UiT
THE ARCTIC
UNIVERSITY
OF NORWAY

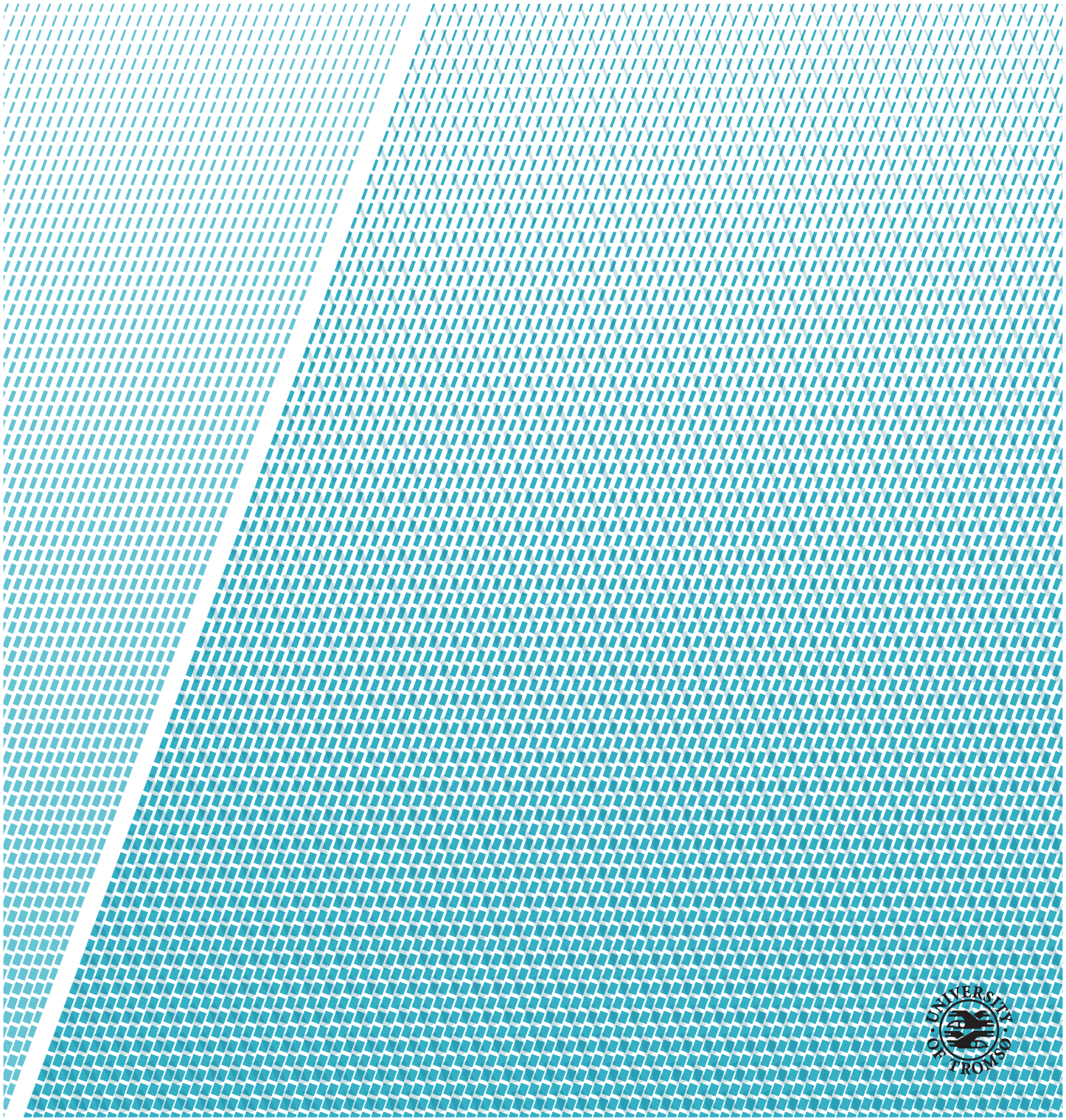
Faculty of Science and Technology
Department of Computer Science

Formal Verification of a Lock-free Split-order Hashmap

—

Åsmund Aqissiaq Arild Kløvstad

INF-2990 Bachelor's thesis in Informatics April 4, 2020



“You don’t need to understand everything at once. You understand one thing, then you pat yourself on the back, have a cup of coffee, and understand one more thing.”
–Nada Amin

Contents

List of Figures **v**

1 Introduction **1**
1.1 Thesis 2
1.2 Method 2
1.3 Scope? 3
1.4 Outline 3

2 Background **5**
2.1 Temporal Logics 6
2.2 TLA+ 10
2.3 Split-Ordered List Hashmap 10

3 TLA+ Specification **13**
3.1 Goal 13
3.2 Hashmap for implementation 14
3.3 Split-ordered List 16

4 Results **21**
4.1 Discussion 21

Appendix **24**

Split-order Specification **25**

List of Figures

2.1 Pnueli's axioms 8
2.2 Pnueli's inference rules 9
2.3 The layout of the split-ordered list 11
2.4 Insertion without bucket splitting 11
2.5 Expansion and bucket splitting 12

3.1 The hashmap specification 15
3.2 Initial state of the split-order map 16
3.3 Insertion into "list" 16
3.4 Inserting a key, value pair into the map 17
3.5 Initializing a bucket 17
3.6 The regular key lookup table 18
3.7 The highest-level specification of split-order map 18
3.8 Implementation of hashmap by split-order map 19

1 Introduction

Concurrent and distributed systems are extremely important in modern software development. Due to the difficulty of developing ever smaller and more powerful CPUs the trend in hardware design since about 2005 has been to increase the number of cores to allow for high levels of parallelization [1]. Additionally important areas of computing such as image processing and machine learning lend themselves well to such parallelization. [citation needed, Phuong?] At the same time software as a service and the massive scale industry giants like Amazon require a complex network of distributed systems to provide their functionality robustly and efficiently [2].

Hash tables are an important data structure for a variety of applications because they allow for data retrieval in constant time. Several lock-based hash tables for concurrent systems exist [citations], but the overhead of lock management and difficulty of resizing often make these impractical or inefficient [3]. A lock free alternative is proposed by Shalev and Shavit in [3]. This approach has proven to be useful [4] and scale better than lock-based approaches [5].

In both small- and large-scale computer systems it is important to ensure correctness. This is especially evident in critical infrastructure, but all scales and importance levels benefit from confidence in the correctness of their systems. [citation needed]

It is therefore troublesome that such systems are incredibly difficult to design, debug and reason about. The complexity of interactions between processes

and sheer number of possible edge cases makes it infeasible for a person to determine correctness.

Early solutions to this problem include Hoare [6], Floyd [7] and Pnueli's [8] temporal logics and Leslie Lamport's Temporal Logic of Actions [9] which seek to formalize the execution of programs in order to reason about them with logic. These formal methods proved useful, but laborious [10].

Building on the work in temporal logics, model checkers seek to minimize the human labor and ingenuity needed to prove correctness. This is done by specifying a model using some system of logic and then letting a model checker exhaustively survey the possible states of the system. This automates the process of proving correctness. One such model checker is the TLC model checker based on Lamport's TLA and incorporated in the TLA+ IDE.

1.1 Thesis

Shalev et al.'s split-ordered list design is a correct extensible hashmap for concurrent systems.

Furthermore it is possible to check this using a formal model checker, and the results of this will correspond to the properties proven by Shalev et al.

1.2 Method

In order to prove the correctness of the hashmap, its behavior will be implemented as a specification in TLA+ [11] and the TLC model checker will be used to test the claimed invariants.

The specification will be developed in stages of increasing granularity, assuming the atomicity of operations to begin with and gradually loosening assumptions.

1.3 Scope?

1.4 Outline

Chapter 2 discusses the motivation for formal verification and model checking, followed by a description of Temporal Logic in Section 2.1 and the TLA+ language in Section 2.2. Finally Shalev et al.'s hashmap design is described in Section 2.3.

Chapter 3 describes the specification of the hashmap in TLA+ and the development of this specification.

Finally **Chapter 4** describes the results of model checking and discusses these in relation to our thesis in Section 4.1.

2

Background

Model Checking: Algorithmic Verification and Debugging [10] In the Turing Lecture by the winners of the 2007 Turing Award, Edmund Clarke, Allen Emerson and Joseph Sifakis they describe the develpment and use of model checkers as a verification method for computer systems. Previous efforts to prove correctness had been focused on formal proofs which have three key shortcomings:

1. they require human ingenuity,
2. they are difficult to work with in concurrent and distributed systems,
3. they scale poorly with system size and complexity.

Instead, they propose algorithmic model checkers.

With this method a Temporal Logic is used to specify the correct behavior of a system and the model checker verifies that this behavior is not violated by exploring the state space of the model. Importantly, such model checkers produce a counter example – an example of incorrect behavior – which makes debugging and correcting the system easier. Key properties of a temporal logic are *expressiveness* and *efficiency*.

Model checking also scales poorly with system complexity, so several techniques are introduced to deal with "state space explosion"

- symbolic checking of ordered binary decision diagrams
- isolation of independent events in concurrent systems
- bounded checking by solving SAT
- reduce state space by increasing level of abstraction
  - if counterexamples are found a lower abstraction level is needed, but "good" properties hold through abstraction mappings

How Amazon Web Services Uses Formal Methods [2] Amazon's AWS services are all underpinned by large and complex distributed systems. This is necessary for high availability, growth and cost-effective infrastructure. Traditionally these systems have been tested by savvy engineers who know what to test and look for. However, some errors are very rare and will very likely slip through such testing. To catch these errors they employ model checking (with TLA+).

The PlusCal or TLA+ specifications work as a tool to bridge the gap between design and implementation. Designs are expressive, but inprecise while the implementation is precise, but hides overall structure. Through a choice of abstraction level, specifications can bridge this gap and provide both. An expressive specification also provides useful documentation of the system.

The key benefits of model checkers at Amazon are:

- a precisely specified design helps make changes and optimizations safely. This usage improves system understanding.
- they are faster than formal proofs
- a correct design and the understanding the specs provide promote better, more correct code.

2.1 Temporal Logics

The Temporal Logic of Programs [8] In The Temporal Logic of Programs [8], Amir Pnueli proposes a unified approach to the verification of both sequential and concurrent programs. His work seeks to unify approaches to to both, while also presenting a system that emulates the design intuition of programmers. The key concepts in this work are *invariance* – which covers par-

tial correctness, clean behavior, mutual exlusion and deadlock freedom – and *eventuality* – which generalizes these notions to cyclic programs and provides a special case of total correctness.

A dynamic discrete system is generalized as a three-tuple $\langle S, R, s_0 \rangle$ where S is the set of possible states, R a transition relation, and s_0 the initial state of the system. In order to make later constructions easier we further specify

$$s = \langle \pi, u \rangle$$

where π is the control component specifying the location in the program and u is the data component describing the state of any variables and data structures, and

$$R(\pi, u) = N(\pi, u) \wedge T(\pi, u)$$

where N describes the control flow and T the change in data such that a step in the execution may be described by

$$R(\langle \pi, u \rangle, \langle \pi', u' \rangle) \iff \pi' = N(\pi, u) \wedge u' = T(\pi, u)$$

To reason about concurrent programs we let states have multiple control components $s = \langle \pi_1, \pi_2, ..., \pi_n, u \rangle$ and randomly choose one control component to update in each step. Finally we let X be the set of all reachable states for the system. A predicate $p(s)$ is **invariant** if $p(s)$ is true $\forall s \in X$.

We can now start to define useful properties of the systems described in this way.

> **Partial correctness** is the claim that given the correct input, a program produces the correct output. We let $\phi(x)$ be the statement "reaching the end state $\implies$ (correct input $\implies$ correct output)". Partial correctness is equivalent to saying ϕ is an invariant.
>
> **Clean execution** means the program does not behave illegally, i.e it does not access illegal memory locations or divide by zero. We may define these restrictions as a predicate to make clean execution equivalent to this predicate being invariant.
>
> **Mutual exclusion**. Given a critical section C, mutual exclusion of the processes π_1 and π_2 is described by the invariance of the predicate $\neg(\pi_1 \in C \wedge \pi_2 \in C)$.

In addition to these properties we wish to reason about *temporal* implications. We let time be described by a $t \in \mathbb{N}$ and $H(p, t)$ denote the value of the predicate p at time t. We then introduce the temporal operator $p \rightsquigarrow q$ to mean

p eventually leads to q, or formally:

$$p \rightsquigarrow q : \forall t_1 \exists t_2 \; s.t \; t_1 \leq t_2, H(p, t_1) \implies H(q, t_2)$$

For all times t_1 there is a later time t_2 such that if p holds at t_1, q will hold at t_2. Armed with eventuality we can define temporally useful properties of systems.

Total correctness is stronger than partial correctness because it also requires that the program reaches an end state. We can express total correctness as $\langle \pi = l_0, u = \phi \rangle \rightsquigarrow \langle \pi = l_m, u = \psi \rangle$ where ϕ denotes correct input, and ψ denotes correct output and l_0, l_m are the start and end labeles of the system, respectively.

Accessibility is the guarantee that some segment S of a program can be reached. It can be expressed by $\pi = l_0 \rightsquigarrow \pi \in S$

Responsiveness. It is often desirable that some request r will be met by a response s. We call this responsiveness and describe it by $r \rightsquigarrow s$.

With these definitions under our belt, Pnueli defines the necessary axioms and inference rules to reason about the correctness of programs.

$$[\forall s, s' p(s) \wedge R(s, s') \implies q(s')] \Rightarrow p \rightsquigarrow q \tag{A1}$$

$$(p \implies q) \Rightarrow p \rightsquigarrow q \tag{A2}$$

Figure 2.1: Pnueli's axioms

These axioms define two ways to establish eventuality. A2 says that any logical implication is also an eventuality. A1 is a little more involved, but states that if for all consecutive states p being true in the first implies q being true in the second, then p eventually leads to q.

Using these axioms and inference rules as well as first-order logic, Pnueli goes on to formalize invariance and eventuality for sequential programs and concurrent programs.

Invariance of the predicate $q(\pi, u)$ is described by the conjunction $\bigwedge_i \pi = l_i \implies q(l_i, u)$, asserting that q is true at all points of execution. This method is called an *attachment* of the predicate to the program.

$$p \rightsquigarrow q, \forall s, s' r(s) \wedge R(s, s') \implies r(s') \Rightarrow (p \wedge r) \rightsquigarrow (q \wedge r) \quad \text{(R1)}$$

$$p \rightsquigarrow q, q \rightsquigarrow r \Rightarrow p \rightsquigarrow r \quad \text{(R2)}$$

$$p_1 \rightsquigarrow q, p_2 \rightsquigarrow q \Rightarrow (p_1 \vee p_2) \rightsquigarrow q \quad \text{(R3)}$$

$$p \rightsquigarrow q \Rightarrow (\exists u p) \rightsquigarrow q \quad \text{(R4)}$$

Figure 2.2: Pnueli's inference rules

In a concurrent program we generalize q to hold when any π_i is updated by N and construct either a full attachment

$$\bigwedge_{i_1, i_2, \ldots, i_n} (\pi_1 = i_1 \wedge \pi_2 = i_2 \wedge \ldots \wedge \pi_n = i_n) \implies q(\pi_1, .., \pi_n, u)$$

shown here for n concurrent execution threads, or the partial attachment

$$\bigwedge_i \pi_1 = i \implies q(\pi_1 = i, \pi_2, u) \wedge \bigwedge_j \pi_2 = j \implies q(\pi_1, \pi_2 = j, u)$$

shown here for two threads π_1 and π_2.

Eventuality is formulated as the temporal implication $\pi = l_1 \wedge p(u) \rightsquigarrow \pi = l_2 \wedge q(u)$. We can then describe the path between l_1 and l_2 by a finite sequence of steps and apply A1 to each step.

Finally, Pnueli introduces two new "tense operators" **F**uture and **G**lobal on predicates such that at some time n

$$F(p) = \exists t \geq n \; s.t \; H(t, p)$$

$$G(p) = \forall t \geq n \; H(t, p)$$

This lets us describe useful properties such as

$p \implies F(q)$ – if p is true now, then at some point in the future q will be true.

$G(p \implies F(q))$ – whenever p is true it will eventually be followed by a state in which q is true. (this is equivalent to $p \rightsquigarrow q$)

2.2 TLA+

2.3 Split-Ordered List Hashmap

Maybe a description of hashmaps in general here to set up the later use of table, list, key etc. or is that assumed knowledge?

Shalev et al. [3] present the first lock-free extensible hash table implemented using only loads, stores and atomic Compare and Swap (CAS).

Hashmaps are a key building block in many important systems [citation needed], but are difficult to implement concurrently. In particular, the resizing (extending) of the table is difficult to do atomically because at the very least a node must be moved from one list to another. In order to avoid conflicts and loss of data in this process some overhead is required which impacts performance.

The key insight of Shalev et al. is to flip the process upside down. Instead of moving nodes between buckets, they suggest moving the buckets among a statically ordered list of nodes. This requires an ordering of the list in which a bucket can always be split into two new buckets while their contents remain correct. A node should always reside in the bucket corresponding to its key $\mod 2^i$ where 2^i is the current size of the table.

Split-Ordered Lists are introduced to solve this problem. By sorting the keys according to their reversed binary representation Shalev et al. obtain a list which can be always be split into buckets $\mod 2^i$. This is because such an ordering corresponds to difference in the keys' ith least significant bit, which is equivalent to having a different remainder $\mod 2^i$.

In order to deal with the problems caused by removing nodes pointed to by hash table entries dummy nodes with the bucket value are introduced. These nodes signify the start of a bucket and are recursively initialized when an item is inserted into an unitialized bucket. To distinguish dummy nodes from regular nodes in the list, regular node keys have their most significant bit set to 1 before being reversed. This order and the structure of the map can be seen in Figure 2.3.

Insertion in to the map is done through atomic CAS instructions on the list. If a bucket is not initialized, a dummy node is created and inserted into the list before the new value is added as shown in Figure 2.4. The map is expanded by doubling the number of buckets and inserting new dummy nodes. Because of the split-ordering of the list, it is always possible to insert a new bucket by splitting an existing one. This process is shown in Figure 2.5.

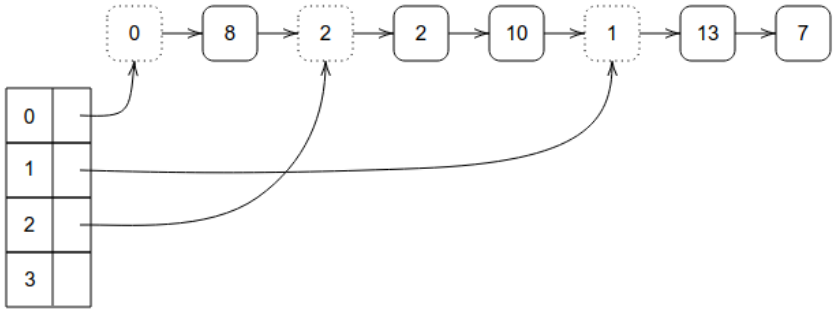
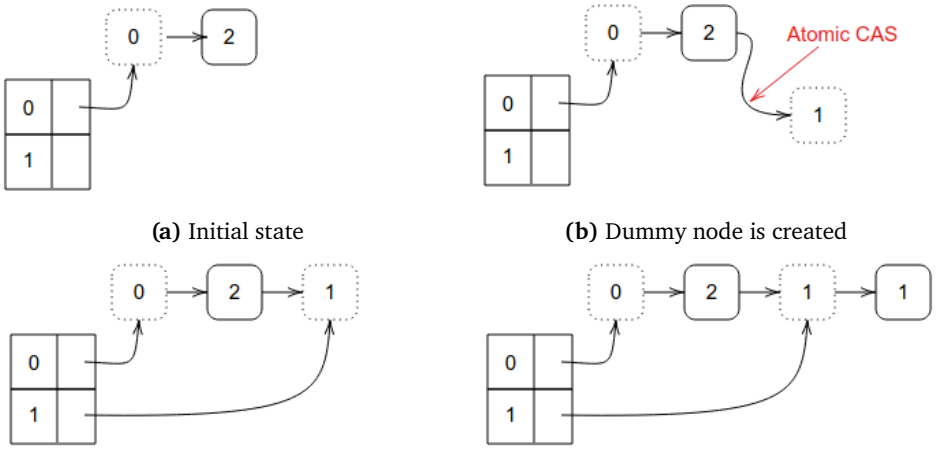


Figure 2.3: The layout of the split-ordered list



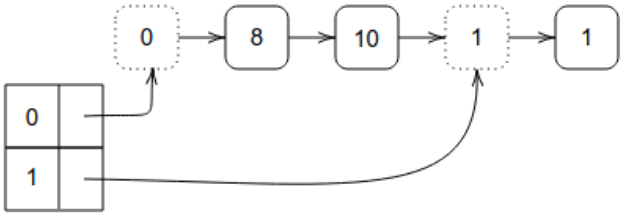
(a) Initial state

(b) Dummy node is created

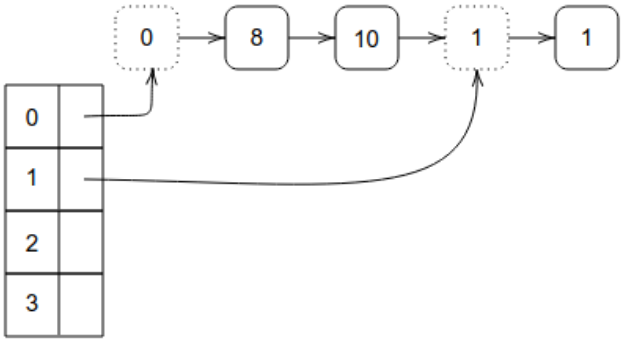
(c) Dummy node is inserted in bucket

(d) New node is inserted

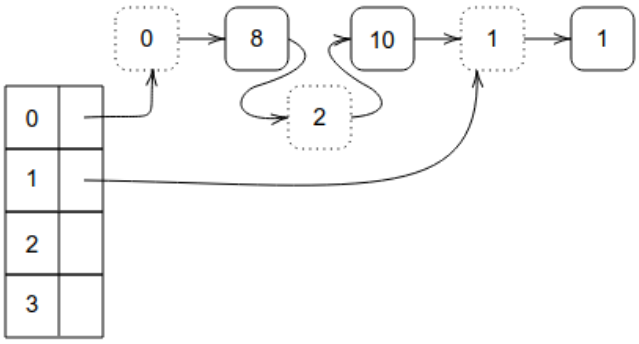
Figure 2.4: Insertion without bucket splitting



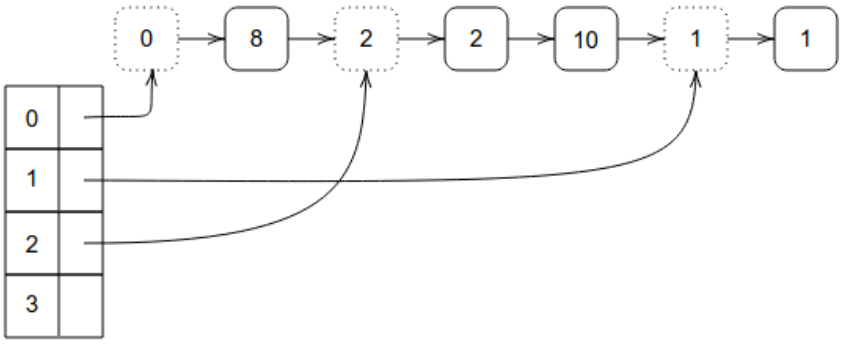
(a) Initial state



(b) Table is expanded



(c) Dummy node for new bucket inserted



(d) Table entry points to bucket node

Figure 2.5: Expansion and bucket splitting

3

TLA+ Specification

This chapter describes the structure of the TLA+ specification and the process of writing the specification. First we introduce the goals and scope of the specification, then describe the definitions of necessary data structures and operations. Finally the pieces are put together to summarize the full specification.

The specification consists of two main parts:

1. a generic hashmap and

2. an [insert name of structure here] *implementing* this map.

The generic specification describes the workings of a hashmap with insert and remove operations and ensures that this structure behaves as specified. The [insert name of structure] specification describes Shalev et al.'s specific structure and algorithms for implementing hashmap functionality.

3.1 Goal

The purpose of our specification is to verify the claims about correctness made by Shalev et al. Because TLA+ is designed for checking liveness and safety properties [12] we will not check claims about the performance of the implementation. This leaves 4 invariants:

- the list beginning at bucket o is always sorted
- if a bucket is initialized, then it points to a dummy node which is in the list beginning at bucket o
- if a key k is in the map, then `insert(k)` fails. Otherwise k is added to the map
- if a key k is in the map, then `remove(k)` removes it from the map. Otherwise it fails.

[MIGHT CHANGE:] Rather than having functions return error codes, the generic specification describes the correct behavior of each. This allows us to check invariants by implementation of this generic specification, since implementing it correctly corresponds to correct behavior. Additionally we will check *type safety*: at every point of execution, all keys and values are members of predefined key- and value-sets, respectively.

3.2 Hashmap for implementation

The generic hashmap specification seen in Figure 3.1 describes how a hashmap *should* behave. It maintains a set of keys *keys* and a mapping *map* from keys to values. Additionally it describes two operations: insert and remove.

Insert adds a key to the set of keys and changes one entry in the map. The key is added with a set union, which preserves key uniqueness. Updating the map uses the `EXCEPT` construct, to say "the map is the same except the new key maps to the new value". The insert action is always enabled since there will always be keys and values in PossibleKeys and PossibleValues.

Remove removes one key from the set and changes its mapping to NULL. This is done through the set difference operator and the same `EXCEPT` construct as in insert. Both operations are idempotent, so attempting to remove a key that is not in the map will result in no change.

The specification also contains temporal formulae that specify correct behavior. These have been checked with the TLC model checker [insert results here maybe?].

Finally the entire hashmap is described by *HashmapSpec*. This temporal formula states that the hashmap specification is fulfilled if the initial state fulfills *HashmapInit* and each action fulfills *HashmapNext*. This formula is used to

MODULE *hashmap*

This module describes a hashmap to be used for testing with Shalev et al.'s split-ordered list implementation of the data structure

EXTENDS *Integers*

CONSTANTS $NULL$, $PossibleKeys$, $PossibleValues$

VARIABLES $keys$, map

Initial state has empty map and no keys

$$HashmapInit \triangleq \land keys = \{\} \land map = [k \in PossibleKeys \mapsto NULL]$$

Insert changes exactly one mapping of the hashmap and adds one key to the set of keys

$$Insert \triangleq \exists\, k \in PossibleKeys : \exists\, v \in PossibleValues : \land keys' = keys \cup \{k\} \land map' = [map \text{ EXCEPT } ![k] = v]$$

Remove sets exactly one mapping to NULL

$$Remove \triangleq \exists\, k \in PossibleKeys : \land keys' = keys \setminus \{k\} \land map' = [map \text{ EXCEPT } ![k] = NULL]$$

Find returns the value stored in the map with key k

$$Find(k) \triangleq map[k]$$

Next is either an insert, a remove or a find Find(k) returns NULL if not in map, otherwise a value

$$HashmapNext \triangleq \lor Insert \lor Remove$$

TypeOK asserts all keys and values are of the right type

$$TypeOK \triangleq \forall\, k \in keys : \land k \in PossibleKeys \land map[k] \in PossibleValues$$

KeyHasValue asserts that every key is mapped to a value

$$KeyHasValue \triangleq \forall\, k \in keys : \neg(map[k] = NULL)$$

The hashmap specification as a temporal formula

$$HashmapSpec \triangleq HashmapInit \land \Box[HashmapNext]_{\langle keys,\, map\rangle}$$

Figure 3.1: The hashmap specification

```
EXTENDS Integers

CONSTANTS NULL, PossibleKeys, PossibleValues, LoadFactor, MaxSize

VARIABLES keys, list, buckets, size, count

ASSUME
    ∧ PossibleKeys ⊆ 0 .. 15
    ∧ NULL ∉ PossibleKeys
    ∧ NULL ∉ PossibleValues

SOInit ≜  ∧ keys    = {}
          ∧ list    = [n ∈ 0 .. 255 ↦ IF n = 0 THEN 0 ELSE NULL]
          ∧ buckets = [m ∈ PossibleKeys ↦ IF m = 0 THEN 0 ELSE NULL]
          ∧ size    = 1
          ∧ count   = 0
```

Figure 3.2: Initial state of the split-order map

```
ListInsert(k, v) ≜ IF list[k] = NULL
  THEN list' = [list EXCEPT ![k] = v] ∧ count' = count + 1
  ELSE UNCHANGED ⟨list, count⟩
```

Figure 3.3: Insertion into "list"

prove a more complex specificaiton implements the semantics of Hashmap.

The theorem $SOSpec \Rightarrow HashmapSpec$ states that any program following the split-order specification also follows the hashmap specification. Hence, any program following the split-order specification is a correct implementation of a hashmap. It is tested by checking the property *HashmapSpec* in a model checking SOSpec.

3.3 Split-ordered List

The split-order specification consists of three main parts. The necessary data structures (list, buckets), operations on the map, and an abstract specification tying them together. This section provides examples and explains the structure of each. The full specification can be found in Appendix 4.1.

Figure 3.2 shows the beginning of the specification. The sets of possible keys and values, as well as the maximum size and load factor of the map are parameters of the model. The variables for a list and buckets are set up, as well as the set of keys, and variables storing the number of entries and current number of

$$
\begin{array}{l}
BucketInsert(k,\ v) \triangleq \\
\text{Either a bucket needs to be initialized} \\
\lor \ \land\ buckets[k\%size] = NULL \\
\ \land\ BucketInit(k\%size) \\
\ \land\ ListInsert(SORegularKey(k),\ v) \\
\ \land\ keys' = keys \cup \{k\} \\
\quad \text{Or the bucket is already initialized} \\
\lor \ \land\ buckets[k\%size] \neq NULL \\
\ \land\ ListInsert(SORegularKey(k),\ v) \\
\ \land\ keys' = keys \cup \{k\} \\
\ \land\ \textsc{unchanged}\ \langle buckets \rangle
\end{array}
$$

Figure 3.4: Inserting a key, value pair into the map

$$
\begin{array}{l}
\textsc{recursive}\ BucketInit(\_) \\
BucketInit(b) \triangleq \textsc{if}\ buckets[Parent(b)] = NULL \land Parent(b) \neq 0 \\
\textsc{then}\ BucketInit(Parent(b)) \\
\textsc{else}\ \ buckets' = [buckets\ \textsc{except}\ ![b] = SODummyKey(b)]
\end{array}
$$

Figure 3.5: Initializing a bucket

buckets.
Their initial values are set such that there is one bucket pointing to the dummy node with 0 as its key and the set of real keys is empty.

The most involved operation is map insertion. This is done in the *BucketInsert* action shown in Figure 3.4. *BucketInserts* specifies the disjunction of having to initialize a bucket or inserting without changing buckets. In either case, a key is added to the set of keys and a node is inserted in the list.
BucketInit, shown in Figure 3.5 is a recursive function initializing a pucket and any of its unitialized parent buckets. The parent of a bucket is a dummy node defined such that it comes before the bucket both in the regular order and the split-order of keys.
Inserting into a list is done by *ListInsert* which inserts a node if there is not already one with the same key and otherwise does nothing. The list is represented as a mapping from split-order keys to values, preserving the order of nodes.

Keys and parent nodes are computed by Shalev et al. through bit operations and reversal. Such low-level operations are not available to us, so both are done with lookup tables. The lookup table for regular split-order keys is shown in Figure 3.6.

Lookup table for bit-reversed keys with MSB set

$$
\begin{array}{l}
SORegularKey(k) \triangleq \ \textsc{case}\ k = 0 \longrightarrow 1 \\
\quad \Box k = 1 \longrightarrow 9 \\
\quad \Box k = 2 \longrightarrow 5 \\
\quad \Box k = 3 \longrightarrow 13 \\
\quad \Box k = 4 \longrightarrow 3 \\
\quad \Box k = 5 \longrightarrow 11 \\
\quad \Box k = 6 \longrightarrow 7 \\
\quad \Box k = 7 \longrightarrow 15 \\
\quad \Box k = 8 \longrightarrow 1 \\
\quad \Box k = 9 \longrightarrow 9 \\
\quad \Box k = 10 \longrightarrow 5 \\
\quad \Box k = 11 \longrightarrow 13 \\
\quad \Box k = 12 \longrightarrow 3 \\
\quad \Box k = 13 \longrightarrow 11 \\
\quad \Box k = 14 \longrightarrow 7 \\
\quad \Box k = 15 \longrightarrow 15
\end{array}
$$

Figure 3.6: The regular key lookup table

$$
\begin{array}{l}
SOInsert \triangleq \ \land\ \exists\, k \in PossibleKeys : \\
\exists\, v \in PossibleValues : \\
\quad BucketInsert(k,\ v) \\
\qquad \land\ \textsc{unchanged}\ size \\
\\
SORemove \triangleq \ \land\ \exists\, k \in PossibleKeys : \\
\quad BucketRemove(k) \\
\qquad \land\ \textsc{unchanged}\ size \\
\\
SONext \triangleq \ \lor\ SOInsert \\
\quad \lor\ SORemove \\
\quad \lor\ BucketGrow \\
\\
SOSpec \triangleq \ SOInit \land \Box[SONext]_{\langle keys,\, list,\, buckets,\, size,\, count \rangle}
\end{array}
$$

Figure 3.7: The highest-level specification of split-order map

A refinement mapping of the hashmap spec with the map defined by the SOFind action

INSTANCE $hashmap$ WITH $map \leftarrow [k \in PossibleKeys \mapsto SOFind(k)]$

Split-order implements hashmap

THEOREM $SOSpec \implies HashmapSpec$

Figure 3.8: Implementation of hashmap by split-order map

Finally, the high-order functioning of the map is specified through *SONext*. *SONext* specifices that in each step either a key-value pair is inserted, a key is removed, or the bucket array grows in size. *SOSpec* then defines the total behaviour of the map to be a sequence of states that initially satisfies *SOInit* and satisfies *SONext* in all steps.

In order to show the implementation of hashmap by SplitOrder, a refinement mapping is defined. This is a way to describe the behavior of SplitOrder in terms of the variables of hashmap. The mapping replaces the variable *map* in hashmap with the *SOFind* operation defined in SplitOrder. It is necessary to perform such a replacement because SplitOrder does not define a simple map from keys to values, but rather uses the structure of buckets and list nodes to find the correct node. The refinement mapping ans subsequent implementation theorem can be seen in Figure 3.8.

4

Results

4.1 Discussion

Bibliography

[1] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. 4th. USA: Prentice Hall Press, 2014. ISBN: 013359162X.

[2] Chris Newcombe et al. "How Amazon Web Services Uses Formal Methods." In: *Commun. ACM* 58.4 (Mar. 2015), pp. 66–73. ISSN: 0001-0782. DOI: 10.1145/2699417. URL: https://doi.org/10.1145/2699417.

[3] Ori Shalev and Nir Shavit. "Split-Ordered Lists: Lock-Free Extensible Hash Tables." In: *J. ACM* 53.3 (2006), pp. 379–405. ISSN: 0004-5411. DOI: 10.1145/1147954.1147958. URL: https://doi.org/10.1145/1147954.1147958.

[4] Daniel Cederman et al. *Lock-free Concurrent Data Structures*. 2013. arXiv: 1302.2757 [cs.DC].

[5] Rodrigo Medeiros Duarte et al. "Concurrent Hash Tables for Haskell." In: *Programming Languages*. Ed. by Fernando Castor and Yu David Liu. Cham: Springer International Publishing, 2016, pp. 110–124. ISBN: 978-3-319-45279-1.

[6] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming." In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: https://doi.org/10.1145/363235.363259.

[7] Robert W. Floyd. "Assigning Meanings to Programs." In: *Proceedings of Symposium on Applied Mathematics* 19 (1967), pp. 19–32. URL: http://laser.cs.umass.edu/courses/cs521-621.Spr06/papers/Floyd.pdf.

[8] Amir Pnueli. "The Temporal Logic of Programs." In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. SFCS '77. USA: IEEE Computer Society, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32. URL: https://doi.org/10.1109/SFCS.1977.32.

[9] L Lamport. "Proving the Correctness of Multiprocess Programs." In: *IEEE Trans. Softw. Eng.* 3.2 (1977), pp. 125–143. ISSN: 0098-5589. DOI: 10.1109/TSE.1977.229904. URL: https://doi.org/10.1109/TSE.1977.229904.

[10] Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. "Model Checking: Algorithmic Verification and Debugging." In: *Commun. ACM* 52.11 (Nov. 2009), pp. 74–84. ISSN: 0001-0782. DOI: 10.1145/1592761.1592781. URL: https://doi.org/10.1145/1592761.1592781.

[11] Leslie Lamport et al. “Specifying and Verifying Systems with TLA+.” In: *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*. EW 10. New York, NY, USA: ACM, 2002, pp. 45–48. DOI: 10.1145/1133373.1133382. URL: http://doi.acm.org/10.1145/1133373.1133382.

[12] Jørgen Aarmo Lund. “Verification of the Chord protocol with TLA+.” In: (2019). URL: https://munin.uit.no/bitstream/handle/10037/15613/thesis.pdf?sequence=2%7B%5C&%7DisAllowed=y.

Split-order Specification

MODULE *SplitOrder*

This module implements a hashmap using Shalev et al.'s split-ordered list structure

EXTENDS *Integers*

CONSTANTS *NULL*, *PossibleKeys*, *PossibleValues*, *LoadFactor*, *MaxSize*

VARIABLES *keys*, *list*, *buckets*, *size*, *count*

ASSUME

$\land\ PossibleKeys \subseteq 0 \,..\, 15$

$\land\ NULL \notin PossibleKeys$

$\land\ NULL \notin PossibleValues$

The Init for split-order keys is initially empty the map maps every possible key to NULL
The list initially contains only the o dummy node

$SOInit \triangleq \quad \land\ keys = \{\}$

$\land\ list = [n \in 0 \,..\, 255 \mapsto \text{IF } n = 0 \text{ THEN } 0 \text{ ELSE } NULL]$

$\land\ buckets = [m \in PossibleKeys \mapsto \text{IF } m = 0 \text{ THEN } 0 \text{ ELSE } NULL]$

$\land\ size = 1$

$\land\ count = 0$

Lookup table for bit-reversed keys with MSB set

$SORegularKey(k) \triangleq$ CASE $k = 0 \rightarrow 1$

$\square k = 1 \rightarrow 9$

$\square k = 2 \rightarrow 5$

$\square k = 3 \rightarrow 13$

$\square k = 4 \rightarrow 3$

$\square k = 5 \rightarrow 11$

$\square k = 6 \rightarrow 7$

$\square k = 7 \rightarrow 15$

$\square k = 8 \rightarrow 1$

$\square k = 9 \rightarrow 9$

$\square k = 10 \rightarrow 5$

$\square k = 11 \rightarrow 13$

$\square k = 12 \rightarrow 3$

$\square k = 13 \rightarrow 11$

$\square k = 14 \rightarrow 7$

$\square k = 15 \rightarrow 15$

Lookup table for bit-reversed keys

$SODummyKey(k) \triangleq$ CASE $k = 0 \rightarrow 0$

$\square k = 1 \rightarrow 8$

$\square k = 2 \rightarrow 4$

$\square k = 3 \rightarrow 12$

$\square k = 4 \rightarrow 2$

$\Box k = 5 \longrightarrow 10$

$\Box k = 6 \longrightarrow 6$

$\Box k = 7 \longrightarrow 14$

$\Box k = 8 \longrightarrow 1$

$\Box k = 9 \longrightarrow 9$

$\Box k = 10 \longrightarrow 5$

$\Box k = 11 \longrightarrow 13$

$\Box k = 12 \longrightarrow 3$

$\Box k = 13 \longrightarrow 11$

$\Box k = 14 \longrightarrow 7$

$\Box k = 15 \longrightarrow 15$

Lookup table for parent buckets

$Parent(b) \triangleq \quad \text{CASE } b = 0 \longrightarrow 0$

$\Box b = 1 \longrightarrow 0$

$\Box b = 2 \longrightarrow 0$

$\Box b = 3 \longrightarrow 1$

$\Box b = 4 \longrightarrow 0$

$\Box b = 5 \longrightarrow 1$

$\Box b = 6 \longrightarrow 2$

$\Box b = 7 \longrightarrow 3$

$\Box b = 8 \longrightarrow 0$

$\Box b = 9 \longrightarrow 1$

$\Box b = 10 \longrightarrow 2$

$\Box b = 11 \longrightarrow 3$

$\Box b = 12 \longrightarrow 8$

$\Box b = 13 \longrightarrow 5$

$\Box b = 14 \longrightarrow 6$

$\Box b = 15 \longrightarrow 7$

Inserting into the "linked list"

$ListInsert(k,\ v) \triangleq \text{IF } list[k] = NULL$

$\text{THEN } list' = [list \text{ EXCEPT } ![k] = v] \land count' = count + 1$

$\text{ELSE UNCHANGED } \langle list,\ count \rangle$

Removing from the "linked list"

$ListRemove(k) \triangleq \text{IF } list[k] = NULL$

$\text{THEN UNCHANGED } \langle list,\ count \rangle$

$\text{ELSE } list' = [list \text{ EXCEPT } ![k] = NULL] \land count' = count - 1$

Recursively initializes buckets

$\text{RECURSIVE } BucketInit(\_)$

$BucketInit(b) \triangleq \text{IF } buckets[Parent(b)] = NULL \land Parent(b) \neq 0$

$\text{THEN } BucketInit(Parent(b))$

$\text{ELSE } buckets' = [buckets \text{ EXCEPT } ![b] = SODummyKey(b)]$

Find the value of the key k in the bucket b Results in the value if b is initialized and k is in b

$ListFind(b,\ k) \triangleq \text{IF } k > b \land list[b] \neq NULL \text{ THEN } list[k] \text{ ELSE } NULL$

SOFind finds a key in the map

$SOFind(k) \triangleq$ IF $buckets[k\%size] = NULL$

THEN $NULL$ should initialize bucket, but also needs a "return value"

ELSE $ListFind(buckets[k\%size],\ k)$

$Min(a,\ b) \triangleq$ IF $a > b$ THEN b ELSE a

$BucketGrow \triangleq$ IF $count \neq 0 \land size \div count > LoadFactor$

THEN $size' = Min(size * 2,\ MaxSize) \land$ UNCHANGED $\langle keys,\ list,\ buckets,\ count \rangle$

ELSE UNCHANGED $\langle keys,\ list,\ buckets,\ count,\ size \rangle$

Inserting into the buckets

$BucketInsert(k,\ v) \triangleq$ Either a bucket needs to be initialized

$\lor \land buckets[k\%size] = NULL$

$\land BucketInit(k\%size)$

$\land ListInsert(SORegularKey(k),\ v)$

$\land keys' = keys \cup \{k\}$

Or the bucket is already initialized

$\lor \land buckets[k\%size] \neq NULL$

$\land ListInsert(SORegularKey(k),\ v)$

$\land keys' = keys \cup \{k\}$

$\land$ UNCHANGED $\langle buckets \rangle$

Removing from the buckets

$BucketRemove(k) \triangleq \land ListRemove(SORegularKey(k))$

$\land keys' = keys \setminus \{k\}$

$\land$ UNCHANGED $\langle buckets \rangle$

$SOInsert \triangleq \ \land \ \exists\, k \in PossibleKeys :$

$\exists\, v \in PossibleValues :$

$BucketInsert(k,\ v)$

$\land$ UNCHANGED $size$

$SORemove \triangleq \ \land \ \exists\, k \in PossibleKeys :$

$BucketRemove(k)$

$\land$ UNCHANGED $size$

The Next for split order

$SONext \triangleq \quad \lor SOInsert$

$\lor SORemove$

$\lor BucketGrow$

Split-order spec

$SOSpec \triangleq SOInit \land \Box[SONext]_{\langle keys,\, list,\, buckets,\, size,\, count \rangle}$

If I can get map to work as intended...

refinement mapping??

INSTANCE $hashmap$ WITH $map \leftarrow [k \in PossibleKeys \mapsto SOFind(k)]$

Split-order implements hashmap

THEOREM $SOSpec \implies HashmapSpec$