number of iteration is reached.

This change also makes the code cleaner and easier to read as pattern matching can be used in place of `if` blocks with bitwise conditions.

It induces a loss of performance as returning a case class instance requires a heap allocation when returning a simple integer does not. We however argue that the loss is minor and, as the frameworks evolve, the code can be refactored to use integer again with only little efforts.

4.3.4 seekEntry and seekEntryOrOpen merge

Recall that following the invariant, `seekEntryOrOpen` finds the right index for all valid keys in `_keys`. As the `apply` and `contains` functions use `seekEntry`, we need to prove an equivalence between `seekEntryOrOpen` and `seekEntry`. They both perform in a similar way but their implementation diverges sufficiently for the equivalence to be non-trivial.

Recall that

- `seekEntry` probes the array looking for either the wanted key or `0`. If `seekEntry` finds the key, it returns `Found(index)` and if it finds `0`, it returns `MissingZero`.
- `seekEntryOrOpen` first looks for the wanted key, `0` or `Long.MinValue`. Then, if it finds `Long.MinValue`, it keeps the index and continues probing as `seekEntry` would. If it then finds the key, it returns `Found(index)` just as `seekEntry`. If it then finds a `0`, it returns `MissingVacant(index)` where `index` is the index of the `Long.MinValue` found in the first part. If it does not encounter any `Long.MinValue`, it performs as `seekEntry`.

A thorough case analysis shows that the only case in which the return values of these 2 functions differ is when the probing process goes by a `Long.MinValue` and the key is not in the array. In that case, `seekEntry` would return `MissingZero(index0)` where `index0` is the index of the 0 found at the end and `seekEntryOrOpen` would return `MissingVacant(indexL)` where `indexL` in the index of the first `Long.MinValue` encountered during probing. The Figure 1 shows visually this case analysis.

As the index return by `seekEntry` is used only if the result is `Found`, the implementation of `seekEntry` can be replaced by `seekEntryOrOpen`'s one with only small changes to return `MissingZero` in place of `MissingVacant`. It returns a wrong index in the sense that it can point to a `Long.MinValue` instead of a `0` but it is not used so it causes no problem.

These two functions are essentially identical except that one ”stops” at the first `Long.MinValue` and returns that index if it finds a 0 later. Most importantly, the probing sequence is identical.

We therefore decide to replace `seekEntry` implementation by the one of `seekEntryOrOpen` with a minor change, making the proof of their equivalence essentially trivial without affecting the code in a substantial manner.

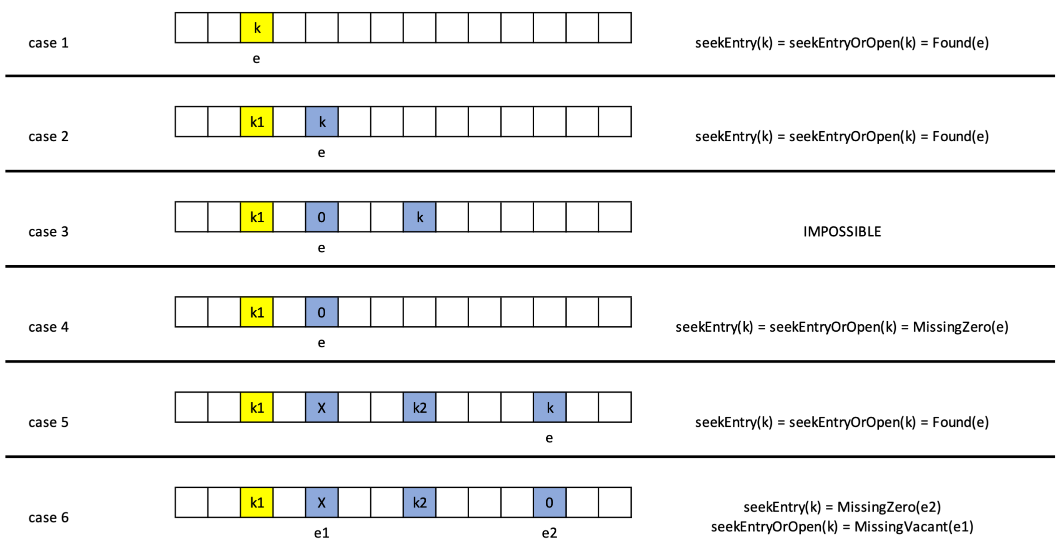
case 1
k
e
seekEntry(k) = seekEntryOrOpen(k) = Found(e)
case 2
k1
k
e
seekEntry(k) = seekEntryOrOpen(k) = Found(e)
case 3
k1
0
k
e
IMPOSSIBLE
case 4
k1
0
e
seekEntry(k) = seekEntryOrOpen(k) = MissingZero(e)
case 5
k1
X
k2
k
e
seekEntry(k) = seekEntryOrOpen(k) = Found(e)
case 6
k1
X
k2
0
e1
e2
seekEntry(k) = MissingZero(e2)
seekEntryOrOpen(k) = MissingVacant(e1)

Figure 1. Case analysis of `seekEntry` and `seekEntryOrOpen` behaviour for a key `k`.Yellow slot is at index `toIndex(k)`, coloured slots are the ones visited by the probing. `X` stands for `Long.MinValue`. Note that case 3 is impossible because it violates the invariant: `seekEntryOrOpen` would not find `k`.

Here is the code of `seekEntry` and `seekEntryOrOpen` after all the adaptations:

```
def seekEntry(k: Long)
            (implicit _keys: Array[Long], mask: Int): SeekEntryResult = {
    require(...)
    val intermediate =
        seekKeyOrZeroOrLongMinValue(0, toIndex(k, mask))(k, _keys, mask)
    intermediate match {
        case Intermediate(undefined, index, x) if (undefined) => Undefined()
        case Intermediate(undefined, index, x) if (!undefined) => {
            val q = _keys(index)
            if (q == k) Found(index)
            else if (q == 0) MissingZero(index)
            else {
                val res =
                    seekKeyOrZeroReturnVacant(x, index, index)(k, _keys, mask)
                res match {
                    case MissingVacant(index) => MissingZero(index)
                    case _                     => res
                }
            }
        }
    }
}
def seekEntryOrOpen(k: Long)
                (implicit _keys: Array[Long], mask: Int): SeekEntryResult = {
    require(...)
    val intermediate =
        seekKeyOrZeroOrLongMinValue(0, toIndex(k, mask))(k, _keys, mask)
    intermediate match {
        case Intermediate(undefined, index, x) if (undefined) => Undefined()
```

```
            case Intermediate(undefined, index, x) if (!undefined) => {
                val q = _keys(index)
                if (q == k) Found(index)
                else if (q == 0) MissingZero(index)
                else {
                    assert(_keys(index) == Long.MinValue)
                    assert(index >= 0 && index < mask + 1)
                    val res =
                        seekKeyOrZeroReturnVacant(x, index, index)(k, _keys, mask)
                    res
                }
            }
        }
    }
```

The two tail recursive functions `seekKeyOrZeroOrLongMinValue` and `seekKeyOrZeroReturnVacant` replace the `while` loops of the original code. We do not reproduce them here but they are accessible on Github[5].

4.3.5 Update behaviour

Following the fixed length of the map and the stop condition in `seekEntryOrOpen` that can now return `Undefined`, the behaviour of the `update` function has to be modified. It now returns a boolean indicating whether the update was performed or not. The update is not performed in the case when `seekEntryOrOpen` returns `Undefined` meaning that no free space is available (for that particular key at least).

4.3.6 External functions

In the original code, all functions are methods of the class. The mutability of the class' fields means that they are implicitly all passed as arguments to all methods (via `this`). This makes the formal reasoning more complicated and the verification conditions that Stainless produces bigger and thus more difficult to verify with a SAT solver. As a consequence, some trivial properties are not implicitly valid for Stainless. For example, the preservation of the `arrayNoDuplicate` invariant was non-trivial even for methods that obviously do not modify the array. It is however not trivial for Stainless that the array does not change if it is passed as argument.

For all these reasons, we move all functions that are not part of the public interface into a companion object. Each function then takes as arguments only the needed parameters. This makes the code less clear at first glance due to the high number of parameters but really helps the verification and the proofs.

4.4. ListMap

We need a reference implementation on which to base the specifications of our map. We then implement a verified `ListMap`. This is an implementation of a map based on a list of tuples. The first element of each tuple is the key and the second one is the corresponding value of a pair. We called this map `ListMapLongKey`[6].

As each key is unique in a map, we enforce the underlying list to be *strictly ordered* by the first element of each tuple. We then implement functions on `List[(Long, V)]` that add, remove, check presence, etc... while preserving this invariant. We then use those functions to implement the map's operations. The `ListMapLongKey` has generically typed values unlike `MutableLongMap`. It was straight forward to implement and we think it can be used later for other proofs.

[5] https://github.com/epfl-lara/bolts/blob/master/data-structures/maps/MutableLongMap.scala
[6] https://github.com/samuelchassot/bolts/blob/16470924fd3d22d1f38117b8871a21804f13243f/data-structures/maps/StrictlyOrderedLongListMap.scala

We then prove some lemmas about properties of the map that help to prove the main `MutableLongMap`. Here are the lemmas used in the proof of `MutableLongMap`:

```scala
def addStillContains[B](lm: ListMapLongKey[B], a: Long, b: B, a0: Long): Unit = {
    require(lm.contains(a0))
    //...
}.ensuring(_ => (lm + (a, b)).contains(a0))

def addApplyDifferent[B](lm: ListMapLongKey[B], a: Long, b: B, a0: Long): Unit = {
    require(lm.contains(a0) && a0 != a)
    //...
}.ensuring(_ => (lm + (a -> b))(a0) == lm(a0))

def addStillNotContains[B](lm: ListMapLongKey[B], a: Long, b: B, a0: Long): Unit = {
    require(!lm.contains(a0) && a != a0)
    //...
}.ensuring(_ => !(lm + (a, b)).contains(a0))

def addCommutativeForDiffKeys[B]
        (lm: ListMapLongKey[B], a1: Long, b1: B, a2: Long, b2: B): Unit = {
    require(a1 != a2)
    //...
}.ensuring(_ => lm + (a1, b1) + (a2, b2) == lm + (a2, b2) + (a1, b1))

def addSameAsAddTwiceSameKeyDiffValues[B]
        (lm: ListMapLongKey[B], a: Long, b1: B, b2: B): Unit = {
    //...
}.ensuring(_ => lm + (a, b2) == (lm + (a, b1) + (a, b2)))

def addRemoveCommutativeForDiffKeys[B]
        (lm: ListMapLongKey[B], a1: Long, b1: B, a2: Long): Unit = {
    require(a1 != a2)
    //...
}.ensuring(_ => lm + (a1, b1) - a2 == lm - a2 + (a1, b1))

def removeNotPresentStillSame[B](lm: ListMapLongKey[B], a: Long): Unit = {
    require(!lm.contains(a))
    //...
}.ensuring(_ => lm - a == lm)

def addThenRemoveForNewKeyIsSame[B](lm: ListMapLongKey[B], a1: Long, b1: B): Unit = {
    require(!lm.contains(a1))
    //...
}.ensuring(_ => lm + (a1, b1) - a1 == lm)

def emptyContainsNothing[B](k: Long): Unit = {
}.ensuring(_ => !ListMapLongKey.empty[B].contains(k))
```

`ListMapLongKey` provides a formally verified implementation of a map. The issue with it is the performance: it would not be usable in a production program. However, it provides a fairly simple implementation of the interface and we can use it as a reference for the specification of our `MutableLongMap`.

4.5. Specification of MutableLongMap

We now present the specification of `MutableLongMap`.

4.5.1 Equivalent ListMap

To use the ListMap in the specification, we first need to implement a function that returns an instance of `ListMapLongKey` that mirrors the current state of `MutableLongMap`. This is performed by the `getCurrentListMap` function. Here is the pseudo code of this function and its helper:

```
def getCurrentListMap(from: Int): ListMapLongKey[Long] = {
    if ((extraKeys & 1) != 0 && (extraKeys & 2) != 0) {
      // it means there is a mapping for the key 0 and the Long.MIN_VALUE
      (getCurrentListMapNoExtraKeys(from) + (0L, zeroValue)) + (Long.MinValue, minValue)
    } else if ((extraKeys & 1) != 0 && (extraKeys & 2) == 0) {
      // it means there is a mapping for the key 0
      getCurrentListMapNoExtraKeys(
        _keys,
        _values,
        mask,
        extraKeys,
        zeroValue,
        minValue,
        from
      ) + (0L, zeroValue)
    } else if ((extraKeys & 2) != 0 && (extraKeys & 1) == 0) {
      // it means there is a mapping for the key Long.MIN_VALUE
      getCurrentListMapNoExtraKeys(from) + (Long.MinValue, minValue)
    } else {
      getCurrentListMapNoExtraKeys(from)
    }
}
def getCurrentListMapNoExtraKeys(from: Int): ListMapLongKey[Long] = {
    if (from >= _keys.length) {
      ListMapLongKey.empty[Long]
    } else if (validKeyInArray(_keys(from))) {
      getCurrentListMapNoExtraKeys(from + 1) + (_keys(from), _values(from))
    } else {
      getCurrentListMapNoExtraKeys(from + 1)
    }
}
```

The implementation is straight forward: it goes recursively through the arrays, adding to the result the pair `(key, value)` if the key is a valid one. At the end, it adds the pairs for the keys `0` and `Long.MinValue` if defined.

We then prove 4 lemmas that ensure that the ListMap returned by this function and the state of the arrays are equivalent:

- `lemmaKeyInListMapIsInArray` proves that if a key is in the ListMap, then `_keys` contains that key.
- `lemmaValidKeyInArrayIsInListMap` proves that if the key at a given index in `_keys` is valid, then it is in the ListMap.
- `lemmaArrayContainsKeyThenInListMap` proves that if `_keys` contains the given key, then it is in the ListMap.
- `lemmaKeyInListMapThenSameValueInArray` proves that if a key is in the ListMap and at a given index in `_keys`, then the value in the ListMap and in `_values` is the same.

These 4 lemmas prove that the ListMap is indeed equivalent to the state of `MutableLongMap`. We can then use `getCurrentListMap(from: 0)` in the specifications of the public functions of `MutableLongMap`.

From now on, we will refer to the ListMap returned by `getCurrentListMap(from: 0)` as `map`:

```
def map = getCurrentListMap(0)
```

4.5.2 Specifications of public functions

We can now define the public functions' specifications using `map`. Basically, we prove that all `MutableLongMap`'s operations preserve the invariant and that they have the same result as the equivalent operations on the ListMap.

```
def contains(key: Long): Boolean = {
    require(valid)
    //...
}.ensuring(res => valid && (res == map.contains(key)))

def apply(key: Long): Long = {
    require(valid)
    //...
}.ensuring(res => valid &&
    (if (contains(key)) Some(res) == map.get(key)
        else res == defaultEntry(key)))

def update(key: Long, v: Long): Boolean = {
    require(valid)
    val oldMap =
        getCurrentListMap(_keys, _values, mask, extraKeys, zeroValue, minValue, 0)
    {
        //...
    }.ensuring(res => valid &&
        (if (res) map.contains(key) && (map == oldMap + (key, v))
         else map == oldMap))
}

def remove(key: Long): Boolean = {
    require(valid)
    val oldMap =
        getCurrentListMap(_keys, _values, mask, extraKeys, zeroValue, minValue, 0)
    {
        //...
    }.ensuring(res => valid && (if (res) map == oldMap - key else map == oldMap))
}
```

Proving these conditions is far from trivial and requires a lot of intermediate lemmas that we do not detail here.

4.6. Custom definitions

The verification of code that makes use of high order functions is really difficult and Stainless struggles to prove even trivial things. To counteract this issue, we have to redefine almost all functions on `List` and `Array`. We do not use `map`, `contains`, `flatMap`, ... We list here some of the functions that we have to define without high order function to make the proof possible.

```
def arrayCountValidKeys(a: Array[Long], from: Int, to: Int): Int
def arrayContainsKey(a: Array[Long], k: Long, from: Int): Boolean
def arrayScanForKey(a: Array[Long], k: Long, from: Int): Int
def arrayNoDuplicates(a: Array[Long],
                      from: Int,
                      acc: List[Long] = Nil[Long]()): Boolean
```

We prove some lemmas about those functions as well, that we do not detail here.

We think that it is interesting to note that, while doing formal verification, the beauty of functional programming must be at least partly set aside. It is almost mandatory to redefine a function for everything on sequences instead of using `map`, `flatMap`, `foldLeft`, and alike. This makes the code less beautiful but easier to reason about and more friendly to work with using framework like Stainless.

5. Conlusion

In this project, we implement and formally verify 2 different maps: one simple and based on a list for specification purposes and one based on a mutable array. The implementation we propose and verify for the mutable one is almost identical to the implementation of the `LongMap` in the Scala standard library. That means that our map can be used in a real production environment, especially at the performance level.

Our `MutableLongMap` can then be reused in any sort of programs, formally verified or not, as a basis on which to rely.

Having a collection of verified data-structures (and other programs) also helps to develop further the Stainless framework to showcase its capabilities and to have a collection of programs to verify during testing.

6. Future work

Our `MutableLongMap` can be refactored to allow generically typed values. It can also be extended to allow variable size.

We do not think that proving the repacking process of the original code and incorporate it in the implementation of the map would bring valuable benefits to the usability of the map. However, it is possible that doing it will help discover bugs in the original code, which would be beneficial to the Scala standard library.

7. Acknowledgements

I want to thank my supervisor, Viktor Kunčak, for his support and his help during this project. I also want to thank Jad Hamza for answering my questions and helping me.

References

[1] Saifullahi Aminu Bello1 Ahmed Mukhtar Liman, Abubakar Sulaiman Gezawa3 Abdurra'uf Garba, and Abubakar Ado. Comparative analysis of linear probing, quadratic probing and double hashing techniques for resolving collusion in a hash table.

[2] Dapeng Liu and Shaochun Xu. Comparison of hash table performance with open addressing and closed addressing: An empirical study. *International Journal of Networked and Distributed Computing*, 3:60–68, 2015.