TÉCNICO LISBOA

Formal Specification and Verification of the Lazy JellyFish Skip List

A Case Study in Iris on the Verification of Concurrent Maps with Version Control

Pedro Luís Ribeiro Carrott

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. João Fernando Peixoto Ferreira

Examination Committee

Chairperson: Prof. Luís Manuel Antunes Veiga
Supervisor: Prof. João Fernando Peixoto Ferreira
Member of the Committee: Prof. Ralf Jung

November 2022

“If you have time to fantasize about a beautiful end,
then just live beautifully until the end.”

— Gintoki Sakata

Acknowledgments

This thesis is the culmination of my hard work throughout the last 5 years, only made possible by the care, love and friendship from the ones most close to me. What I have learned through the years and the fond memories etched in my heart constitute a cornerstone for the person I have become.

To my advisor, João Ferreira, for introducing me to the programming languages research area, as I was fortunate enough to attend the curricular unit on programming languages lectured by him. The application of formal methods to software verification provides an attractive way to combine my mathematical reasoning with my interest in contributing to society. For that I am grateful to him, as well as for his guidance and useful comments during the development of this work and the writing of this thesis.

An acknowledgment must also be made to the acquaintances who began as colleagues and ended as friends, with whom I shared my best, as well as my lowest moments during this long journey. To Pedro Matono, my longest friend who has always been there for me at moments I would rather not mention in an official document with my name on it. To Tiago Gonçalves, for being the first friend I made in college, a friendship which I hope we will both carry throughout the course of our lives. To David Martin, for being the greatest person I know, may we once again overcome difficult challenges at the sound of Darude Sandstorm. To Bernardo Conde, for sharing the same passion in Japanese pop culture as me and providing free tech support whenever I showed complete inadequacy in handling a computer. To André Silva, for his humility and knowledge, strengths on which he does not face much competition. To Nuno Saavedra, for providing me the dankest moments in my life, as well as enlightening me on the power that memes hold over the hearts of people. To Rodrigo Antunes, for making me a more cultured person as a result of his friendship.

I would like to thank my mother for her constant support and providing me the opportunities to reach this stage in my life. As my second year was marked by the passing away of my father, the strength she showed and keeps showing has been a source of inspiration for me. I can only hope that my father would share the pride my mother has kindly expressed on my growth and achievements. I thank both of them for the life I have been able to enjoy thus far. A special mention to Ruby, the adorable little puppy who would not cease to cheerfully attack me as I heroically struggled to write this.

Last but not least, to Rick Astley, for never giving me up and for never letting me down.

Abstract

Concurrent append-only skip lists are widely used in data store applications, so as to maintain multiple versions of the same data with different timestamps, rather than delete outdated information. One such skip list implementation is JellyFish, which greatly mitigates the drop in performance witnessed in other skip lists induced by the append-only design. JellyFish accomplishes this feat by storing in each node a consistent timeline of values as a linked list, instead of inserting new nodes in the skip list.

In this work, we present a lock-based variant of JellyFish, using a lazy synchronization strategy, and formally verify its functional correctness. We further show that this data structure satisfies the specification of a concurrent map. To reason about concurrent updates on values, we define a novel resource algebra over timestamped domains. Using the `argmax` operator for this algebra, we prove that concurrent updates to the map always maintain the most recent values. We also show that updates to a node maintain its history of values consistent. Our proofs are mechanized in Coq using the concurrent separation logic of Iris.

Keywords

Separation Logic; Specification; Formal Verification; Concurrent Data Structures; Iris; Coq.

Resumo

As *skip lists* concorrentes mais utilizadas em aplicações de armazenamento de dados permitem apenas inserções, de modo a manter múltiplas versões dos dados com diferentes *timestamps*, ao invés de remover informação desatualizada. Uma implementação para este género de *skip lists* é a JellyFish, que consegue mitigar a perda de desempenho que se observa em outras implementações devido à não-remoção de nós. A JellyFish obtém esta melhoria ao guardar em cada nó uma lista representando a linha cronológica dos valores associados à chave, em vez de inserir novos nós na lista.

Neste trabalho, apresentamos uma variante da JellyFish baseada em trincos, usando uma estratégia de sincronização *lazy*, e verificamos formalmente a sua correção funcional. Mostramos ainda que a estrutura de dados satisfaz a especificação de um mapa concorrente. Para raciocinar sobre atualizações concorrentes a valores do mapa, definimos uma nova álgebra de recursos sobre domínios com *timestamps*. Usando o operador `argmax` para esta álgebra, conseguimos provar que escritas concorrentes no mapa mantêm cada chave associada ao seu valor mais recente. Mostramos também que estas escritas preservam a consistência da história de valores do nó atualizado. As provas estão mecanizados em Coq através da lógica de separação concorrente do Iris.

Palavras Chave

Lógica de Separação; Especificação; Verificação Formal; Estruturas de Dados Concorrentes; Iris; Coq.

Contents

1 Introduction **1**

1.1 Contributions 3

1.2 Organization of the Document 4

2 Background **5**

2.1 Skip Lists 7

2.1.1 The JellyFish Skip List 8

2.2 Iris 9

2.2.1 Separation Logic 9

2.2.2 Modalities 10

2.2.3 Weakest Precondition 11

2.2.4 Invariants 12

3 A Lazy Variant of JellyFish **15**

3.1 Data Initialization 17

3.2 Data Retrieval 18

3.3 Data Updates 19

4 Reasoning about Timestamped Domains **21**

4.1 Ghost State in Iris 23

4.1.1 Resource Algebras 23

4.1.2 Frame-preserving Updates 24

4.1.3 Authoritative Ghost State 25

4.2 The `argmax` Resource Algebra 25

4.2.1 Map Composition 25

4.2.2 Value Composition 26

4.2.3 Map Insertion 27

4.2.4 Local Updates on Maps 27

5 **Specification for the Lazy JellyFish Skip List** 29
5.1 Rules and Hoare Triples . . . 31
5.2 Representation Predicate . . . 33
5.3 Invariant for the Bottom List . . . 34
5.3.1 Partial Views . . . 34
5.3.2 Set Membership . . . 36
5.3.3 Invariant Definition . . . 36
5.3.4 Lock Resources . . . 37
5.4 Invariant for Sublists . . . 37
5.4.1 Sublist Relation . . . 38
5.4.2 Ghost Tokens . . . 38
5.4.3 Invariant Definition . . . 40
5.5 Value Updates . . . 40
5.5.1 Vertical List . . . 41
5.5.2 Local Fragment . . . 41
5.6 Coq Formalization . . . 43

6 **A Simple Client** 47
6.1 Code Overview . . . 49
6.2 Proof Overview . . . 50
6.3 Timestamp Assumptions . . . 52

7 **Discussion and Related Work** 53
7.1 Concurrent Data Structures in Iris . . . 55
7.1.1 Logical Atomicity . . . 55
7.1.2 Concurrent Skip Lists . . . 56
7.2 Mechanized Verification of Skip Lists . . . 57
7.3 Specifications for Concurrent Maps . . . 57
7.3.1 Key-value Specifications . . . 57
7.3.2 Client Reasoning . . . 60

8 **Conclusion** 65
8.1 Future Work . . . 67

Bibliography 69

List of Figures

2.1 Skip list traversal 7
2.2 The JellyFish skip list 8

3.1 Pseudocode for the lazy JellyFish skip list 18

5.1 Specification for the lazy JellyFish skip list 32
5.2 Definition of the representation predicate and invariants 34
5.3 Authoritative ghost state for a linked list 35
5.4 Example of an incorrect implementation of `put` 35
5.5 Ghost state relating consecutive levels 38
5.6 Specification for the `putH` procedure 39
5.7 Specification for the `update` procedure 40
5.8 Representation predicate and invariant from the Coq formalization 43
5.9 Alternative definition for the representation predicate and invariant 45

6.1 Client program with concurrent writes 49
6.2 Verified client using the map specification 50

7.1 Logically atomic key-value specification for timestamped values 61
7.2 Verified client using the key-value specification 62

Acronyms

AFP Archive of Formal Proofs

PCM Partial Commutative Monoid

RA Resource Algebra

1

Introduction

Contents

1.1 Contributions 3
1.2 Organization of the Document 4

A map is an abstraction for a data structure which associates an identifying key to each value it stores. This abstraction has been vastly studied in computer science holding many real-world use cases. For instance, data store applications make use of *concurrent* maps to index data in a thread-safe environment. In fact, most of these applications maintain a history of values for each key in the map, so as to record different versions of the same data, instead of deleting outdated information. This append-only design, however, tends to hamper the performance of these concurrent maps, depending on how the old values are stored.

As an abstraction, maps can be implemented in several ways, with different assurances on their performance. While self-balancing trees are a classical approach, a more efficient implementation is the skip list. Structurally, skip lists are very similar to balanced trees but maintain their balance through a probabilistic strategy, rather than explicit rebalancing procedures. This difference makes the skip list preferable in concurrent environments, which is why it is the most widely used map implementation in data store applications. In particular, JellyFish [32] is a state-of-the-art skip list implementation whose performance surpasses that of other skip lists used in industry. As an append-only skip list, JellyFish efficiently supports version control by storing a list of values in every node. This list corresponds to the node's history of values with each value being assigned to a timestamp. To ensure that chronological order is maintained, new values are never added to the list if their assigned timestamps are less recent than any value already in the list.

Concurrent implementations, however, tend to be more error-prone than other sequential solutions. Due to the non-deterministic scheduling of operations, concurrency makes it harder to reason about the state of shared data. Poor synchronization between threads can lead to errors in the logic of the program, such as returning incorrect results to queries or introducing inconsistencies to data when performing updates. To quell these concerns, formal verification of concurrent data structures has gained traction in recent years. Through the use of formal methods, correctness and functionality can be proven for concurrent operations on shared resources. Not only do formally verified data structures provide assurances to any client program on how the data is handled, but concise specifications can also be used to prove the correctness of the programs themselves in a modular manner. Defining a general enough specification to cover all use cases is therefore one of the main challenges regarding the formal verification of concurrent data structures.

1.1 Contributions

In this work, we verify the functional correctness of an original lock-based variant of JellyFish. This implementation adopts a lazy synchronization strategy, which makes it easier to reason about its correctness. Our work proves that this skip list satisfies the specification of a concurrent map with timestamped

values, which we use to verify an illustrative client program. Using the concurrent separation logic of Iris [16–18,20], we reason about updates to the map by defining a novel resource algebra for the `argmax` operator. This operator abstracts the expected behaviour for map updates, since it always retains the value with the maximum argument, which in this context corresponds to the value with the most recent timestamp. Our proofs are mechanized in Coq and available at:

https://github.com/sr-lab/iris-jellyfish

To the best of our knowledge, this is the first effort to verify (a variant of) the JellyFish skip list and to reason about concurrent maps with version control through timestamped domains. The main contributions of this work can thus be summarized as follows:

- The first verification effort of the JellyFish design for concurrent append-only skip lists;
- A new concurrent map specification, which supports version control through the use of timestamps;
- A mechanized proof that our skip list implementation satisfies the concurrent map specification;
- A novel resource algebra in the concurrent separation logic of Iris for the `argmax` operation.

1.2 Organization of the Document

This thesis is organized as follows. We begin by presenting in Chapter 2 some general information about skip lists and Iris. In Chapter 3, we show and explain our lazy JellyFish skip list implementation. In Chapter 4, we abstract from the concrete implementation and discuss how we can use Iris to reason about concurrent maps with timestamped values. In Chapter 5, we define a concurrent map specification and explain how we can prove that our implementation satisfies such a specification. Chapter 6 contains a proof sketch for an example client using our concurrent map specification. Finally, we discuss in Chapter 7 the contributions of our work by comparison with related work, leaving some concluding remarks in Chapter 8.

2

Background

Contents

2.1 Skip Lists 7
2.2 Iris 9

We begin this chapter with an overview of skip lists, followed by a high-level description of the JellyFish design for concurrent append-only skip lists. We then introduce Iris, the Coq framework that we use to reason about our lazy variant of JellyFish.

2.1 Skip Lists

A skip list [25] is a list-based data structure, which can be considered a generalization of a sorted linked list. This data structure is composed of a maximum number of levels, where level 0 corresponds to the complete linked list and each level l is a sublist of level $l-1$. Since each level contains progressively fewer elements of the original list, maintaining all lists sorted allows searches in higher levels to skip elements that would otherwise be traversed in a standard linear search.

In Figure 2.1, a 3-level skip list is shown, illustrating how a key search is performed. The colored arrows in the figure are numbered according to the order by which a thread checks them when searching for key 17. The green arrows represent the path taken by a thread, while the red arrows represent excluded paths, since these would skip over the node with key 17. The first green arrow can skip over the node with key 5, since any key higher than 13 will also be higher than 5.

The search begins at the head of the list in the top level and will continue in the same level as long as the searching thread keeps finding keys lower than 17. Whenever the *next* key in the current level is higher than 17, the thread descends to the next level and continues the search from the node with the *current* key. If the key is not found upon reaching the bottom level, then we can conclude that it does not belong to the skip list. Otherwise, the search can stop in any level, as soon as the key is found.

Each level can be implemented as its own linked list, with each key having a different physical node per level. A more efficient implementation consists in storing an array of successor pointers for each node, without duplicating the same key. If the skip list is used to store key-value pairs, then the array-based implementation also avoids updating the value of multiple nodes, which is particularly useful when you transition to a concurrent context.

Concurrent skip lists are easier to implement than concurrent self-balancing trees, while offering better performance for the same functionality. Herlihy *et al.* [14] state that this advantage is a result of the skip list's probabilistic nature, which keeps the data structure balanced through all levels, while trees require periodic rebalancing operations that can serve as a bottleneck. Skip lists solve the balancing

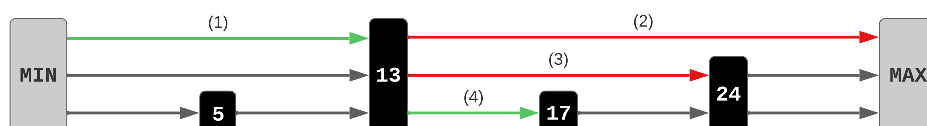


Figure 2.1: Skip list traversal

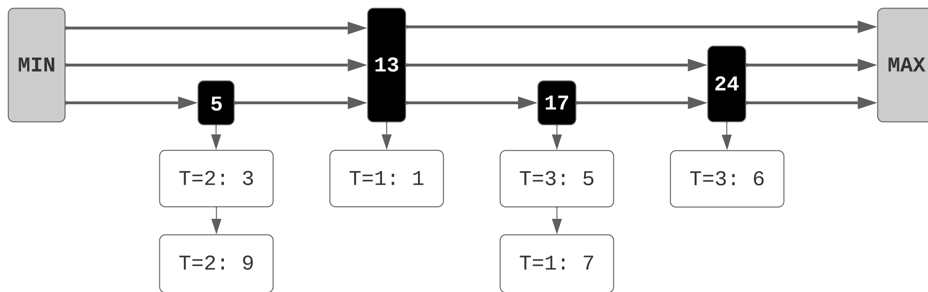


Figure 2.2: The JellyFish skip list

problem by choosing from a probabilistic distribution the height for a new node. Traditionally, the heights follow a geometric progression over a fixed probability, which is expected to converge to a balanced structure. However, the correctness of the different skip list algorithms is agnostic to the considered height distribution, so we do not consider its probabilistic nature in this work.

2.1.1 The JellyFish Skip List

Several concurrent skip lists have been proposed, including append-only structures which are widely used in data store applications (*e.g.*, RocksDB[1] and LevelDB[2]). Rather than overwrite existing values, this design choice maintains a record of all values that have been stored during the data structure's lifetime. However, this append-only nature can induce a heavy cost in performance depending on the chosen implementation.

One approach is to insert additional nodes with different timestamps, which greatly degrades performance, since traversals through the skip list have to visit more nodes as updates are made to the structure. A more efficient approach is to maintain a list of timestamped values per node, referred to as the node's *vertical list*. Updates to a key are then made by prepending a new value to its vertical list, maintaining the skip list itself with the same nodes. This approach corresponds to the design of the JellyFish skip list [32].

The skip list in Figure 2.2 illustrates the JellyFish design. Each node contains its own vertical list reflecting the history of values associated to the node's key. This history remains consistent by keeping the values sorted by timestamp, as we see in key 17. The node's most recent value 5 from timestamp 3 is found at the head of the list, preceded by value 7 which was inserted at timestamp 1. The node with key 5 also contains a history with two values, both with timestamp 2, since JellyFish allows value updates on a key as long as the new value is, at least, as recent as the current value. Chronological order is thus maintained, ensuring that concurrent updates do not introduce inconsistencies in the data.

[1] https://github.com/facebook/rocksdb
[2] https://github.com/google/leveldb

JellyFish is one of many lock-free concurrent skip list algorithms. However, while a non-blocking solution is desirable to reduce contention, this approach makes it harder to reason about the correctness of these algorithms. For instance, in the `ConcurrentSkipListMap` from the Java concurrency package[3], "*certain interleavings can cause the usual skip-list invariants to be violated*" [13]. For this reason, we consider a lock-based skip list algorithm, which follows the design of JellyFish and employs the lazy synchronization strategy of the lazy list of Heller *et al.* [12]. Inspired by the concurrent skip list of Herlihy *et al.* [13], the key idea behind the algorithm is to treat each level as an individual lazy list. The implementation for this lazy JellyFish skip list is discussed in detail in Chapter 3.

2.2 Iris

To verify the correctness of the lazy JellyFish skip list we have mechanized proofs for its methods in the Coq proof assistant using the Iris proof mode [19,21]. We now provide an overview on some of the main features of Iris. For a more detailed description of the logic, the reader may refer to Jung *et. al.* [17].

Iris is a concurrent separation logic that allows reasoning about deep correctness properties for fined-grained concurrent programs written in higher-order imperative languages [17]. In other words, Iris allows reasoning about programs containing parallel composition and synchronization mechanisms between threads, such as **CAS** instructions or fine-grained locking. Moreover, Iris allows reasoning about a wide variety of programming languages that support concurrency, being parameterized by the considered language. A HeapLang implementation of our skip list algorithm is considered in this work.

HeapLang[4] is a programming language provided by the Iris framework as an example on how to reason about the logic. It is an untyped higher-order λ-calculus and it possesses enough of the features that are present in most imperative programming languages that allow reference manipulation and concurrency. HeapLang has the usual allocation, load and store operations for dealing with dynamic memory allocation and control. The language also provides a fork directive for the creation of threads, as well as synchronization primitives such as **CmpXchg** (compare-and-exchange) and **FAA** (fetch-and-add).

2.2.1 Separation Logic

Iris is built on top of separation logic [24, 26], which allows reasoning about resource management. Although many abstractions may fit this notion of resource (*e.g.*, time), allocated memory in the heap provides a natural way to understand the semantics for resources. With this in mind, separation logic introduces three main constructs: the *points-to* connective $\hookrightarrow$, the *separating conjunction* $*$ and the *separating implication* $-\!\!*$ (also referred to as magic wand).

[3] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentSkipListMap.html
[4] https://gitlab.mpi-sws.org/iris/iris/blob/master/docs/heap_lang.md

The *points-to* assertion $l \hookrightarrow x$ expresses the mapping between a heap cell l and its current contents x. This assertion also implies ownership of said heap cell, *i.e.*, exclusive read and write privileges for that particular heap fragment. This notion of ownership is particularly relevant when reasoning about fine-grained concurrency, since we may want to consider that different threads have exclusive access to disjoint fragments of the heap.

The way in which we can express this disjunction of resources is through the use of the *separating conjunction* $*$. An assertion $P * Q$ describes the resources which can be separated in two disjoint fragments, with one fragment satisfying P and another satisfying Q. Although the standard conjunction $\wedge$ from propositional logic also requires both P and Q to be satisfied simultaneously, this notion of resource separation is what distinguishes both conjunction operands from each other. An important observation we make is that $P \nvdash P * P$ if P is not persistent (we discuss persistence of propositions further ahead), while $P \vdash P \wedge P$. For example, we cannot have $l \hookrightarrow x * l \hookrightarrow x$, since l is a single heap cell and cannot be disjoint into two separate fragments.

The separating conjunction allows us to reason about local resources using the following rules:

$$\textsc{Frame}\quad \frac{\{\,P\,\}\quad e\quad \{\,Q\,\}}{\{\,P * F\,\}\quad e\quad \{\,Q * F\,\}} \qquad\qquad \textsc{Par}\quad \frac{\{\,P_1\,\}\quad e_1\quad \{\,Q_1\,\}\qquad \{\,P_2\,\}\quad e_2\quad \{\,Q_2\,\}}{\{\,P_1 * P_2\,\}\quad e_1 \,||\, e_2\quad \{\,Q_1 * Q_2\,\}}$$

In Frame, the resource described by F is referred to as a frame. Since it appears in both the pre and postconditions of the Hoare triple, the rule allows us to remove those resources from the context, using only P to prove Q. This approach allows us to apply local reasoning on the execution of e by only considering the resources which are directly affected by the program.

Local reasoning is also expressed in Par, a rule which extends standard separation logic with concurrent reasoning: we can reason about the concurrent execution of programs e_1 and e_2 by splitting disjointly the resources between both threads. On termination, both programs must provide disjoint resources which satisfy the required postcondition when combined.

Finally, we have the *separating implication* $-\!\!*$, which is analogous to the standard implication $\Rightarrow$ from propositional logic. For resources R_1 that satisfy $P -\!\!* Q$, if we have resources R_2 disjoint from R_1 that satisfy P, then combining R_1 with R_2 we obtain resources that satisfy Q. In other words, we have that $P * (P -\!\!* Q) \vdash Q$, which is analogous to *modus ponens* from propositional logic [8].

2.2.2 Modalities

Dietrich defines modalities as "*expressions that qualify assertions about the truth of statements*" [8]. We will now discuss about three modalities that the logic of Iris provides: the *persistence* modality $\square$, the *later* modality $\triangleright$ and the *update* modality $\Rrightarrow$.

Although separation logic allows us to reason about exclusive ownership of resources, not all propositions express this notion and may therefore hold true for all threads, regardless of how resources are distributed between them. Such propositions are considered *persistent* and we express the persistence of a proposition P as $\square P$. Since these propositions hold for any resource distribution, we can duplicate these propositions as we see fit.

The assertion $x = 2$ is persistent, as it establishes an equivalence between the variable x and the value 2, rather than describing a resource which can be owned. On the other hand, $l \hookrightarrow x$ is not persistent, since it expresses ownership of the resource named l, which cannot be duplicated to a disjoint set of resources (*i.e.*, $l \hookrightarrow x * l \hookrightarrow x$ is not possible).

Step-indexing is introduced in Iris through the use of the *later* modality $\triangleright$: the proposition $\triangleright P$ expresses that P holds true only at the *next step-index*, *i.e.*, only after a reduction step is taken. This idea is expressed in the following rule:

$$\textsc{Ht-rec}\quad \frac{\{\, P \,\} \quad e\,[(\mathtt{rec}\, f(x) = e)/f,\; v/x] \quad \{\, \Phi \,\}_{\mathcal{E}}}{\{\, \triangleright P \,\} \quad (\mathtt{rec}\, f(x) = e)\; v \quad \{\, \Phi \,\}_{\mathcal{E}}}$$

The rule states that we first take a reduction step on the function application before introducing P as a precondition. The later modality is a key feature of the logic for reasoning about invariants, as we will discuss further ahead.

We also need to reason about resource updates that may invalidate current assertions. In Iris, the *update* modality $\Rrightarrow$ allows us to define a proposition $\Rrightarrow P$, meaning that the current resources can be updated to a new state that satisfies P, as long as the update is *frame-preserving*. An update is said to be frame-preserving when it does not invalidate assertions about other resources (*i.e.*, a frame) disjoint from the updated resources.

2.2.3 Weakest Precondition

Separation logic extends from Hoare logic, meaning that proof rules are defined in the form of Hoare triples. In Iris, a Hoare triple is a proposition of the form $\{P\}\; e\; \{v.\, Q\}_{\mathcal{E}}$ or, alternatively, $\{P\}\; e\; \{\Phi\}_{\mathcal{E}}$, where $\Phi : Val \rightarrow Prop$. For each triple, propositions P and Q represent the pre and postconditions, respectively, for executing program e and, since triples are defined as propositions themselves, Iris allows the definition of nested triples. However, triples in Iris contain an additional parameter $\mathcal{E}$ called a *mask*, which serves as a namespace for the invariants that can be opened. As we will see soon, invariants are given names and each invariant must only be used once before closing it, so keeping this mask is crucial for assuring soundness.

The base definition for Hoare triples in Iris, however, is that of a *weakest precondition*. The weakest precondition assertion $\mathtt{wp}_{\mathcal{E}}\; e\; \{\Phi\}$ removes the precondition from the context, since "*it is often easier and*

more direct to use the weakest precondition assertion instead of (derived) Hoare triples" [2]. Weakest preconditions can be used to define Iris Hoare triples as follows:

$$\{P\}\, e\, \{\Phi\}_{\mathcal{E}} \triangleq \Box(P \twoheadrightarrow \mathtt{wp}_{\mathcal{E}}\, e\, \{\Phi\})$$

The persistence modality is required to make the Hoare triple duplicable and, since P is defined as the required precondition for e to terminate with a return value that satisfies Φ, then combining these persistent resources with resources that satisfy P will guarantee the weakest precondition for e to meet its specification.

2.2.4 Invariants

We now discuss how we can reason about shared resources in Iris. Having already introduced persistent propositions and how these can be duplicated for any resource, we turn our attention to *invariants*. In Iris, an invariant is a proposition of the form $\boxed{I}^{\mathcal{N}}$, where I is an Iris proposition and $\mathcal{N}$ is the namespace associated to this invariant. Invariants describe the state of shared resources among all threads, so all invariants must be persistent. However, to use these shared resources, a thread must momentarily claim exclusive ownership of them and perform any operation over those resources locally. This process corresponds to *opening* the invariant and its semantics are formalized in the following rule:

$$\frac{\{\triangleright I * P\} \quad e \quad \{v.\, \triangleright I * Q(v)\}_{\mathcal{E}\setminus\mathcal{N}} \qquad \mathtt{atomic}(e) \qquad \mathcal{N} \subseteq \mathcal{E}}{\boxed{I}^{\mathcal{N}} \vdash \{P\} \quad e \quad \{v.\, Q(v)\}_{\mathcal{E}}}\ \textsc{Inv}$$

When we open the invariant to perform an operation over the corresponding resources, we must assure that the invariant still holds afterwards. Hence, the rule requires the proposition used as an invariant to hold locally both in the pre and postconditions.

However, we note that the rule requires $\triangleright I$ to hold and not I, which is due to the impredicativity of the logic. By impredicativity we mean the possibility of defining nested invariants (since invariants themselves are propositions) and without the later modality this impredicativity would yield the logic of Iris unsound [17].

Additionally, the rule only allows an invariant to be open once, as we observe from the mask in both triples. When we open the invariant to reason about the operation locally, we remove the namespace $\mathcal{N}$ from the mask, so as to assure that the invariant is not reopened, since further inconsistencies may arise otherwise [17].

Another restriction is that the operation on these shared resources must be atomic, since "*the rule allows us to temporarily use and even break the invariant, but after a single atomic step (i.e., before any other thread could take a turn), we have to establish it again"* [17]. In other words, by considering a single step, we can avoid reasoning about other threads interfering with its execution, assuring that the invariant will be preserved throughout all steps of the program, no matter how many times it is opened.

Although the rule assures us that invariants are preserved throughout a program's lifetime, we need to consider that concurrent operations might still affect the results of each thread depending on the order of execution. We thus need to reason about how these operations can be composed in a consistent manner, which is possible in Iris through ghost state. We provide a detailed description of Iris' ghost state in Chapter 4 when discussing how to reason about the lazy JellyFish skip list as a concurrent map implementation.

3

A Lazy Variant of JellyFish

Contents

3.1 Data Initialization 17
3.2 Data Retrieval 18
3.3 Data Updates 19

Lazy JellyFish is an append-only skip list implementation where the data is organized in memory following the JellyFish design [32], while concurrent updates to it employ a lock-based lazy synchronization strategy. Its implementation is shown in Figure 3.1, highlighting with a solid contour the usual operations on maps: `new`, `get` and `put`. A dashed contour highlights the private `update` procedure, which is responsible for updating a node's vertical list. To match the JellyFish design, a node for this data structure is represented as a tuple (k, v, l, n) where:

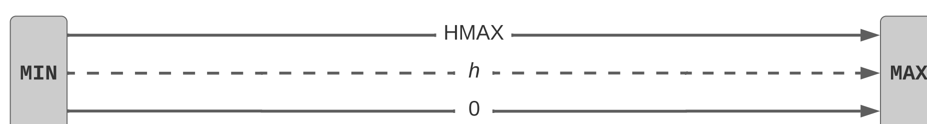
- k is an integer for the node's key;
- v is a pointer for the node's vertical list;
- l is an array for the node's locks in each level;
- n is an array for the node's successors in each level.

In turn, the vertical list of a skip list node contains nodes represented as tuples (v, t, p) where:

- v is a value;
- t is the assigned timestamp;
- p is a pointer to the previous value's node.

3.1 Data Initialization

The skip list's construtor, `new`, initializes the data structure with two nodes, referred to as the left and right sentinels, with keys `MIN` and `MAX`, respectively; these values bound the range of valid keys for the map. Both arrays of the left sentinel are created with `HMAX`$+1$ entries using the **`AllocN`** primitive. All entries of the successor array point to the right sentinel, while `initLocks` creates a different lock for each entry of the lock array. Pointer arithmetic is used to index array entries: we write $a{+}i$ instead of the standard C-style notation $a[i]$. The right sentinel, on the other hand, has no successors or locks, since keys will always be lower than `MAX`, remaining constant throughout the skip list's lifetime. Since the sentinels do not correspond to actual entries of the map, no value is associated to these nodes, *i.e.*, these nodes have no vertical list. The constructor returns a pointer to the left sentinel. The following diagram illustrates the initial state of the skip list, where the dashed arrow may refer to any level of height h between 0 and `HMAX`:



```
initLocks locks lvl ≜
  if lvl = HMAX + 1 then ()
  else   locks + lvl ← newlock ();
         initLocks locks (lvl + 1)
```

```
new ≜
  let tail = (MAX, NULL, NULL, NULL) in
  let next = AllocN (HMAX + 1) (ref tail) in
  let locks = AllocN (HMAX + 1) () in
    initLocks locks 0;
    ref (MIN, NULL, locks, next)
```

```
find pred k lvl ≜
  let succ = !!(pred.next + lvl) in
    if k ≤ succ.key then (pred, succ)
    else find succ k lvl
```

```
findAll pred k lvl h ≜
  let (pred, succ) = find pred k lvl in
    if lvl = h then (pred, succ)
    else findAll pred k (lvl − 1) h
```

```
get p k ≜
  let (_, succ) = findAll !p k HMAX 0 in
    if k ≠ succ.key then None
    else let v = !(succ.val) in Some (v.val, v.ts)
```

```
link pred lvl n ≜
  let new = !n in
    new.next + lvl ← !(pred.next + lvl);
    new.locks + lvl ← newlock ();
    pred.next + lvl ← n
```

```
createAndLink pred k v t h ≜
  let next = AllocN (h + 1) () in
  let locks = AllocN (h + 1) () in
  let val = ref (v, t, NULL) in
  let n = ref (k, val, locks, next) in
    link pred 0 n;
    n
```

```
update node v t ≜
  let val = !(node.val) in
    if t < val.ts then ()
    else node.val ← (v, t, ref val)
```

```
findLock pred k lvl ≜
  let (pred, _) = find pred k lvl in
  let lock = !(pred.locks + lvl) in
    acquire lock;
    let succ = !!(pred.next + lvl) in
      if k ≤ succ.key then (pred, succ)
      else   release lock;
             findLock succ k lvl
```

```
insert pred lvl n ≜
  let k = !n.key in
  let (pred, _) = findLock pred k lvl in
  let lock = !(pred.locks + lvl) in
    link pred lvl n;
    release lock
```

```
tryInsert pred k v t h ≜
  let (pred, succ) = findLock pred k 0 in
  let lock = !(pred.locks) in
    if k = succ.key
    then   update succ v t;
           release lock;
           None
    else   let n = createAndLink pred k v t h in
             release lock;
             Some n
```

```
insertAll curr k v t h lvl ≜
  if lvl = 0 then tryInsert curr k v t h
  else   let (pred, _) = find curr k (lvl − 1) in
         let opt = insertAll pred k v t h (lvl − 1) in
           match opt with
           | None     ⇒   None
           | Some n   ⇒   insert curr lvl n;
                          Some n
```

```
putH p k v t h ≜
  let (pred, _) = findAll !p k HMAX h in
  insertAll pred k v t h h
```

```
put p k v t ≜
  let h = randomLevel in
  let _ = putH p k v t h in ()
```

Figure 3.1: Pseudocode for the lazy JellyFish skip list

3.2 Data Retrieval

The `get` method performs a search for a key and returns its current value. Being parameterized by the left sentinel pointer p and the lookup key k, it first invokes `findAll` to traverse the skip list as it would in a sequential setting. The `find` procedure iterates over a single level, returning a pair of nodes with the greatest key lower than k and the lowest key equal to or greater than k within the current level. Starting from the top level, `findAll` calls `find` to obtain the current level's pair and continues the search in the next level starting from the first node of the returned pair. The traversal terminates once the nodes from the bottom list are obtained.

The pair of nodes from the bottom list is then returned by `findAll`, containing the last node in the skip list with key lower than k and the first node with key equal to or greater than k, following a case analysis for the latter's key. If the key differs from k, then k is not in the map and so there is no value associated to k. Otherwise, k belongs to the map's domain and its most recent value is stored in the head of its vertical list. We then retrieve the value and timestamp from this vertical list node.

Given that concurrent updates to the map might occur, the `get` method is optimistic in the sense that it might return outdated results. However, a correct result might still be invalidated by a concurrent write taking place immediately after the search's conclusion, regardless of the chosen implementation. Consider the following operations on a skip list stored in p:

$$
\left.\begin{array}{l}
\mathtt{let}\ opt = \mathtt{get}\ p\ 10\ \mathtt{in} \\
\ \mathtt{match}\ opt\ \mathtt{with} \\
\ \mid \mathtt{None} \quad\quad\ \Rightarrow\ () \\
\ \mid \mathtt{Some}(v,t)\ \Rightarrow\ \mathtt{put}\ p\ 10\ (v{+}1)\ (t+1)
\end{array}\right\|\quad \mathtt{put}\ p\ 10\ 5\ 1
$$

The left thread attempts to increment the value of key 10 (if it exists), while the right thread simply updates the same key with some arbitrary value 5. If the `get` operation terminates before the right thread's update takes place, then opt will return the current value v stored in the skip list. However, the right thread might immediately invalidate v by overwriting it with 5. As such, the left thread's increment will be based on an old value, rather than the new value 5, leading to potentially inconsistent results. As this issue will affect any implementation, it is preferable to opt for a non-blocking approach, since it reduces contention, making the `findAll` procedure suitable as a helper function for the `put` operation.

3.3 Data Updates

Concurrent writes on the map are done through the `put` method, which can insert new nodes or update existing ones. Updating a key k consists in associating it to a new value v and timestamp t. In case the key is not found, a new node must be created with a given height h chosen from a probabilistic distribution through `randomLevel`. The random height h is then passed to `putH` along with the remaining arguments to execute the map update. If an insertion occurs, then the new node's memory address is returned by `putH`. Although this return value is ignored by `put`, it is not immediately ignored by `putH` in order to reason about properties of the procedure, as we will see in Chapter 5.

Since the levels above h will not contain the new node, `findAll` initially traverses the skip list until level h. The `insertAll` procedure is then called to either insert a new node in all remaining levels or update an existing node. To preserve the sublist relation between levels, the traversal continues until the bottom level, so that insertions may be executed bottom up. If a new node is inserted in the bottom list, then this node is propagated to the upper levels for insertion; otherwise, an existing node, already connected to all intended levels, has its value and timestamp updated upon reaching the bottom level.

To ensure that insertions and updates are done correctly, both `insert` and `tryInsert` first call `findLock`. Lazily, `findLock` tries to acquire the only lock it needs to perform the intended insertion or update. It first executes `find` to obtain the node with the greatest key lower than k in the current level and then acquires its lock. Since we only obtain exclusive ownership of the lock *after* acquiring it, we check if the node's successor still has a key equal to or greater than k. If not, the node no longer holds the greatest key lower than k; `find` is then continued from the successor to obtain a new node, repeating the process. Otherwise, we have acquired exclusive ownership of the lock and the successor's key is guaranteed to be equal to or greater than k.

If the keys are different, `tryInsert` will create a node for the new key-value pair. While the list insertion is performed through the `link` procedure at all levels, the new node is only created at the bottom level. Furthermore, we only validate the successor's key at the bottom level. Due to the sublist relation, if k is not in the bottom list, then it is not in any of its sublists. It is thus safe to invoke `insert` in the upper levels without validating the keys, as long as there is only one thread performing the insertions. This is guaranteed by having all threads reaching the bottom level first to claim the right for creating and inserting the new node. The threads which acquire the lock after the node has been inserted will see that the successor of the locked node has key k, so they will attempt to update its value.

The sequence of values in a node's vertical list should reflect a monotonic increase of the associated timestamps. Therefore, `update` fails when the node's timestamp is more recent than t, since it would yield an inconsistent timeline. On the other hand, a successful update occurs when t is more recent, as well as when the timestamps are equal. A new node containing v and t is then prepended to the head of the vertical list, extending the current history of values of the node.

4

Reasoning about Timestamped Domains

Contents

4.1 Ghost State in Iris 23
4.2 The argmax Resource Algebra 25

We now focus on reasoning about the correctness of the lazy JellyFish skip list. A key feature of Iris that allows us to reason about mutable shared state is its capability of defining, through *resource algebras*, an abstract or auxiliary state of the program, to which we call *ghost state*. In this chapter, we describe ghost state and present a novel resource algebra for reasoning about values with timestamps.

4.1 Ghost State in Iris

Concurrent operations on shared data can lead to different states, depending on the order of execution and the interleavings between atomic steps of each operation. However, analysing all possible combinations does not scale when we are dealing with several threads and operations per thread. A better approach is to reason about operations locally, without taking into account the whole picture, *i.e.*, independently of what other threads might be doing.

Ghost state provides a way of doing this by matching the physical state of shared data with an abstract state where certain properties must hold. For instance, if we choose to model this abstract state as a Partial Commutative Monoid (PCM), we can ensure that operations on the shared state are commutative and associative. These two properties are useful when reasoning about concurrency, because they imply that, for any set of operations, the order of execution is irrelevant, eliminating the need for a combinatorial analysis.

4.1.1 Resource Algebras

Ghost state in Iris is defined as a Resource Algebra (RA), a broader algebraic definition than PCMs. Akin to PCMs, RAs possess a carrier (or domain) $\mathcal{M}$ and a composition operator $(\cdot)$. The differences between both constructs are present in two other properties that define a PCM: partiality and the unit element. Partiality is a useful notion, since it allows us to express that a given operation might be undefined between certain elements; in RAs, operations are total, but some combinations of elements are deemed invalid through a predicate $\overline{\mathcal{V}}$, capturing a similar effect. Regarding the unit element, PCMs require a single unit for all elements, while RAs generalize this notion by requiring a unit (if any) for each element, defined by a core function. If the core is the same for all elements, then we obtain a unital RA, which only differs from a PCM in terms of partiality. Absence of a core is denoted by the element $\perp$.

To consolidate these ideas, we give the set union RA as an example. We first define the carrier for this RA as all sets over a given type. We then define the set union as the operation for this RA, which is possible since it is a commutative and associative operation. The core function can be defined as the empty set for all elements, meaning that this RA is actually a unital RA. Finally, we define all combinations as valid for this operation, since the standard set union does not impose restrictions on its operands.

To truly understand the role of validity in RAs, we can consider the *disjoint* set union RA. Unlike with the standard set union, the disjoint set union is only allowed between sets where their intersection equals the empty set. Thus, if both sets contain some element in common, then the RA must deem these pairings as invalid. On all other aspects, both RAs are essentially the same.

The RA operator can be used to compose ghost state through the following rule:

$$\boxed{a}^{\gamma} * \boxed{b}^{\gamma} \dashv\vdash \boxed{a \cdot b}^{\gamma}$$

The dashed lines denote that we are dealing with a ghost variable (an auxiliary variable, rather than an in-memory program variable) associated to the ghost name γ, while the separating conjunction is used to express the ownership of disjoint *ghost* resources, which can be owned separately by different threads. Through this rule, separate resources which correspond to the same ghost variable can be combined into a single resource using the operation of the underlying RA. We can thus reason about changes to resources locally and the RA handles how those local changes can be combined to reflect the global state.

4.1.2 Frame-preserving Updates

If we consider the set union RA, then we have that $a \cdot x = a \cup x$ and we can see that performing an update on a, using the RA operator, entails an update on the full set, *i.e.*, for any a', we have that $(a \cup a') \cup x = (a \cup x) \cup a'$. We can do this update, because the thread that owns x is not affected by the changes, since all elements in x will still be present in the full set. In other words, the update on a is *frame-preserving*, as it does not disturb the frame containing x. Formally, a frame-preserving update can be defined as:

$$a \rightsquigarrow b \triangleq \forall\, x \in \mathcal{M} \uplus \{\bot\}.\, \overline{\mathcal{V}}(a \cdot x) \Rightarrow \overline{\mathcal{V}}(b \cdot x)$$

In other words, a can be updated to b, as long as no frame x is affected by the update. Naturally, we have that $\{a, b\} \subseteq \mathcal{M}$. The $\bot$ element is included[1] in the set of possible values for x to allow updates on elements with no frame. We can further expand this definition to that of a *local* update such that:

$$(a_1, b_1) \rightsquigarrow_{\mathsf{L}} (a_2, b_2) \triangleq \forall\, x \in \mathcal{M} \uplus \{\bot\}.\, \overline{\mathcal{V}}(a_1) \wedge a_1 = b_1 \cdot x \Rightarrow \overline{\mathcal{V}}(a_2) \wedge a_2 = b_2 \cdot x$$

A local update requires a stronger constraint on the frame x, since it enforces x to yield a_i by composition with b_i when attempting to update both a_1 and b_1. The relevance of local updates becomes clear in the context of authoritative ghost state.

[1] $\uplus$ stands for the *disjoint* set union

4.1.3 Authoritative Ghost State

An authoritative RA is built on top of some other RA, being constituted by an *authoritative* resource (preceded by $\bullet$) and a *fragment* resource (preceded by $\circ$). Through the underlying RA, a fragment resource can be decomposed into multiple fragments by applying the fragment composition rule:

$$\circ\,(f_1 \cdot f_2) = \circ\, f_1 \cdot \circ\, f_2$$

The rule also allows fragments to be combined such that the combination of all fragments yields the indivisible authoritative fragment. The relation between the authoritative and the fragment resources can thus be expressed through the rule:

$$\boxed{\bullet\, a \cdot \circ\, f}^{\gamma} \vdash \exists\, x \in \mathcal{M} \uplus \{\bot\}.\, a = f \cdot x$$

In other words, a fragment f must be included in the authoritative element a. This is a useful property to reason about concurrency, since it enables us to reason about a shared authoritative resource while each thread possesses its own private fragment. Updates to a private fragment should thus reflect updates to the authoritative resource, which is where local updates play their role.

Frame-preserving updates for an authoritative RA require that updating a fragment has no effect on other fragments. On one hand, those fragments should constitute a frame to our update. On the other hand, the authoritative resource should be updated such that it remains equal to the combination of all fragments. Therefore, if the frame fragments complete the initial fragment to obtain the authoritative resource, then the same frame should complete the updated fragment to obtain the new authoritative resource. This notion coincides with the local update definition, meaning that a frame-preserving update is defined as:

$$\frac{(a_1, f_1) \rightsquigarrow_{\mathsf{L}} (a_2, f_2)}{\bullet\, a_1 \cdot \circ\, f_1 \rightsquigarrow \bullet\, a_2 \cdot \circ\, f_2}$$

4.2 The `argmax` Resource Algebra

Our goal is to verify an implementation for a concurrent map. Since ghost state must represent an accurate abstraction of the data we are reasoning about, we require a suitable RA over maps. To define such a RA, we need to first understand how composition between maps should be applied.

4.2.1 Map Composition

For maps with no keys in common, we can simply merge both maps into a single map with all key-value pairs, similarly to the set union. However, when both maps have some key in common, the associated value in each map might differ from one another. In this scenario, what value should the key possess?

One way of handling this issue would be to invalidate such combinations, just like we did with the disjoint set union RA. Another approach would be to combine both values, following the composition rule for singleton maps, where $\{ k : v \}$ expresses a mapping of key k to a value v:

$$\{ k : x \} \cup \{ k : y \} = \{ k : x \cdot y \}$$

For this rule to be applicable, we need to make value composition possible, which means that the carrier of our map RA must map keys of a given type to values that belong to *another* RA. This leaves us with a new question: what value RA makes sense in the context of our problem? If two threads put different values for the same key, then what value should the key be mapped to? The answer to this question will depend on the timestamps of each insertion.

4.2.2 Value Composition

The map should always store the value with the most recent timestamp. If both timestamps are equal, then both values will be prepended to the key's history, but their relative order will depend on the scheduler. Otherwise, the value with the most recent timestamp will become the new head of the key's vertical list, while the least recent insertion will only succeed if it gets scheduled first. In other words, combinations between values yield the value (or one of the values) with the *maximum* timestamp, which means that we will need to define a RA for the `argmax` operation.

To the best of our knowledge, our work is the first to formalize the `argmax` RA and use it in the verification of concurrent maps. The main complication with this operation consists in how one should handle when multiple arguments have the maximum value. We address this concern by defining the carrier for our RA as pairs between *sets of arguments* and *values*. We can then define the RA operator such that combining two pairs yields the pair with the maximum value. If the values are equal, then a new pair is returned, containing the same value and the union between both arguments. Finally, all combinations are defined as valid and a `botZ` element is added to the carrier to serve as unit. The `argmax` operator is thus defined such that for all sets of arguments (a and b) and values (i and j):

$$\begin{cases} (a, i) \cdot (b, j) = (b, j) \text{ if } i < j \\ (a, i) \cdot (b, i) = (a \cup b, i) \\ (a, i) \cdot \texttt{botZ} = (a, i) \end{cases}$$

In the context of maps with timestamped values, this novel RA will ensure that each key is associated to the timestamp at which its most recent update occurred, as well as to a set of values, which were all inserted with the key's associated timestamp. We require this set, rather than the actual value stored in memory, to keep track of every value that might be at the head of the key's vertical list. Since updates with the same timestamp will result in a non-deterministic ordering of operations, any of those updates can be the last one to be prepended to the list.

4.2.3 Map Insertion

Updates to a map can also be made through explicit *insertion* of new values rather than map composition. The map $\langle k : v \rangle m$ is a result of replacing the value of key k in m with value v. Both operations on maps are related through the following rules:

$$\frac{m[k] = \texttt{None}}{m \cup \{\ k : v\ \} = \langle k : v \rangle m} \qquad \frac{m[k] = \texttt{Some}(v_i)}{m \cup \{\ k : v\ \} = \langle k : v_i \cdot v \rangle m}$$

If the m does not contain key k, then insertion and composition are equivalent. Otherwise, the inserted value must be the composition between the existing value v_i and the new value v. In simpler terms, we can perform the insertion by letting value composition determine what value should be stored. This equality can be proven as follows:

$$\begin{aligned} m \cup \{\ k : v\ \} = \langle k : v_i \rangle (m \setminus \{k\}) \cup \{\ k : v\ \} = (m \setminus \{k\}) \cup \{\ k : v_i\ \} \cup \{\ k : v\ \} = \\ = (m \setminus \{k\}) \cup \{\ k : v_i \cdot v\ \} = \langle k : v_i \cdot v \rangle (m \setminus \{k\}) = \\ = \langle k : v_i \cdot v \rangle m \end{aligned}$$

We first begin by replacing m with $\langle k : v_i \rangle (m \setminus \{k\})$, as removing k from m and then inserting its initial value v_i yields the original map. Since k is not in $m \setminus \{k\}$, we can replace insertion with composition, combine both singleton maps and replace composition with insertion. Finally, insertion will overwrite any existing value for k, so we can just replace $m \setminus \{k\}$ with m.

4.2.4 Local Updates on Maps

The introduction of a `botZ` element to the carrier of the `argmax` RA stems from the local update rules on maps. Since we need to reason about concurrent updates on maps, the authoritative RA enables us to reason about the contributions of each thread separately, through the application of local updates. However, the requirements for performing local updates on maps will vary depending on the key's presence (or absence) in the considered maps.

If the key is not in the authoritative map, then it cannot be in any existing fragment. As such, a thread can insert a key k into the authoritative map m_A and its own fragment m_F, as long as the associated value x is valid with respect to the value RA. This idea is expressed by the rule:

$$\frac{m_A[k] = \texttt{None} \quad \overline{\mathcal{V}}(x)}{(m_A, m_F) \rightsquigarrow_{\mathsf{L}} (\langle k : x \rangle m_A, \langle k : x \rangle m_F)}$$

Conversely, if the authoritative map contains the key, then some existing fragment must also contain that map entry. However, a thread performing an update on that key might not be in possession of that

fragment. It is therefore necessary to consider the scenario where the local fragment possesses an entry for the key, as well as when it does not.

If the updating thread is in possession of some fragment which contains an entry for the key, then we only need to assure that the new values for the key (a and f) in the authoritative and fragment maps can be obtained by performing a local update on the initial values in those maps (a_i and f_i). Both maps can thus be (locally) updated such that:

$$\frac{m_A[k] = \texttt{Some}(a_i) \quad m_F[k] = \texttt{Some}(f_i) \quad (a_i, f_i) \rightsquigarrow_{\mathsf{L}} (a, f)}{(m_A, m_F) \rightsquigarrow_{\mathsf{L}} (\langle k : a \rangle m_A, \langle k : f \rangle m_F)}$$

Alternatively, if the thread's local fragment has no entry for the key, then we do not have an f_i element for the pair required by local updates. For this reason, the local update rule for this scenario assumes the existence of a unit element ε for the pair, since a unit is compatible with any frame by definition. Such an update can take place by applying:

$$\frac{m_A[k] = \texttt{Some}(a_i) \quad m_F[k] = \texttt{None} \quad (a_i, \varepsilon) \rightsquigarrow_{\mathsf{L}} (a, f)}{(m_A, m_F) \rightsquigarrow_{\mathsf{L}} (\langle k : a \rangle m_A, \langle k : f \rangle m_F)}$$

It is due to this rule that we include `botZ` in our carrier. Without a unit element, we would not be able to prove local updates for this scenario. However, we still need to define a local update rule for the `argmax` RA. A general rule for local updates, which suffices for the purposes of this work, states that you can simply add the same frame z to both values in the authoritative and fragment resources:

$$\frac{}{(x, y) \rightsquigarrow_{\mathsf{L}} (x \cdot z, y \cdot z)}$$

5

Specification for the Lazy JellyFish Skip List

Contents

5.1 Rules and Hoare Triples 31
5.2 Representation Predicate 33
5.3 Invariant for the Bottom List 34
5.4 Invariant for Sublists 37
5.5 Value Updates 40
5.6 Coq Formalization 43

Ghost state allows us to reason about concurrent maps in the abstract world of RAs. We now show how these abstractions can help us in reasoning about the physical state of a concurrent map and define a specification for its operations. We begin by describing the high-level specification and then present in detail its underlying definition.

5.1 Rules and Hoare Triples

We require a *representation predicate* to describe the known state of the map: due to the chosen implementation for this map, we refer to the representation predicate as `IsSkipList`. On one hand, this predicate refers to the physical state of the map, so it should be parameterized by the pointer for the left sentinel of the skip list which implements the map; on the other hand, it must reflect the abstract state of the map, so it should also be parameterized by the abstract map that matches the physical layout of the structure.

In a concurrent setting, however, threads won't always have full knowledge of the map's state. For instance, when two threads insert concurrently in different positions of the map, they are not aware of what the other thread is doing, because they are handling separate data. Thus, we need to reason about the map with only partial knowledge at our disposal, meaning that the `IsSkipList` predicate should describe a partial view (or fragment) of the map, rather than the full view. We can then use the map RA operator to perform local updates on partial views and combine these partial views to obtain a more complete view of the map. The SkipSep rule shown in Figure 5.1 is what allows us to combine fragments in this manner, as well as to share ownership of the map between threads by splitting views into fragments.

`IsSkipList` is defined as follows. The first parameter is a pointer to the head of the skip list and the second parameter is a map with partial knowledge of the full map. The third parameter indicates whether we are in possession of the full view: it corresponds to a fraction ranging between 0 (exclusive) and 1 (inclusive). Full knowledge of the map's state corresponds to a fraction of 1, while splitting a view results in views with smaller fractions. The full view can only be obtained without sharing ownership, since excluding fractions of 0 ensures that it can only be split into fragments with a fraction lower than 1. The fourth parameter provides the required ghost names, as views can only be combined if they refer to the same ghost state.

Our concurrent append-only map contains three public methods: `new`, `get` and `put`. In Figure 5.1, we show the Hoare triples for each operation using the `IsSkipList` predicate. The specification for `new` (SkipNew) is rather straightforward: no resources are needed as a precondition and creating the skip list returns the full fraction of an empty map. The other operations, however, possess more complex semantics.

SkipSep
$$\texttt{IsSkipList}(p, M_1, q_1, \gamma) * \texttt{IsSkipList}(p, M_2, q_2, \gamma) \dashv\vdash \texttt{IsSkipList}(p, M_1 \cup M_2, q_1 + q_2, \gamma)$$

SkipNew
$$\frac{}{\{\,\texttt{True}\,\}\quad \texttt{new}\quad \{\,p.\, \exists\, \gamma.\, \texttt{IsSkipList}(p, \varnothing, 1, \gamma)\,\}}$$

SkipGet
$$\frac{\texttt{MIN} < k < \texttt{MAX}}{\{\,\texttt{IsSkipList}(p, M, 1, \gamma)\,\}\quad \texttt{get}\, p\, k \quad \left\{ v^?.\ \begin{array}{l} \texttt{IsSkipList}(p, M, 1, \gamma) * ((v^? = \texttt{None} * M[k] = \texttt{None}) \vee \\ (\exists\, v, S, t.\, v^? = \texttt{Some}(v, t) * M[k] = \texttt{Some}(S, t) * v \in S)) \end{array} \right\}}$$

SkipPut
$$\frac{\texttt{MIN} < k < \texttt{MAX}}{\{\,\texttt{IsSkipList}(p, M, q, \gamma)\,\}\quad \texttt{put}\, p\, k\, v\, t \quad \{\,\texttt{IsSkipList}(p, M \cup \{k : (\{v\}, t)\}, q, \gamma)\,\}}$$

Figure 5.1: Specification for the lazy JellyFish skip list

Searches in our concurrent skip list are optimistic, meaning that, if some thread is searching for a key that another thread is inserting or updating, the searching thread might return an outdated result. For this reason, the specification for `get` (SkipGet) is only defined for the scenario where we have full ownership of the data structure. Since we know that no other thread is in possession of some map fragment, we can be certain that no concurrent write will interfere with the search. We can then prove that searching in the data structure for any key within the valid key range is equivalent to performing a lookup for the same key in the abstract map. Using the `argmax` RA for value composition, we can show that the key's set of values stored in the abstract map necessarily contains the value returned by the search. In other words, the sets of values in the abstract map encompass all possible results for the search method. We will now see how the `put` specification (SkipPut) ensures that none of these sets contain more than their possible results.

The `put` method attempts to update a key with a given value and timestamp such that the key is within the valid key range. If the key has not yet been inserted into the map, then a new node with the given key, value and timestamp will be created and linked to all levels within its height range, updating the physical state of the map. To express this change in the abstract state, the postcondition of SkipPut simply combines the view we have of the map with a singleton map that associates the key to a singleton set containing only the given value (*i.e.*, the only value that we know is associated to the key) and to the given timestamp. Since we know that the key is not in the map, adding a new key to our partial view is equivalent to adding it to the full view. While it is simpler to understand why this singleton composition works when we insert a new key, this update to the abstract map also works when we attempt to update an existing key.

Under the assumption that the key already exists in the map, we can infer that the abstract map results from a composition with some singleton map for that same key. Furthermore, while the key may not exist in our partial view, we know that updates to partial views are equivalent to updates to the full view. Thus, that singleton map, which we extract from the full view, will be combined with the singleton

map from the update to our partial view. We can then apply the composition rule for singleton maps and obtain a new singleton map for the same key, with its value being the combination between the existing value and the new one. Using the `argmax` RA, this combination retains the values with the most recent timestamp, which is the intended behaviour for updates to a node's vertical list.

When both timestamps are equal, a set with both values is kept in the abstract map, even though `update` will prepend the new value to the vertical list. To understand why this is correct we need to consider that we are dealing with *concurrent* updates. Although we know that the thread that comes last will be the one to define the key's new value, we do not know the order by which these updates occur. Combining values in this manner ensures that the abstract map will always associate a key to every value that might be at the head of its vertical list, regardless of the update order.

We note that SkipPut is a very simple specification when compared to SkipGet. This is due to the abstraction provided by Iris' ghost state. The map RA perfectly abstracts the `put` operation, while the specification for `get` is defined based on this abstraction. In other words, `get` does not possess its own abstraction, leading to a more convoluted and less readable specification, as opposed to `put`.

5.2 Representation Predicate

To define the `IsSkipList` predicate, we first need to consider that operations on the skip list can be decomposed into local operations on the linked list of each level. This property allows us to reason about the correctness of the whole data structure by reasoning about each level independently. Thus, we can define a new predicate to describe the invariant resources for a given skip list level, while `IsSkipList` simply asserts ownership of the resources for all levels.

However, while all levels possess the same physical structure, the bottom level is the only one which contains all key-value pairs of the map. Since both `put` and `get` perform a traversal until the bottom, all reasoning about insertions, updates and lookups on the map can be done at this level; the other levels serve only to guide traversals through the optimal path. Additionally, the bottom level is the only level that is not a sublist of some other level. Therefore, we define the `IsSkipList` predicate using two distinct invariant definitions: one for the bottom list and another for its sublists.

The `IsSkipList` predicate is thus defined using a unique invariant for each level, as shown in Figure 5.2. The invariants are represented by a solid border: the `BotListInv` predicate holds the shared resources for the bottom list, while the `SublistInv` predicate describes the upper levels. The parameter p should point to the left sentinel and the corresponding points-to assertion is made persistent ($\square$), since it will always point to the same node. The parameter γ is a list containing the names for the ghost variables of each level. The parameters M and q are used in the assertion within the dashed border to express the partial view of the map, as we will now see by describing the bottom list invariant.

Representation Predicate

$$\texttt{IsSkipList}(p, M, q, \gamma) \triangleq \exists\, head.\ p \hookrightarrow_{\square} head * head.\texttt{key} = \texttt{MIN} *$$

$$\left[\circ_q M\right]^{\gamma_F^0} * \boxed{\texttt{BotListInv}(head, \gamma^0)}^{\texttt{levelN}(0)} * \mathop{\ast}_{i=1}^{\texttt{HMAX}} \boxed{\texttt{SublistInv}(i, head, \gamma^i, \gamma^{i-1})}^{\texttt{levelN}(i)}$$

Invariants

$$\texttt{BotListInv}(head, \gamma) \triangleq \exists\, M, S, L.$$

$$\left[\bullet M\right]^{\gamma_F} * \left[\bullet S\right]^{\gamma_A} * \left[\texttt{KeyRange} \setminus S.\texttt{keys}\right]^{\gamma_T} *$$

$$M.\texttt{keys} = S.\texttt{keys} * S \equiv_P L * \texttt{Sorted}(L_{\texttt{cat}}) *$$

$$\mathop{\ast}_{i=0}^{|L|} \left(\begin{array}{l} \texttt{IsNext}(0, L_{\texttt{cat}}[i], L_{\texttt{cat}}[i+1]) * \\ \texttt{HasLock}(0, L_{\texttt{cat}}[i], \texttt{InBotLock}) \end{array} \right) *$$

$$\mathop{\ast}_{n \in S} \left(\exists\, v, vs.\ \begin{array}{c} n.\texttt{val} \hookrightarrow_{1/2} v * v.\texttt{val} \in vs * \\ M[n.\texttt{key}] = \texttt{Some}(vs, v.\texttt{ts}) \end{array} \right)$$

$$\texttt{SublistInv}(lvl, head, \Gamma, \gamma) \triangleq \exists\, S, L.$$

$$\left[\bullet S\right]^{\Gamma_A} * \left[\texttt{KeyRange} \setminus S.\texttt{keys}\right]^{\Gamma_T} *$$

$$S \equiv_P L * \texttt{Sorted}(L_{\texttt{cat}}) *$$

$$\mathop{\ast}_{i=0}^{|L|} \left(\begin{array}{l} \texttt{IsNext}(lvl, L_{\texttt{cat}}[i], L_{\texttt{cat}}[i+1]) * \\ \texttt{HasLock}(lvl, L_{\texttt{cat}}[i], \texttt{InSubLock}) \end{array} \right) *$$

$$\mathop{\ast}_{n \in S} \left(\left[\circ \{n\}\right]^{\gamma_A} * \left[\{n.\texttt{key}\}\right]^{\gamma_T} \right)$$

Lock Resources

$$\texttt{InBotLock}(n, 0) \triangleq \exists\, s, succ.\ n.\texttt{next}[0] \hookrightarrow_{1/2} s * s \hookrightarrow_{\square} succ * (succ = \texttt{tail} \lor \exists\, v.\ succ.\texttt{val} \hookrightarrow_{1/2} v)$$

$$\texttt{InSubLock}(n, lvl) \triangleq \exists\, s.\ n.\texttt{next}[lvl] \hookrightarrow_{1/2} s$$

where $\texttt{levelN}(i)$ maps level i to its invariant namespace
$\texttt{KeyRange} \triangleq \{k : \mathbb{Z} \mid \texttt{MIN} < k < \texttt{MAX}\}$
$\texttt{tail} \triangleq (\texttt{MAX}, \texttt{NULL}, \texttt{NULL}, \texttt{NULL})$
$L_{\texttt{cat}} \triangleq [head] +\!\!+\ L +\!\!+\ [\texttt{tail}]$
$\texttt{IsNext}(lvl, pred, succ) \triangleq \exists\, s.\ pred.\texttt{next}[lvl] \hookrightarrow_{1/2} s * s \hookrightarrow_{\square} succ$
$\texttt{HasLock}(lvl, node, R) \triangleq \exists\, \gamma, l.\ node.\texttt{lock}[lvl] \hookrightarrow_{\square} l * \texttt{IsLock}(\gamma, l, R(node, lvl))$
`IsLock` is the predicate for the lock invariant

Figure 5.2: Definition of the representation predicate and invariants

5.3 Invariant for the Bottom List

Since the bottom list represents the full view of the abstract map, its shared state should reflect an equivalence between all entries of the abstract map and the existing physical nodes in memory. However, the `IsSkipList` predicate only expresses a partial view of the map, so we need relate *private* partial views to the *shared* full view. This can be accomplished with the authoritative RA presented in Chapter 4.

5.3.1 Partial Views

The idea is to maintain inside the invariant an authoritative resource as the full view of the map, with each thread holding their own fragment resource as a partial view. By opening the invariant, a thread obtains temporary ownership of the authoritative map and can perform a frame-preserving update to it and to the fragment it possesses, reflecting the intended changes to the map. In other words, a given thread's fragment resource can be seen as the combination of all its contributions to the map, meaning that combining all existing fragments should yield a resource containing the whole map. For this reason, the `IsSkipList` predicate asserts ownership of a fragment resource to reflect the partial view.

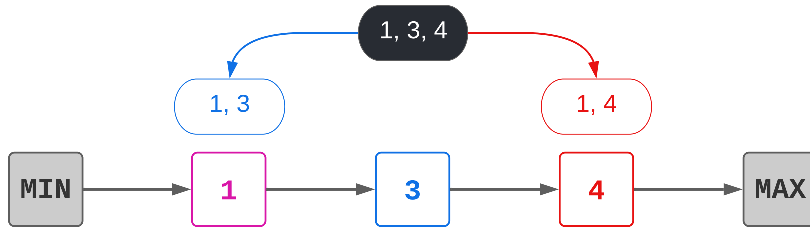


Figure 5.3: Authoritative ghost state for a linked list

Figure 5.3 provides a graphical example of how this authoritative ghost state can match the physical state of the list. The authoritative resource is represented as a *black* round box containing all keys of the linked list. The *white* round boxes represent fragment resources containing only a subset of those same keys. The underlying map RA allows for composition of maps with the same keys, which is why both fragments intersect in key 1. Furthermore, the red-bordered fragment containing keys 1 and 4 demonstrates that the keys in a fragment are not required to be directly linked in the list, *i.e.*, fragments do not represent a contiguous portion of the linked list.

However, the rule for fragment composition can lead to problems as the one shown in Figure 5.4. If we were to consider an incorrect implementation of the `put` operation, where a random element is also successfully inserted, then we could separate that additional element from the partial view. The proof could then be completed by discarding the fragment with the random element, even though we know the operation is incorrectly implemented. This issue can be solved by adding fractions to the fragments, updating the fragment composition rule to:

$$\circ_{q_1+q_2}(f_1 \cdot f_2) = \circ_{q_1} f_1 \cdot \circ_{q_2} f_2$$

This change to the rule ensures that fragments can only be split by reducing their corresponding fractions. By enforcing the postcondition of SkipPut to preserve the owned fraction, the random element can no longer be discarded, since that would yield a smaller fraction than required. For instance, in the figure example we cannot discard the fragment with key 4, since the partial view would not maintain its fraction of 1 due to the restriction on any fraction q being greater than 0. We can thus prove that the `put` operation only performs the intended update, with the thread's fragment resource reflecting its result.

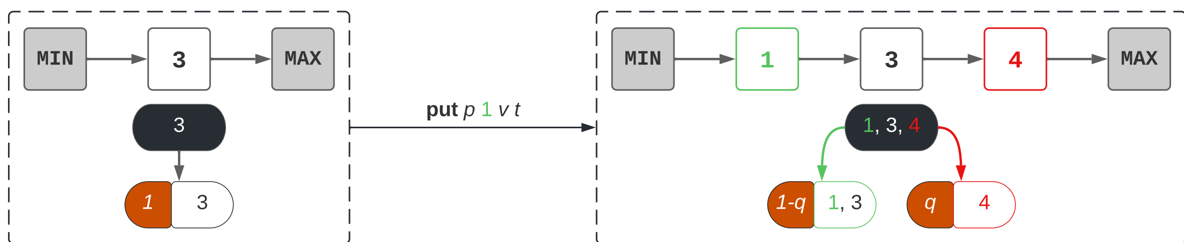


Figure 5.4: Example of an incorrect implementation of `put`

5.3.2 Set Membership

Although fractions are necessary to express partial views accurately, the authoritative RA by itself is more adequate for dealing with set membership assertions. Ownership of a fragment containing a singleton set entails that the singleton's element belongs to the authoritative set, without requiring any knowledge about the rest of the set or what fraction of the set we currently hold. This property of the authoritative RA is useful for verifying concurrent traversals when we do not have full ownership of the data structure.

Traversals are a recursive procedure where we loop over some (or all) nodes of the set until the desired node is found. For all node visits, the invariant is that the current node is one that belongs to the set. Thus, a set membership assertion for the visited node is necessary as a precondition to prove the correctness of a traversal that passes through that node. In other words, the ghost state for the bottom list should also contain an authoritative set of nodes, matching the physical nodes of the skip list.

5.3.3 Invariant Definition

The `BotListInv` predicate in Figure 5.2 shows the invariant for the bottom list. Since the shared resources can be updated, the invariant is existentially quantified by a map M, so as to not retain a constant inside the invariant. Combining all partial views will give us the full map, so no information is lost; we only require that such a map exists as an authoritative resource in the invariant. All updates to a fragment resource must necessarily be applied to the authoritative resource, so we can assert that the full view obtained by combining all map fragments is equivalent to the authoritative resource M.

The invariant also contains an authoritative resource S for the set of nodes, enforcing that the set must contain the same keys as the abstract map M. However, due to the unordered nature of sets, we are unable to express that the physical list (including the sentinels) must always remain sorted. For this reason, the invariant is also existentially quantified by a list L containing the same nodes as S. The chain of successor pointers created by the nodes should thus reflect the order of this list.

For each contiguous pair of nodes in $L_{\texttt{cat}}$ (L concatenated with both sentinels), the `IsNext` predicate asserts that the first node of the pair should point to the second one. We require two distinct points-to assertions to relate a node with its successor: one for the node's array entry and another for the successor pointer it stores. Since each key can be present in more than one level, each node holds an array of successors, whose entries can be overwritten as new nodes are inserted into the data structure. Furthermore, it is possible for different nodes to have the same successor in different levels, meaning that each array entry should store a pointer instead of the node itself. To ensure that the successor pointer remains unchanged, we resort to the persistent points-to assertion, granting read-only access to its contents.

The value of each node should also reflect the key-value pairs from the abstract map M. However, M contains a set of possible values for each key, while a physical node can only store one actual value in memory. So, for each node we assert that a lookup for the node's key in M should return the node's timestamp, as well as a set of values containing the node's real value.

5.3.4 Lock Resources

Finally, we need to consider that both the value and the successor of every node are allowed to be updated. However, while every thread possesses unrestrained read access to these fields, write access is only granted once a thread acquires the associated lock. It is thus necessary to split the ownership of these fields between what is inside the bottom list invariant for read access and what is protected inside the lock invariant for write access. We solve this issue again by introducing fractions to points-to assertions, where write access is granted only when we hold the full fraction.

The `InBotLock` predicate, which describes the resources protected by a node's lock, contains a fraction of the points-to assertion for the node's array entry, as well as a fraction of the points-to assertion for its successor's value; the remaining fractions are kept in the bottom list invariant. The persistent assertion for the successor pointer is also stored inside the lock invariant so as to connect both assertions. The locks themselves are also stored in an array, but since they remain the same during the node's lifetime, the points-to assertion for each lock may also be made persistent.

As can be interpreted from the `BotListInv` definition, the sentinel nodes are treated as an exception among the nodes. Due to the keys being restricted to `KeyRange`, the left sentinel has no predecessors and the right sentinel has no successors. These nodes never have their values updated as well, since they do not represent actual map entries. We can see that the left sentinel's value is not protected, since it has no predecessors to invoke the `InBotLock` predicate. The right sentinel's value is also ignored by the disjunction in `InBotLock`, when invoked by its predecessors. Since the right sentinel never changes successors, there is no need to invoke `HasLock` on it.

5.4 Invariant for Sublists

We have seen how to reason about the skip list's underlying map using the invariant for the bottom list. We now turn our attention to its sublists, which serve to guide the search through the skip list towards the intended node in the bottom list. Since values are never consulted during traversals in the upper levels, no map logic is necessary for the sublist invariant. However, the information kept within the sublist invariant should pertain to the level directly beneath it, which we did not consider with the bottom list for obvious reasons. Thus, the relation between consecutive levels in the skip list is our main concern in defining the sublist invariant.

5.4.1 Sublist Relation

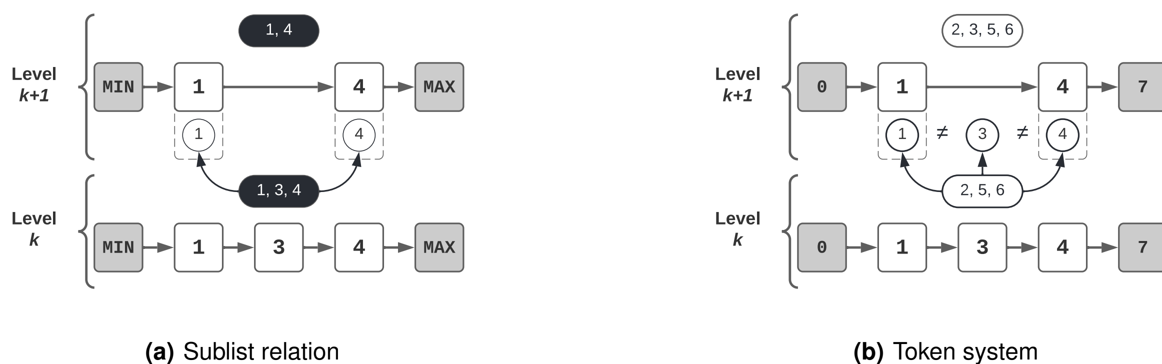
The key property behind the skip list design is that each level contains a sublist of the list contained in the level below it. In other words, when we stop the search in one level, we can continue the search in the next level from the same key without visiting its predecessors. Therefore, concluding a traversal in one level should provide the necessary context to verify the traversal in the next level.

As we discussed previously, to verify a traversal we require a set membership assertion for the node we are currently visiting, which can come in the form of a fragment resource from the authoritative set in the intended level. A node in the upper level should thus contain a fragment from the lower level, so that the succeeding traversal may be proven correct. Figure 5.5(a) captures this idea, associating the fragments of each node in level $k{+}1$ to the authoritative resource in level k.

The lower fragment can be obtained by updating the authoritative resource when inserting the node in the corresponding level. Since insertions are performed bottom-up, we obtain the fragment from the lower level before inserting in the upper level. The node can thus be inserted in the upper level, storing the fragment inside the level's invariant and returning a new fragment from the level's authoritative set.

5.4.2 Ghost Tokens

While the lower level fragments are required to express the sublist relation, verifying the insertion procedure within a given level requires additional information. When inserting in the bottom list, we check if the key from the successor node is equal to the key we want to insert; this verification step ensures that only one thread inserts the key in all intended levels. Since the absence of the key in the bottom level implies its absence in all levels, the inserting thread can insert in any upper level without checking the key of the successor in that level. To prove that the insertion is correct, however, we must still show that the new key and the successor's key are distinct, even though it is not made explicit in the code. We can take advantage of the fact that these insertions occur bottom-up to reason about this issue.



(a) Sublist relation **(b)** Token system

Figure 5.5: Ghost state relating consecutive levels

The idea is for every node to hold a token associated to its key; similarly to the aforementioned fragments, this token can only be obtained after inserting the node in the level below. Since we perform the insertion in the lower level before the upper level, we already have the token from the former when attempting to insert the node in the latter. Furthermore, we know that each node must hold its own token, so the successor from the upper level must hold a token from the lower level. By ensuring that each level contains a single token per key, we can infer that the token we hold and the successor's token must refer to different keys. Therefore, the keys are distinct and the node can be inserted in the upper level, without explicitly checking the keys in the code. The token is then stored in the invariant and a new token is extracted from the invariant to use in the next insertion.

This token system can be constructed using the *disjoint* set union RA (described in Chapter 4), with each level containing `KeyRange` as the initial set of tokens. A token for a given key is merely a singleton set containing that key and can be obtained by extracting the key from the set of available tokens. As the RA requires sets to be disjoint, the remaining keys in the set will not include the extracted key. Thus, the following tokens to be extracted will necessarily contain different keys from the tokens that have already been extracted, meaning that ownership of two tokens implies that their respective keys must differ from one another.

A graphical visualization of the token system can be seen in Figure 5.5(b), showing how it can be used to establish the desired relation between two consecutive levels. The round boxes represent the set of available tokens for the corresponding level. The keys for each set match with the set difference between `KeyRange` and the keys already in the level, where we consider $\texttt{MIN}{=}0$ and $\texttt{MAX}{=}7$. The keys contained in the upper level necessarily hold their own token extracted from the set in the lower level. All tokens are distinct, meaning that we can prove that key 3 is not in level $k{+}1$, since we hold a different token from the one held by key 4.

These tokens are also useful to prove that the levels at which the key is inserted reflect its intended height. As the key will not be inserted in level $h{+}1$, the `putH` specification in Figure 5.6 includes in its postcondition the token obtained from the insertion at level h. However, ownership of this token will depend on whether we inserted the key, as indicated by the return value $n^?$. If the optional $n^?$ comes up empty, then a node has already been created for the key. As such, some other thread holds the token for that insertion, so it cannot appear in our postcondition. Otherwise, $n^?$ contains some pointer to a node we have created for key k and so we must hold a token associated to the ghost variable γ_T^h from level h.

$$\{\, \texttt{IsSkipList}(p, M, q, \gamma) \,\} \quad \texttt{putH}\, p\; k\; v\; t\; h \quad \left\{ n^?. \begin{array}{c} \texttt{IsSkipList}(p, M \cup \{k : (\{v\}, t)\}, q, \gamma) \ast \\ \left(n^? = \texttt{None} \vee \boxed{\{k\}}^{\gamma_T^h} \right) \end{array} \right\}$$

Figure 5.6: Specification for the `putH` procedure

Given that the authoritative RA maintains the sublist relation between levels, ownership of this token results in the following observations. First, the token's key is not in the set of available keys from level h; hence, the key belongs to that level, as well as to all the levels below it. Second, the token is not associated with any node of level $h+1$; hence, the key does not belong to that level, as well as to neither of the levels above it. In other words, we guarantee that the insertion takes place at the intended height, meaning that any property of the chosen height distribution can be used to reason about the skip list.

5.4.3 Invariant Definition

The sublist invariant `SublistInv` can be seen in Figure 5.2, differing from `BotListInv` in some aspects. Map logic is no longer required for the upper levels, meaning that the authoritative map is removed and the map lookup assertions for each node are replaced with the associated fragment and token. Additionally, since values are only updated upon reaching the bottom level, acquiring upper level locks does not grant the required resources to overwrite a node's value, meaning that `InSubLock` only contains the node's array entry for the successor pointer.

The bottom list must also provide fragments and tokens to its upper level. While the fragments can be obtained from the authoritative set in the bottom list invariant, the tokens cannot be obtained from anything we discussed about this invariant. This is why, even though the tokens are not necessary for reasoning about the bottom level, the `BotListInv` predicate includes a hitherto unmentioned set of keys for the available tokens. In each level, we exclude from its set of tokens all keys that belong to its set of nodes S, ensuring that all available tokens refer to keys that have yet to be inserted in that level.

5.5 Value Updates

The `IsSkipList` predicate is defined such that a concurrent map specification can be proven for the lazy JellyFish skip list. However, while `BotListInv` ties every key to its current value, the previous values are never accounted for in the invariant. As such, the invariant alone does not guarantee the correctness of the version control mechanism employed by value updates on the data structure. We ensure that the history of values for an updated node is preserved by proving the Hoare triple shown in Figure 5.7. We first explain how the node's vertical list maintains a consistent timeline and then show how the local fragment reflects the changes to the map performed by `update`.

$$\left\{ \begin{array}{c} \boxed{\texttt{BotListInv}(head, \gamma)}^{\texttt{levelN}(0)} * \left[\circ_q M\right]^{\gamma_F} * \\ \left[\circ \{node\}\right]^{\gamma_A} * node.\texttt{val} \hookrightarrow_{1/2} val \end{array} \right\} \; \texttt{update}\; node\; v\; t \; \left\{ \begin{array}{c} \left[\circ_q (M \cup \{ node.\texttt{key} : (\{v\}, t) \})\right]^{\gamma_F} * \\ \textbf{if}\ t < val.\texttt{ts}\ \textbf{then}\ node.\texttt{val} \hookrightarrow_{1/2} val \\ \textbf{else}\ \exists\, p.\, node.\texttt{val} \hookrightarrow_{1/2} (v, t, p) * p \hookrightarrow_{\square} val \end{array} \right\}$$

Figure 5.7: Specification for the `update` procedure

5.5.1 Vertical List

To reason about a node's history, the specification for `update` describes how the procedure should affect the head of the node's vertical list. As such, the specification includes in the precondition a fraction of the points-to assertion for $node$.`val` asserting val to be its initial value, while the postcondition presents the two possible outcomes for attempting to write in this location based on the considered timestamps.

We can assert ownership of $node$.`val` as a precondition, since we only update a node after acquiring its predecessor's lock. By acquiring the lock, we obtain exclusive ownership of the resources protected by it, which include the given points-to assertion. The remaining fraction of the assertion can be obtained by opening the `BotListInv` invariant, gaining write access to the memory location. While other threads may also open the invariant to read the pointer's value, none of them can alter its contents, since they have not attained the resources protected by the lock.

At the end of the procedure, we must retain ownership of the pointer, so as to assure that we are able to release the lock by returning the necessary resources to its invariant. The value stored in this memory location may differ from val depending on the timestamps of the new and current values. If the new timestamp is less recent than the current timestamp (**`then`** branch), no update should occur and the vertical list should remain unchanged. Otherwise (**`else`** branch), the new value and timestamp should be prepended to the vertical list's head.

Being at the head of the vertical list, the new value should now point to the previous value, ensuring that the history of values is not forgotten. By making this points-to assertion persistent, we guarantee that the predecessor for a prepended value is immutable, yielding an immutable chain (or history) of values by construction. In other words, an update either returns the same immutable history of values or an immutable extension of it. Discriminating both cases using timestamps further ensures that timestamps within a given history grow monotonically, avoiding inconsistent timelines.

5.5.2 Local Fragment

As a result of the `update` procedure, the thread's partial view should be updated by combining its local map with the singleton map associating $node$.`key` to the timestamped value $(\{v\}, t)$. This update on the fragment resource can be accomplished by opening the `BotListInv` invariant, retrieving the authoritative map and performing a local update on both resources. Considering A to be the authoritative map and aliasing $node$.`key` as k and $(\{v\}, t)$ as p, we wish to perform the following frame-preserving update:

$$\bullet\, A \cdot \circ_q M \rightsquigarrow \bullet\, (A \cup \{\, k : p \,\}) \cdot \circ_q (M \cup \{\, k : p \,\})$$

To perform a frame-preserving update on authoritative and fragment resources, we first need to prove that we can perform the corresponding local update. However, the local update rules on maps only allow us to perform *insertions* instead of compositions. We thus need to rewrite the frame-preserving update

such that the update on both resources corresponds to an insertion. Since the `update` procedure is executed only when the key is known to be in the map, we can apply the rule which relates map insertion with map composition in that scenario. Knowing that the authoritative map already associates k to some timestamped value p_i, the following equality can thus be obtained:

$$A \cup \{ k : p \} = \langle k : p_i \cdot p \rangle A$$

The fragment resource, however, only holds partial knowledge of the map, meaning that the fragment may or may not hold an entry for k. Given that the empty map $\varnothing$ constitutes the unit for map composition, we can avoid performing a case analysis by observing that $M = M \cdot \varnothing$ and splitting the fragment, as follows:

$$\circ_q M = \circ_q (M \cdot \varnothing) = \circ_{q-q'} M \cdot \circ_{q'} \varnothing$$

As a result, we obtain a smaller fragment containing $\varnothing$ for some fraction $q' < q$. By applying a local update on $\varnothing$, we can then join the updated fragment to the fragment containing M, recovering the fraction q we held beforehand. As $\varnothing$ holds no key, we can be sure that any lookup will return empty, obtaining the following equality:

$$\{ k : p \} = \varnothing \cup \{ k : p \} = \langle k : p \rangle \varnothing$$

Since $\varnothing$ represents the unit, M will be combined with the intended singleton. Therefore, we only need to prove the following frame-preserving update:

$$\bullet A \cdot \circ_{q'} \varnothing \rightsquigarrow \bullet \langle k : p_i \cdot p \rangle A \cdot \circ_{q'} \langle k : p \rangle \varnothing$$

The rule for authoritative frame-preserving updates can be applied directly, taking into account the fragment with fraction q':

$$\frac{(A, \varnothing) \rightsquigarrow_{\mathsf{L}} (\langle k : p_i \cdot p \rangle A, \langle k : p \rangle \varnothing)}{\bullet A \cdot \circ_{q'} \varnothing \rightsquigarrow \bullet \langle k : p_i \cdot p \rangle A \cdot \circ_{q'} \langle k : p \rangle \varnothing}$$

To prove the required local update, we consider the scenario where the second map does not contain an entry for the key, applying the following local update rule:

$$\frac{A[k] = \texttt{Some}(p_i) \quad \varnothing[k] = \texttt{None} \quad (p_i, \texttt{botZ}) \rightsquigarrow_{\mathsf{L}} (p_i \cdot p, p)}{(A, \varnothing) \rightsquigarrow_{\mathsf{L}} (\langle k : p_i \cdot p \rangle A, \langle k : p \rangle \varnothing)}$$

We know that A associates k to p_i and that k is not in $\varnothing$, so all that is left is to prove the local update on $(p_i, \texttt{botZ})$. We simply apply the general rule for local updates by choosing p as our frame. The proof is concluded as the following equality holds:

$$(p_i \cdot p, \texttt{botZ} \cdot p) = (p_i \cdot p, p)$$

In essence, an update to a local fragment must reflect an update to the authoritative map. Furthermore, we show that the update on the authoritative map corresponds to inserting a new value for k, determined by the `argmax` composition between p_i and p. The semantics for value updates are thus captured by the `argmax` RA, constituting a perfect abstraction for the `update` procedure.

5.6 Coq Formalization

The definitions in Figure 5.2 are not an exact transcription from the Coq formalization. Although our aim thus far has been to highlight the differences between both invariant definitions, in Coq we intend to reuse as much proof effort as possible. In Figure 5.8 we show a definition for the representation predicate and its invariants closer to the formalization we have mechanized in Coq. This alternative definition simplifies how the actual proofs are carried out and allows us to generalize proofs from the sublists to the bottom list whenever map logic is not required.

Representation Predicate

$$\texttt{IsSkipList}(p, M, q, \gamma) \triangleq \exists\, head.\; p \hookrightarrow_{\square} head * head.\texttt{key} = \texttt{MIN} *$$

$$\boxed{\circ_q M}^{\gamma_F^{\texttt{HMAX}}} * \boxed{\texttt{LazyListInv}(0, head, \gamma^{\texttt{HMAX}}, \texttt{None})}^{\texttt{levelN}(0)} * \mathop{\huge *}_{i=1}^{\texttt{HMAX}} \boxed{\texttt{LazyListInv}(i, head, \gamma^{\texttt{HMAX}-i}, \texttt{Some}(\gamma^{\texttt{HMAX}-i+1}))}^{\texttt{levelN}(i)}$$

Invariant

$$\texttt{LazyListInv}(lvl, head, \Gamma, \gamma^?) \triangleq \exists\, M, S, L.$$

$$\boxed{\bullet S}^{\Gamma_A} * \boxed{\texttt{KeyRange} \setminus S.\texttt{keys}}^{\Gamma_T} * S \equiv_P L * \texttt{Sorted}(L_{\texttt{cat}}) * \texttt{HasMap}(M, S, \Gamma, \gamma^?) *$$

$$\mathop{\huge *}_{i=0}^{|L|} \Big(\texttt{IsNext}(lvl, L_{\texttt{cat}}[i], L_{\texttt{cat}}[i+1]) * \texttt{HasLock}(lvl, node, \gamma^?) * \big(L_{\texttt{cat}}[i] = head \vee \texttt{InNode}(L_{\texttt{cat}}[i], M, \gamma^?)\big)\Big)$$

Authoritative Map

$$\texttt{HasMap}(M, S, \Gamma, \gamma^?) \triangleq \begin{array}{l} \textbf{match}\; \gamma^? \;\textbf{with} \\ \mid \texttt{Some}\ _ \;\Rightarrow\; \texttt{True} \\ \mid \texttt{None} \;\Rightarrow\; \boxed{\bullet M}^{\Gamma_F} * M.\texttt{keys} = S.\texttt{keys} \end{array}$$

Lock Resources

$$\texttt{InLock}(node, lvl, \gamma^?) \triangleq \exists\, s.\; node.\texttt{next}[lvl] \hookrightarrow_{1/2} s * \begin{array}{l} \textbf{match}\; \gamma^? \;\textbf{with} \\ \mid \texttt{Some}\ _ \;\Rightarrow\; \texttt{True} \\ \mid \texttt{None} \;\Rightarrow\; \exists\, succ.\; s \hookrightarrow_{\square} succ * \\ \qquad (succ = \texttt{tail} \vee \exists\, v.\; succ.\texttt{val} \hookrightarrow_{1/2} v) \end{array}$$

Node Resources

$$\texttt{InNode}(node, M, \gamma^?) \triangleq \begin{array}{l} \textbf{match}\; \gamma^? \;\textbf{with} \\ \mid \texttt{Some}\ \gamma \;\Rightarrow\; \boxed{\circ \{node\}}^{\gamma_A} * \boxed{\{node.\texttt{key}\}}^{\gamma_T} \\ \mid \texttt{None} \;\Rightarrow\; \exists\, v, vs.\; \begin{array}{c} node.\texttt{val} \hookrightarrow_{1/2} v * v.\texttt{val} \in vs * \\ M[node.\texttt{key}] = \texttt{Some}(vs, v.\texttt{ts}) \end{array} \end{array}$$

where $\texttt{levelN}(i)$ maps level i to its invariant namespace
$\texttt{KeyRange} \triangleq \{k : \mathbb{Z} \mid \texttt{MIN} < k < \texttt{MAX}\}$
$\texttt{tail} \triangleq (\texttt{MAX}, \texttt{NULL}, \texttt{NULL}, \texttt{NULL})$
$L_{\texttt{cat}} \triangleq [head] +\!\!+\ L +\!\!+\ [\texttt{tail}]$
$\texttt{IsNext}(lvl, pred, succ) \triangleq \exists\, s.\; pred.\texttt{next}[lvl] \hookrightarrow_{1/2} s * s \hookrightarrow_{\square} succ$
$\texttt{HasLock}(lvl, node, \gamma^?) \triangleq \exists\, \gamma, l.\; node.\texttt{lock}[lvl] \hookrightarrow_{\square} l * \texttt{IsLock}(\gamma, l, \texttt{InLock}(node, lvl, \gamma^?))$
$\texttt{IsLock}$ is the predicate for the lock invariant

Figure 5.8: Representation predicate and invariant from the Coq formalization

We first note that the indices for γ have changed in the definition of `IsSkipList`: the ghost names for level i are no longer associated to γ^i, but rather to $\gamma^{\texttt{HMAX}-i}$. This is a result of building the levels for the skip list in a bottom-up fashion. Since the list type in Coq is defined as a cons list, the list of ghost names resembles a stack where the ghost names for the higher levels are the last ones to be prepended to the head of the list. As the skip list is traversed top-down, applying induction principles on such a list makes it easier to reason about how recursion unfolds for the lower levels.

The second difference in the `IsSkipList` predicate is the use of a `LazyListInv` predicate for both the bottom list and the sublists. This new representation predicate alludes to the fact that every level can be seen as an individual lazy list. The $\gamma^?$ parameter refers to an optional which determines whether level lvl contains a level $lvl{-}1$ beneath it with ghost names γ.

The `LazyListInv` invariant contains the same authoritative set of nodes S, as well as the same set of tokens as both the `BotListInv` and the `SublistInv` invariants. The sorted list L which permutes S must also reflect through `IsNext` the chain of pointers created by each node's successor. In essence, all resources which refer to the underlying list structure and abstract the list as a set of keys capture the intersection between both previously defined predicates.

The $\gamma^?$ variable will determine the remaining resources for the invariant. A `None` is passed to the bottom list, indicating that `HasMap` must contain the authoritative map M. As with the other invariant definitions, an equivalence must be established between the keys of both M and the set of nodes S. On the other hand, the sublists require no map logic, so `HasMap` contains no additional resources, ignoring for now the value γ contained in $\gamma^?$.

The value γ is also ignored by `InLock`. In all levels, the resources protected by a node's lock include the array entry for its successor in the corresponding level. The bottom list, however, contains additional resources as it also protects the value of the successor node. If the successor is not the right sentinel, then `InLock` must assert fractional ownership of its vertical list, along with the persistent points-to assertion which binds the successor to the node's array entry.

The value γ contained in $\gamma^?$ is used to assert ownership of the required fragments and tokens in the sublists. Since γ refers to the ghost names of the lower level, `InNode` asserts ownership of the fragments and tokens for every $node \in S$. If no γ is available, then we are dealing with the bottom list and `InNode` ensures that every node corresponds to a key-value pair in M, holding fractional ownership of the corresponding vertical list. The disjunction is needed, as $L_{\texttt{cat}}[0]$ corresponds to the head of the list, which does not belong to S.

`IsLazyList` is always existentially quantified by a map M, even though M is ignored by the sublists. Our formalization requires this peculiarity, since resources exclusive to the bottom list are grouped together in the same separating conjunction with resources of both list types. Figure 5.9 provides an alternative definition where these resources are explicitly separated according to the value of $\gamma^?$.

Representation Predicate

$$\texttt{IsSkipList}(p, M, q, \gamma) \triangleq \exists\, head.\, p \hookrightarrow_{\square} head * head.\texttt{key} = \texttt{MIN} *$$

$$\boxed{\circ_q\, M}^{\gamma_F^{\texttt{HMAX}}} * \boxed{\texttt{LazyListInv}(0, head, \gamma^{\texttt{HMAX}}, \texttt{None})}^{\texttt{levelN}(0)} * \mathop{\Large *}_{i=1}^{\texttt{HMAX}} \boxed{\texttt{LazyListInv}(i, head, \gamma^{\texttt{HMAX}-i}, \texttt{Some}(\gamma^{\texttt{HMAX}-i+1}))}^{\texttt{levelN}(i)}$$

Invariant

$$\texttt{LazyListInv}(lvl, head, \Gamma, \gamma^?) \triangleq \exists\, S, L.$$

$$\boxed{\bullet\, S}^{\Gamma_A} * \boxed{\texttt{KeyRange} \setminus S.\texttt{keys}}^{\Gamma_T} * S \equiv_P L * \texttt{Sorted}(L_{\texttt{cat}}) *$$

$$\mathop{\Large *}_{i=0}^{|L|} \Big(\texttt{IsNext}(lvl, L_{\texttt{cat}}[i], L_{\texttt{cat}}[i+1]) * \texttt{HasLock}(lvl, node, \gamma^?)\Big) *$$

$$\begin{array}{l} \textbf{match}\ \gamma^?\ \textbf{with} \\ \mid \texttt{Some}\ \gamma \quad \Rightarrow \quad \mathop{\Large *}_{n \in S} \left(\boxed{\circ\, \{n\}}^{\gamma_A} * \boxed{\{n.\texttt{key}\}}^{\gamma_T} \right) \\ \mid \texttt{None} \quad \Rightarrow \quad \exists\, M.\, \boxed{\bullet\, M}^{\Gamma_F} * M.\texttt{keys} = S.\texttt{keys} * \mathop{\Large *}_{n \in S} \left(\exists\, v, vs.\ \begin{array}{c} n.\texttt{val} \hookrightarrow_{1/2} v * v.\texttt{val} \in vs * \\ M[n.\texttt{key}] = \texttt{Some}(vs, v.\texttt{ts}) \end{array} \right) \end{array}$$

Lock Resources

$$\texttt{InLock}(node, lvl, \gamma^?) \triangleq \exists\, s.\, node.\texttt{next}[lvl] \hookrightarrow_{1/2} s * \begin{array}{l} \textbf{match}\ \gamma^?\ \textbf{with} \\ \mid \texttt{Some}\ _ \quad \Rightarrow \quad \texttt{True} \\ \mid \texttt{None} \quad \Rightarrow \quad \exists\, succ.\, s \hookrightarrow_{\square} succ * \\ \qquad\qquad\qquad (succ = \texttt{tail} \lor \exists\, v.\, succ.\texttt{val} \hookrightarrow_{1/2} v) \end{array}$$

where $\texttt{levelN}(i)$ maps level i to its invariant namespace
$\texttt{KeyRange} \triangleq \{k : \mathbb{Z} \mid \texttt{MIN} < k < \texttt{MAX}\}$
$\texttt{tail} \triangleq (\texttt{MAX}, \texttt{NULL}, \texttt{NULL}, \texttt{NULL})$
$L_{\texttt{cat}} \triangleq [head] \mathbin{+\!\!+} L \mathbin{+\!\!+} [\texttt{tail}]$
$\texttt{IsNext}(lvl, pred, succ) \triangleq \exists\, s.\, pred.\texttt{next}[lvl] \hookrightarrow_{1/2} s * s \hookrightarrow_{\square} succ$
$\texttt{HasLock}(lvl, node, \gamma^?) \triangleq \exists\, \gamma, l.\, node.\texttt{lock}[lvl] \hookrightarrow_{\square} l * \texttt{IsLock}(\gamma, l, \texttt{InLock}(node, lvl, \gamma^?))$
$\texttt{IsLock}$ is the predicate for the lock invariant

Figure 5.9: Alternative definition for the representation predicate and invariant

While this alternative definition provides a more natural generalization of the invariant resources, having a single separating conjunction for all node resources is more useful when mechanizing the proofs. For the alternative definition, not only is some proof effort duplicated for the each separating conjunction, it is also necessary to establish a relation between them. For instance, when a new node is inserted, the separating conjunctions must agree on the new set S and list L, which may not be a trivial endeavour to prove.

6

A Simple Client

Contents

6.1 Code Overview 49

6.2 Proof Overview 50

6.3 Timestamp Assumptions 52

Data structure specifications provide an abstraction which can be used to verify programs without worrying about how the data is handled internally by the data structure. In this chapter, we use the specification from the previous chapter to verify a client program, showing how our `argmax` RA can be used to reason about concurrent updates to the lazy JellyFish skip list. We begin by explaining the code, justifying its expected return value. We then provide a detailed sketch of the steps required for proving that the program verifies the intended behaviour. Finally, we discuss the assumptions we make about the client, which enable us to verify the program using our specification.

6.1 Code Overview

We consider the client program shown in Figure 6.1. The program first invokes `new` to create a new skip list and binds the return value to a variable p. Two threads are then spawned to execute concurrent updates on keys 10 and 20 through successive calls to `put`, with each thread updating key 10 twice and key 20 once. For both threads, we consider a local counter for the timestamps, which increments every time `put` is invoked. Each counter begins at 0 and its succeeding values are hard-coded into the client code. When the program returns to a sequential execution, `get` is invoked for both updated keys to consult the resulting state of the skip list.

Comparing the updates on key 20, we can see that their respective timestamps differ (1 and 0). Given that the update from the left thread has the most recent timestamp of the two, it should be the one to reflect the most recent value for the key. As for the updates on key 10, the last update by each thread is done at timestamp 2, meaning that we cannot disambiguate which thread defines the most recent value for the key. All we can say about key 10 is that the first updates, performed at less recent timestamps (0 and 1), will be overwritten.

The program returns a pair based on these updates. For key 20, we can guarantee that the return value must correspond to the left thread's update at timestamp 1. For key 10, however, either update from timestamp 2 may reflect its most recent value, depending on how the operations were scheduled. As such, we can only ensure that `get` returns either 3 or 6 for key 10, while it is guaranteed to return 2 for key 20.

$$
\begin{array}{c}
\mathbf{let}\ p = \mathtt{new}\ \mathbf{in} \\
\begin{array}{r||l}
\mathtt{put}\ p\ 10\ 1\ 0; & \mathtt{put}\ p\ 20\ 5\ 0; \\
\mathtt{put}\ p\ 20\ 2\ 1; & \mathtt{put}\ p\ 10\ 2\ 1; \\
\mathtt{put}\ p\ 10\ 3\ 2; & \mathtt{put}\ p\ 10\ 6\ 2;
\end{array} \\
(\mathtt{get}\ p\ 10, \mathtt{get}\ p\ 20)
\end{array}
$$

Figure 6.1: Client program with concurrent writes

6.2 Proof Overview

In Figure 6.2 we annotate the client program with the assertions derived from the rules and Hoare triples used to prove its correctness. Besides the rules from the map specification, other rules are applied to simplify the resulting assertions, making the proof easier to read and understand. Each simplification step is thus denoted by a vertical downward arrow, tagged by the corresponding rule. Violet assertions are associated to sequential parts of the program, while blue and red contents refer to thread-specific contributions as a result of a concurrent execution.

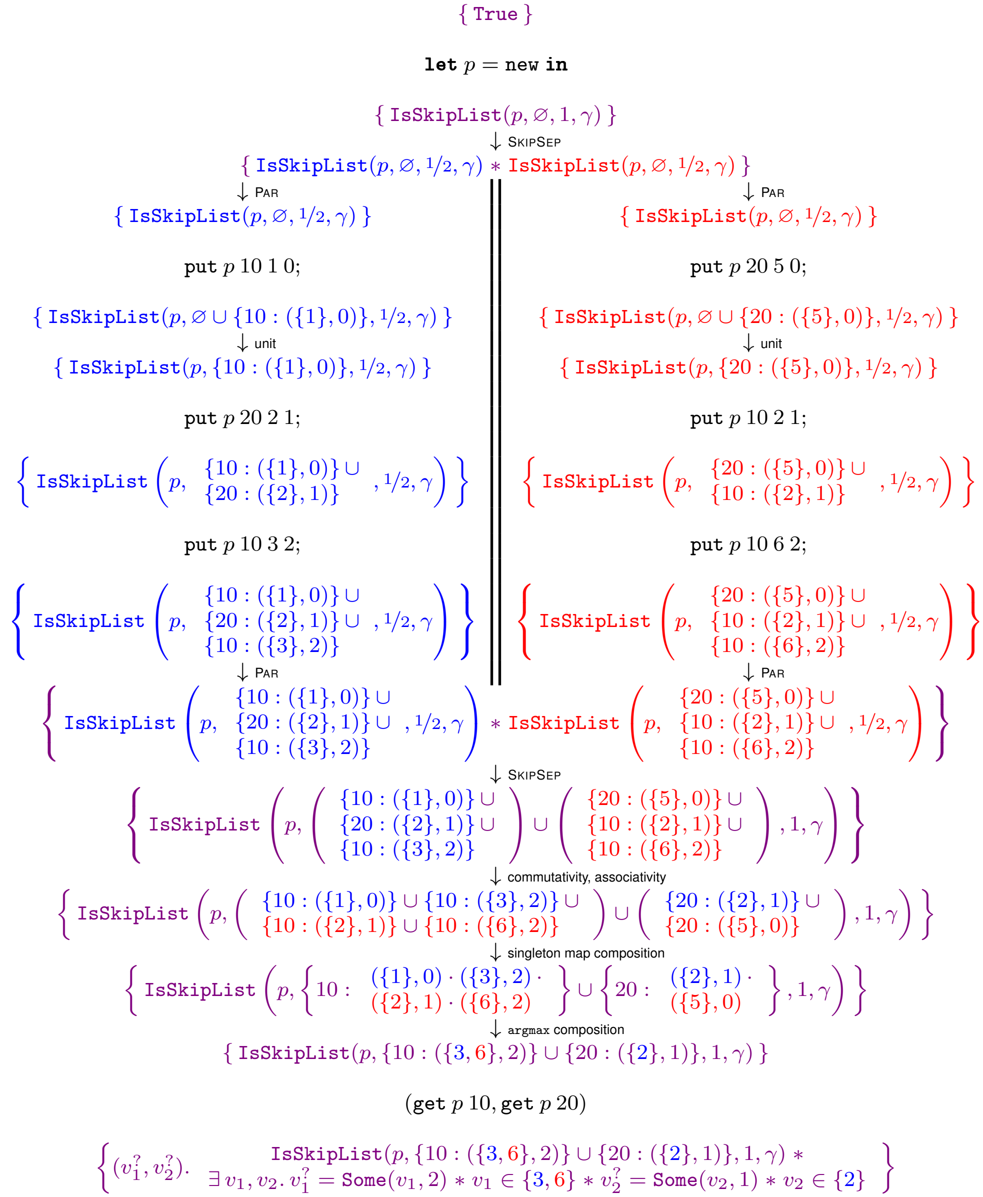


Figure 6.2: Verified client using the map specification

No initial resources are required at the start of the program, so we can apply SkipNew to obtain ownership of the map after executing `new`. The resulting `IsSkipList` assertion is obtained expressing full knowledge of an empty map stored in p and associated to some γ. The concrete value of γ is not required, since it only provides a context on which both threads should agree on. This agreement is necessary to apply SkipSep, as both `IsSkipList` assertions must refer to the same γ.

To perform concurrent operations on the data structure, we need to share knowledge of its current state between both threads. As the empty map can be decomposed as $\varnothing = \varnothing \cup \varnothing$ and the full fraction can be divided into two halves, we apply SkipSep to obtain two separate `IsSkipList` assertions, each reflecting a partial view of the underlying map. These views can then be split using the concurrency rule from separation logic, providing a partial view for each thread. Local reasoning can now be applied on the operations performed by one of the threads, without concerning ourselves with the operations performed by the other thread.

For each thread to have its updates reflected on its local view, we repeatedly apply SkipPut on each `put` call. By applying this rule, each thread can update its partial knowledge of the map, regardless of the updates performed by the other thread. The simplicity of this part of the proof showcases how the map RA perfectly abstracts the `put` operation, with the `argmax` RA perfectly abstracting the value updates performed by the `update` procedure.

Only when we return to a sequential execution do we reason about how these concurrent changes affect each other. Having both threads finished their respective updates, we recover the full view of the map by once again applying SkipSep to combine the views of both threads. At this point, we know all updates that have been performed on the map, so we can simplify the resulting composition to obtain our current knowledge of the map's state.

First, we make use of the commutative and associative properties of RAs to separate the singleton maps for key 10 from the singletons for key 20. We then compose both groups of singletons using the singleton map composition rule from Chapter 4. The resulting composition yields two singletons where the key is associated to the composition of all values from its respective updates. Finally, these values are composed using the `argmax` operator, leaving key 10 associated to both values from timestamp 2 and key 20 associated to the value from the left thread's update at timestamp 1. The remaining updates are discarded by the `argmax` operator, as they are associated to less recent timestamps.

The program terminates by executing `get` on both keys and returning a pair with the obtained results. Applying SkipGet on both `get` calls, we obtain the right-hand side of the disjunction in the postcondition, since keys 10 and 20 are clearly in the map. Thus, we are able to prove that `get` can return either value from timestamp 2 for key 10, while it is guaranteed to return the value from the left thread's update at timestamp 1 for key 20. The returned pair is therefore in accordance with what we expected from the program's behaviour.

6.3 Timestamp Assumptions

In the previous example, we assumed that a thread would always increment a local counter after calling `put`. Unfortunately, removing that assumption can lead to an over-approximate estimate of our current knowledge of the map. For instance, if a thread updates a given key twice with the same timestamp, then we can be sure that the second update will overwrite the first one. The `argmax` operator, however, will retain both values, since it combines local updates as it would with updates from different threads.

The set of possible values for a key can therefore contain additional values besides the ones which we cannot disambiguate. This behaviour is not strictly incorrect, since it maintains all possible values within the set, but it provides less precise information about the map's state. On the other hand, if we can assume that a thread will never update a key with the same timestamp more than once, then we can guarantee that such precision is not lost with the `argmax` operator.

Additionally, our client example presents hard-coded timestamps, which is unlikely to happen in real-world programs. Timestamps are usually dynamically assigned through a shared counter or some other complex mechanism, which might incur some additional effort for proving the correctness of the corresponding client. We argue, however, that the `argmax` RA should be able to capture the required semantics for such timestamps, as long as the aforementioned constraint is satisfied.

7

Discussion and Related Work

Contents

7.1 Concurrent Data Structures in Iris . . . 55
7.2 Mechanized Verification of Skip Lists . . . 57
7.3 Specifications for Concurrent Maps . . . 57

We now discuss our results by comparison with related work. First, we present work in the context of formal verification of concurrent data structures in Iris. Namely, we introduce the notion of logical atomicity and present the work on which we base most of our proof effort. We also present known work of mechanized efforts to verify skip list algorithms. Finally, we discuss alternative approaches on specifying concurrent operations on maps.

7.1 Concurrent Data Structures in Iris

Iris has been used to reason about concurrent data structures, verifying (a) a contextual refinement of other simpler concurrent implementations [29,30], (b) correctness under a weak memory model [23] and (c) template algorithms for search structures [22]. Although the latter closely relates to the contributions of our work (as skip lists can be seen as search structures), their template algorithms do not seem be directly applicable to skip lists. Due to the skip list being a data structure composed of multiple linked lists, each level is its own search structure, so their work would need to be generalized if it were to be applied to any skip list implementation. Furthermore, the obtained specifications are not well suited for verifying generic client programs. Rather, they focus on proving a strong correctness property of concurrent operations: logical atomicity [6,10,15,18].

7.1.1 Logical Atomicity

A concurrent operation is said to be logically atomic if a single atomic step is responsible for changing the state of the program so as to satisfy its specification. In other words, the effective changes to the state occur at a single point of execution, which is referred to as the operation's linearization point. Logical atomicity therefore allows us to reason about non-atomic operations *as if* they were atomic. For instance, this property allows us to open invariants around logically atomic operations by adapting the INV rule to:

LA-INV
$$\frac{\langle\, \triangleright I * P \,\rangle \quad e \quad \langle\, v.\, \triangleright I * Q(v) \,\rangle_{\mathcal{E}\setminus\mathcal{N}} \qquad \mathcal{N} \subseteq \mathcal{E}}{\boxed{I}^{\mathcal{N}} \vdash \langle\, P \,\rangle \quad e \quad \langle\, v.\, Q(v) \,\rangle_{\mathcal{E}}}$$

Angled brackets denote logically atomic triples and program e is no longer required to be atomic. Iris already provides ways to reason about logical atomicity in this manner making invariants easier to handle for more complex programs. However, the specification obtained by Krishna *et. al.* [22] assumes full knowledge of the state of the structure, making it unclear how such a specification could be used to share ownership between threads and to combine the results of concurrent operations. On the other hand, although our specification does not guarantee logical atomicity, it is expressive enough to allow the verification of highly concurrent client programs, as we've shown in Chapter 6.

7.1.2 Concurrent Skip Lists

The only work in Iris which we are aware of reasoning about concurrent skip lists is that of Tassarotti and Harper [28]. They extended Iris to support probabilistic reasoning, proving correctness and temporal properties of a two-level concurrent append-only skip list, which is a probabilistic data structure. While their work focused more on probabilistic properties, we avoid reasoning about such aspects, proving correctness of the data structure independently of the height distribution.

Our mechanization is deeply influenced by theirs, generalizing their arguments to an arbitrary number of levels. While their proofs constitute the base of our reasoning on traversals and updates at the list level, we complement their work by using the authoritative RA to reason about the sublist relation for traversals and by introducing the token system to reason about insertions in consecutive levels.

Additionally, we simplify the ghost state used to reason about operations at the list level. Since their skip list implements a set rather than a map, their invariant holds an authoritative set of keys for the partial views and an authoritative set of physical nodes with a slightly different structure. Although these differences arise naturally from small differences between both implementations, these ghost resources are enough to verify the skip list similarly to how we did. However, two additional ghost variables are considered by Tassarotti and Harper.

The disjoint set union RA is used to reason about insertions at the list level. After obtaining the soon-to-be predecessor and successor nodes, it must be proven that the new key can only exist between these nodes. We prove this by reasoning about the sorted state of the list, but Tassarotti and Harper proceed differently: the tokens associated to all keys between the two nodes are included in the resources protected by the predecessor's lock. As such, one can only obtain the token for the new key by extracting it from this interval, yielding two new disjoint intervals: one to be protected by the predecessor and another by the new node. The token can then be added to a set of tokens in the invariant, reflecting the keys contained in the list.

The map RA is also used to associate each key to elements of an agreement RA over nodes, whose value composition enforces nodes to be equal. Although we know that each node has a unique key, nodes are simply tuples from Coq's perspective. As such, two tuples with the same key might not be equal if the remaining fields differ, which is why they resort to a ghost map assuring a one-to-one correspondence between keys and nodes. Their formalization requires this correspondence, because their partial views are represented as sets of nodes containing the keys which parameterize the representation predicate. Therefore, the map is necessary to prove that two threads with some key in common must agree on the node for that key. This issue is avoided by representing partial views as sets of keys and stating in the invariant an equivalence between authoritative resources as we do.

7.2 Mechanized Verification of Skip Lists

Considering other verification frameworks, we highlight: (a) the TSL theory for reasoning about skip lists with an arbitrary number of levels [27], (b) the verification in Agda of correctness properties of authenticated append-only skip lists [3] and (c) the skip list entry in the Archive of Formal Proofs (AFP) [11].

The AFP is a repository of proof libraries mechanized in Isabelle. One of these libraries contains proofs of probabilistic properties of skip lists, proving that the expected height for an unbounded skip list scales logarithmically with the size of the structure. This result is then used to prove the same asymptotic complexity for the expected traversal length. Considering a skip list of n keys and a geometric progression over p for the height distribution, the harmonic number definition (which is known to be $\Theta(\texttt{log}(n))$) can be used to obtain the following bounds for the expected height H and the expected traversal length L:

$$-\frac{\texttt{harm}(n)}{\texttt{log}(p)} - 1 \leq H\,n \leq -\frac{\texttt{harm}(n)}{\texttt{log}(p)} \qquad\qquad \frac{1}{p} \cdot H\,n = L\,n$$

These works, however, deal with sequential skip lists, so they tackle inherently different concerns from the ones we have discussed in this work. For concurrent skip lists, Abdulla *et al.* [1] present the only other work we are aware of providing mechanized proofs, automating their reasoning in a self-developed tool. Their verification tool was implemented in OCaml and, as far as we could tell, does not build on top of any known state-of-the-art theorem prover.

7.3 Specifications for Concurrent Maps

Our proposed specification for concurrent operations on maps holds some limitations. Since SkipGet is undefined for partial ownership of the map, certain client applications (*e.g.*, producer-consumer clients) are unable to use our specification. We will now look at two works, which propose different approaches from the one described in this work, and discuss how our work could be improved based on their results.

7.3.1 Key-value Specifications

As a thread performs its updates, our specification accumulates all its contributions in its partial view. However, the previous updates are unnecessary to express the results of the current update, so an alternative approach would be to define a specification based on each key-value pairing of the map. In the work by da Rocha Pinto *et. al.* [5], they define such a specification using two predicates:

- $\texttt{in}(p, k, v)$: the map stored in p associates key k to value v;
- $\texttt{out}(p, k)$: the map stored in p does not contain a mapping for key k.

The postcondition for put should thus contain resources described by the in predicate, while removals should return resources described by out. To enable sharing, these predicates are also annotated with a fraction to indicate if the thread holds exclusive access to the map entry, as well as a protocol tag to indicate how threads should combine their results. For example, in $\mathtt{out}_{\mathtt{ins}}(p,k)_{1/2}$ the fraction $1/2$ informs us that we do not hold exclusive ownership of this entry, while the ins tag assures us that threads are only allowed to perform insertions in this entry. Threads can then combine their resources to obtain the full fraction, applying the composition rules associated to the given protocol tag.

The main downside to this approach is that the use cases for the obtained specification are restricted to the considered hard-coded protocols. For instance, the protocols in the original paper do not describe what should happen when threads update the same key with different values. As such, a program similar to our client example cannot be verified, requiring a new protocol to be defined and the associated proofs to be done from scratch. Xiong *et. al.* [31] provide an alternative key-value specification, which focuses on proving correctness properties of the map implementation, allowing different protocols to be defined on top of this specification.

This alternative key-value specification is built from a logically atomic specification considering full knowledge of the map, assuring a strong correctness property for the data structure, similarly to the work of Krishna *et. al.* [22]. Different protocols can then be established on top of the obtained key-value specification by reasoning about an appropriate PCM. To verify a client program it thus suffices to think of an appropriate algebra (be it a PCM or a RA) to abstract the required protocol, obtaining for free correctness properties from the logically atomic specification.

Their formalization is done in the logic of TaDA [6], a predecessor to Iris, which contains ingredients such as shared regions and guard algebras, akin to Iris' invariants and RAs, respectively. Their efforts were not mechanized in any known framework, but their results could further extend the contributions of our work. Although we have not mechanized such extensions, we now discuss how their approach could be formalized in Iris. As with JellyFish, we consider maps which associate each key to a timestamped value. A logically atomic specification for put should thus resemble the following:

$$\langle\, \mathtt{Map}(p, M, \gamma)\,\rangle \quad \mathtt{put}\; p\; k\; v\; t \quad \left\langle \begin{array}{lll} \textbf{match}\ M[k]\ \textbf{with} & & \\ \mid \mathtt{None} & \Rightarrow & \mathtt{Map}(p, \langle k : (v,t)\rangle M, \gamma) \\ \mid \mathtt{Some}(v_i, t_i) & \Rightarrow & \textbf{if}\ t < t_i\ \textbf{then}\ \mathtt{Map}(p, M, \gamma) \\ & & \textbf{else}\ \mathtt{Map}(p, \langle k : (v,t)\rangle M, \gamma) \end{array} \right\rangle$$

The Map predicate asserts ownership of the full map, referring to M as the full view. Since we aim to separate the correctness proof from client reasoning, no mention is made to RA composition in the postcondition. Rather, it is made explicit how the state should change as a result of the operation: the new value and timestamp will be inserted in the map unless the timestamp of the update is less recent than the current timestamp. Logical atomicity ensures that this change occurs after a single atomic step of the non-atomic put operation, meaning that M actually refers to the state of the map as that step

is taken, instead of the state when `put` is called. As a result, we can reason about thread composition using only that point of execution where the operation effectively takes place.

`Map` can be constructed from `IsSkipList`, forcing the fraction to equal 1 so as to reflect the full view of the map. However, an authoritative RA builds on top of another RA, which in this case corresponds to the map RA. Since we want to avoid reasoning about value composition at this stage, we instead require a RA that encapsulates any given type to define our view of the map. The agreement RA is one such RA and can be defined by the following rules:

$$\mathsf{ag}_{q_1}(v) \cdot \mathsf{ag}_{q_2}(v) = \mathsf{ag}_{q_1+q_2}(v) \qquad\qquad \mathsf{ag}_1(v_i) \rightsquigarrow \mathsf{ag}_1(v)$$

Composition is valid only when both elements agree on the same value v and fractions are included to know when we possess exclusive ownership of the value. Knowing that no other thread holds partial ownership of the value to be agreed on, we are able to change it to any other value of the same type. We can thus encapsulate the abstract map inside an agreement RA and split it into two fractions: one inside the invariant, replacing the authoritative map, and the other outside the invariant, replacing the local fragment. Information about the state of the entire map is kept through the outer fraction, which can be updated by opening the invariant and combining both fractions to obtain the full fraction.

Proving `put` to be logically atomic should pose a challenge as to how the resources are managed throughout the proof. However, the reasoning applied to those resources should remain the same, as we only update the partial view when we open the invariant to insert/update the key's node. Since the invariant can only remain open for an atomic step, the corresponding store operation constitutes the linearization point for `put`, making it logically atomic. Once the proof is completed for the full map specification, we can use it to prove the following key-value specification:

$$\left\langle \mathtt{Key}(p,k,v_i^?,\gamma) \right\rangle \quad \mathtt{put}\; p\; k\; v\; t \quad \left\langle \begin{array}{lll} \textbf{match } v_i^? \textbf{ with} & & \\ \mid \mathtt{None} & \Rightarrow & \mathtt{Key}(p,k,\mathtt{Some}(v,t),\gamma) \\ \mid \mathtt{Some}(v_i,t_i) & \Rightarrow & \textbf{if } t < t_i \textbf{ then } \mathtt{Key}(p,k,v_i^?,\gamma) \\ & & \textbf{else } \mathtt{Key}(p,k,\mathtt{Some}(v,t),\gamma) \end{array} \right\rangle$$

The $\mathtt{Key}(p,k,v^?,\gamma)$ assertion expresses the mapping of key k to an optional $v^?$. Allowing $v^?$ to be `None` means that we can explicitly assert that k does not belong to the map. Updates on $v^?$ should respect the same constraints as in the full map specification. The `Key` predicate can be constructed from `Map` as follows:

$$\mathtt{Key}(p,k,v^?,\gamma) \triangleq \left[\mathsf{ag}_{1/2}\left(v^?\right)\right]^{\gamma^k} * \boxed{\exists M,\Gamma.\, \mathtt{Map}(p,M,\Gamma) * \mathop{\huge *}_{z\,\in\,\mathtt{KeyRange}} \left[\mathsf{ag}_{1/2}\left(M[z]\right)\right]^{\gamma^z}}^{\mathtt{kvN}}$$

Since each `Key` resource will refer to a different key, the non-duplicable `Map` resource is kept inside an invariant for some map M associated to some ghost names Γ. We then need to establish an equivalence between each value in the map and the value from its respective `Key` assertion. Once again, the agreement RA fits our needs, keeping for each key a $1/2$ fraction inside the invariant and another

outside, which also ensures that `Key` resources are exclusive per key. As there can only be a single invariant containing the `Map` resource, it must contain the halves of all keys, where `KeyRange` may refer to any set of valid keys for the map. For JellyFish we considered all integers between `MIN` and `MAX`; Xiong *et. al.* [31] consider all non-zero integers.

The key-value specification can be proven by opening the invariant around `put`. As a result, we obtain the `Map` resource and can then apply the full map specification as if `put` were an atomic operation. Both agreement fractions are combined, so that the corresponding value can be updated according to the postcondition of the full map specification. After that logically atomic operation, the invariant is closed and the postcondition of the key-value specification is satisfied.

To complete the key-value specification, we still need some predicate to describe the state of the whole map, making it easier to handle the `Key` resources. For this purpose, we further define the `Collect` predicate:

$$\texttt{Collect}(p, S, \gamma) \triangleq \mathop{\Large\ast}_{k \,\in\, \texttt{KeyRange} \setminus S} \texttt{Key}(p, k, \texttt{None}, \gamma)$$

`Collect` only contains keys that do not belong to the map. The S parameter keeps track of all keys that we extract from `Collect` in order to perform operations on those keys. New `Key` resources can be extracted from or returned to `Collect` by applying the following rule:

$$\texttt{Collect}(p, S, \gamma) \dashv\vdash \texttt{Collect}(p, S \uplus \{k\}, \gamma) * \texttt{Key}(p, k, \texttt{None}, \gamma)$$

Both specifications could be adapted such that each key is mapped to its vertical list, instead of just the last value. Any mapping to (v_i, t_i) should be replaced by $[(v_i, t_i)] +\!\!+ H$, for some history of values H, and should be updated to $[(v, t)] +\!\!+ [(v_i, t_i)] +\!\!+ H$ if $t_i \leq t$. In case no such mapping exists, then the value should be initialized with the singleton list $[(v, t)]$.

7.3.2 Client Reasoning

A logically atomic key-value specification satisfies a strong correctness criterion for concurrent operations on shared data. Unfortunately, it does not provide a natural way to reason about client programs and how concurrent operations should be composed. To reason about such matters, one needs only to think of a RA to express the protocol followed by threads on conflicting operations. We will now show how the `argmax` RA can complement the key-value specification to verify the client program from Chapter 6.

We have seen previously that the pairs from the `argmax` RA differ from the timestamped values stored in the map. The `argmax` RA keeps track of a set rather than the actual value, due to the uncertainty derived from concurrent updates with the same timestamp. As such, we can assert an equivalence between some map value $v^?$ and some `argmax` value $v_A^?$ through the following definition:

$$\texttt{SomeEquiv}(v^?, v_A^?) \triangleq \begin{array}{l}(v^? = \texttt{None} * v_A^? = \texttt{None}) \vee \\ \exists\, v, S, t.\, v^? = \texttt{Some}(v, t) * v_A^? = \texttt{Some}(S, t) * v \in S\end{array}$$

If we were to consider the entire vertical list, then $v^?$ should instead equal $\texttt{Some}([(v, t)] +\!\!+\, H)$ for some H. Using this definition, we can thus define the following representation predicate for key-value pairings:

$$\texttt{FKey}(p, k, v_F^?, q, \gamma) \triangleq \left[\circ_q\, v_F^?\right]^{\gamma^k} * \boxed{\exists\, v^?, \Gamma.\, \texttt{Key}(p, k, v^?, \Gamma) * \exists\, v_A^?.\left[\bullet\, v_A^?\right]^{\gamma^k} * \texttt{SomeEquiv}(v^?, v_A^?)}^{\texttt{keyN}(k)}$$

We follow the same approach of using a fractional authoritative RA to express partial views, but this time for values. An FKey resource asserts ownership of a fragment resource, while the authoritative value is kept inside an invariant. The Key resource is also kept inside the invariant, so that it can be shared between all threads which hold some partial view of the value. The invariant can be open around put to apply the key-value specification using the Key resource, updating the authoritative and fragment resources accordingly. A predicate with the same interpretation as Collect can also be defined as:

$$\texttt{FCollect}(p, S, \gamma) \triangleq \mathop{\Large *}_{k \,\in\, \texttt{KeyRange} \setminus S} \texttt{FKey}(p, k, \texttt{None}, 1, \gamma)$$

With both predicates we can finally present the logically atomic key-value specification for maps with timestamped values shown in Figure 7.1. KEYSEP enables value sharing and composition, much like SKIPSEP does for map views, while KEYCOL allows us to obtain FKey resources. KEYNEW is a standard Hoare triple for a map constructor, which returns the resources describing an empty map. To ensure correctness of the get operation, KEYGET is defined as logically atomic, even though it does not change the state of the map. As with SKIPGET, KEYGET considers the full view, since the argmax operator does not provide a way to reason about concurrent reads. The SomeEquiv predicate is used to establish an equivalence between the returned result $v^?$ and the known argmax value $v_F^?$. Finally, KEYPUT updates the initial value $v_F^?$ by composition with $\texttt{Some}(\{v\}, t)$. Optional composition contains None as its unit, while composition of Some elements is given by $\texttt{Some}(a) \cdot \texttt{Some}(b) = \texttt{Some}(a \cdot b)$.

KEYSEP

$$\texttt{FKey}(p, k, v_1^?, q_1, \gamma) * \texttt{FKey}(p, k, v_2^?, q_2, \gamma) \dashv\vdash \texttt{FKey}(p, k, v_1^? \cdot v_2^?, q_1 + q_2, \gamma)$$

KEYCOL

$$\texttt{FCollect}(p, S, \gamma) \dashv\vdash \texttt{FCollect}(p, S \uplus \{k\}, \gamma) * \texttt{FKey}(p, k, \texttt{None}, 1, \gamma)$$

KEYNEW

$$\frac{}{\{\,\texttt{True}\,\}\quad \texttt{new}\quad \{\,p.\,\exists\,\gamma.\,\texttt{FCollect}(p, \varnothing, \gamma)\,\}}$$

KEYGET

$$\frac{k \in \texttt{KeyRange}}{\langle\,\texttt{FKey}(p, k, v_F^?, 1, \gamma)\,\rangle\quad \texttt{get}\ p\ k\quad \langle\, v^?.\,\texttt{FKey}(p, k, v_F^?, 1, \gamma) * \texttt{SomeEquiv}(v^?, v_F^?)\,\rangle}$$

KEYPUT

$$\frac{k \in \texttt{KeyRange}}{\langle\,\texttt{FKey}(p, k, v_F^?, q, \gamma)\,\rangle\quad \texttt{put}\ p\ k\ v\ t\quad \langle\,\texttt{FKey}(p, k, v_F^? \cdot \texttt{Some}(\{v\}, t), q, \gamma)\,\rangle}$$

Figure 7.1: Logically atomic key-value specification for timestamped values

In Figure 7.2, we show how this specification can be used to verify the client from Chapter 6. After creating the map and obtaining the `FCollect` resource from KeyNew, we apply KeyCol to extract the `FKey` resources for keys 10 and 20, as they are the only keys to be updated. We no longer require the `FCollect` resource, so we discard it from the environment and apply KeySep to split each `FKey` resource. These resources are then split between each thread, such that both threads hold a view of both keys.

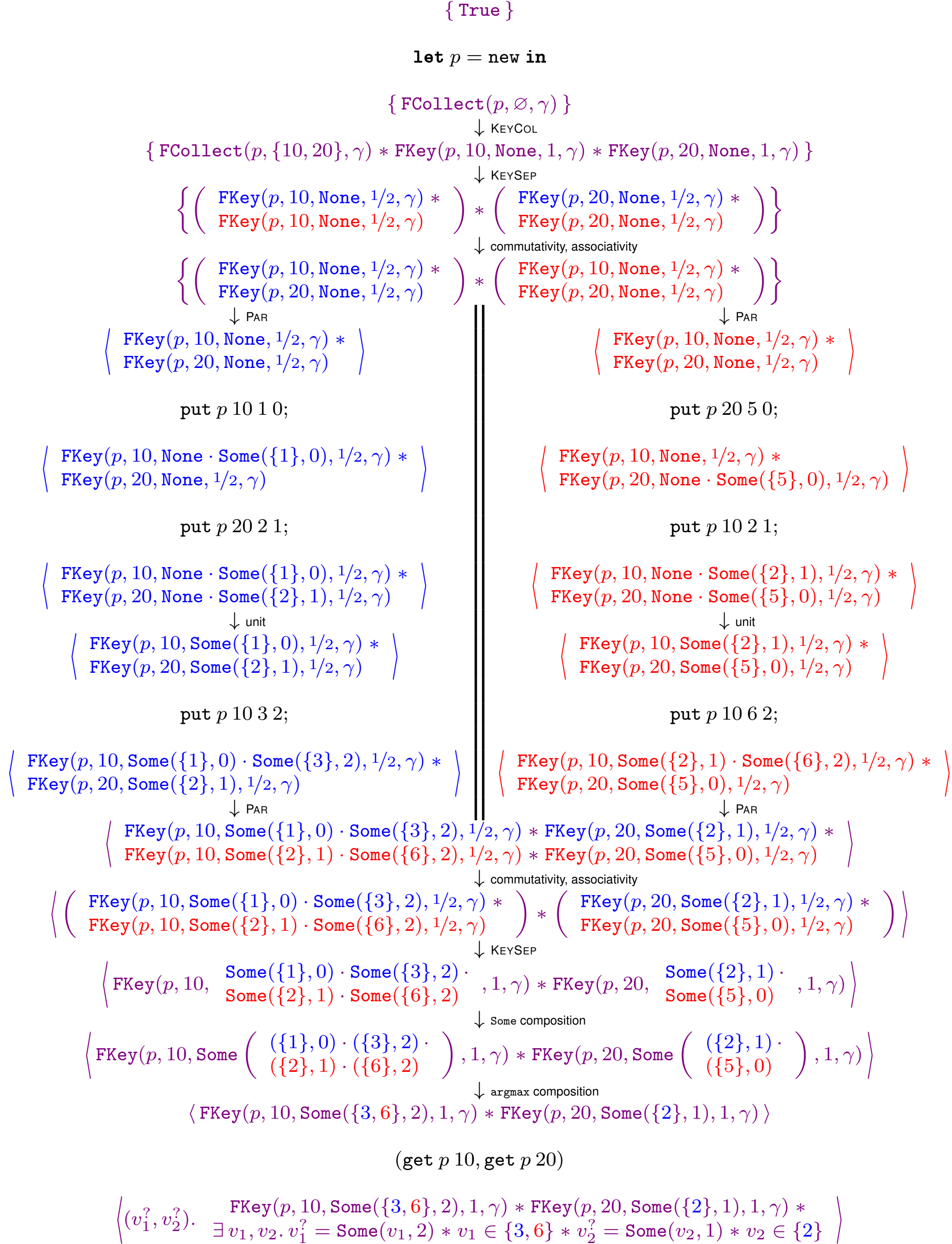


Figure 7.2: Verified client using the key-value specification

For each put call, the corresponding FKey resource is updated accordingly by application of KEYPUT. The contributions of both threads can then be combined, separating FKey resources for key 10 from FKey resources for key 20. KEYSEP is once again applied to obtain the full view of each FKey resource. The Some composition rule is applied, followed by the argmax composition, obtaining the resulting known state of each key. By application of KEYGET, we obtain the same return value as the proof from Chapter 6.

Alternatively, a specification for partial views of the map can also be built from the full map specification. We can do so by defining the following FMap predicate:

$$\mathtt{FMap}(p, M_F, q, \gamma) \triangleq \boxed{\circ_q M_F}^{\gamma} *$$

$$\boxed{\exists M, \Gamma.\, \mathtt{Map}(p, M, \Gamma) * \exists M_A.\, \boxed{\bullet M_A}^{\gamma} * \mathop{\Huge *}_{k \in \mathtt{KeyRange}} \mathtt{SomeEquiv}(M[k], M_A[k])}^{\mathtt{mapN}}$$

The Map resource is kept inside the invariant, so that it can be shared between all threads. The full map and the authoritative map must be equivalent, hence the SomeEquiv relation required for all valid keys. While the key-value specification allows us to define explicitly how values should be composed, both specifications can be used to reason about most of the same clients, as long as the value composition operator is the same. The main advantage of the key-value specification is that it enables the verification of clients where some keys may not be shared, whereas a Map resource corresponds to a fraction of the entire map, making all keys shared by default.

To support reasoning for different client programs, one only needs to change the value RA of either FKey or FMap for that purpose. No additional proof effort is required to reason about structural changes to the map, separating the correctness proof from the verification of functional properties. Through RAs, protocols can be defined for how threads interact with each other, without concerning ourselves with how the structure handles the data internally.

For instance, to redefine the argmax operator so as to cover the limitations of SkipGet, we would need to redo all the proofs from scratch, even though the changes would be minimal. On the other hand, the key-value specification could be adapted by simply modifying the RA used in FKey, preserving the logically atomic specification which handles Key resources. An alternative definition of KEYGET can thus be proven by opening the invariant and applying the get specification for Key resources, leaving the new RA to handle how the result from get should relate to the results of other threads.

8

Conclusion

Contents

8.1 Future Work 67

We presented a lock-based variant of the JellyFish skip list and showed that its implementation satisfies a concurrent map specification. Conflicts on concurrent updates are handled through timestamps, which we reason about using a novel resource algebra for the `argmax` operation. We match the abstract state of the map with the physical state of the skip list by only applying map logic to the bottom level. Our proofs are generalized for any number of levels and height distribution by reasoning about each level independently, tying consecutive levels by ghost state. We discuss how the obtained specification could be further improved by enforcing logical atomicity for the map's operations and defining a more general key-value specification. This work contributes to the understanding of complex list-based structures, providing a new approach for reasoning about concurrent maps.

8.1 Future Work

We leave as future work the adaptation of our proofs to the original JellyFish [32] and other concurrent skip list implementations [4, 7, 9, 13]. Any changes to the proofs should account for the lock-free nature of most concurrent skip lists, as well as optimizations, such as stopping the search when the key is found in the upper levels.

Our proofs could be refactored to provide a logically atomic specification for maps over timestamped domains. Such a specification could be used to build an alternative specification for reasoning about individual key-value pairings, leading to significant proof reuse. Different client specifications could be defined by constructing a suitable RA to reason about the specification for the underlying data structure.

The `argmax` RA could be extended to account for limitations of the protocol it establishes for thread interaction. Namely, when a thread updates a key more than once with the same timestamp, we can be sure that the last update will remain in the map; the `argmax` operator, however, will retain all values. Changing the RA could also allow a new `get` specification, considering partial ownership of the map.

The main issue of SkipGet and KeyGet is that they represent sequential specifications, in the sense that they reflect how the state of the structure should remain unchanged by the end of a search. While it is true that searches should be read-only procedures, representation predicates for concurrent reasoning reflect *knowledge* of the state, which should be updated depending on the search results. As such, the specification for `get` should reflect an update to the partial view, similar to:

$$\{\, \texttt{IsSkipList}(p, M, q, \gamma) \,\} \quad \texttt{get}\, p\, k \quad \{\, v.\, \texttt{IsSkipList}(p, M \cup \{k : v\}, q, \gamma) \,\}$$

These `get` updates should agree with `put` updates when combining all views. While the map RA operator is an abstraction of `put` and `argmax` an abstraction of `update`, no such abstraction is defined for `get`. Either `argmax` could be modified to cover that gap or a new operator could be defined establishing a coupling with `argmax`, much like `get` and `put` are connected by the same data structure. We leave as a research question whether RAs are a strong enough construct to express such a notion.

Bibliography

[1] ABDULLA, P. A., JONSSON, B., AND TRINH, C. Q. Fragment Abstraction for Concurrent Shape Analysis. In *Programming Languages and Systems* (Cham, 2018), A. Ahmed, Ed., vol. 10801 of *Lecture Notes in Computer Science*, Springer, pp. 442–471.

[2] BIRKEDAL, L., AND BIZJAK, A. Lecture notes on iris: Higher-order concurrent separation logic. *Aarhus University* (2020).

[3] CACCIARI MIRALDO, V., CARR, H., MOIR, M., SILVA, L., AND STEELE JR., G. L. Formal Verification of Authenticated, Append-Only Skip Lists in Agda. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New York, NY, USA, 2021), CPP 2021, Association for Computing Machinery, p. 122–136.

[4] CRAIN, T., GRAMOLI, V., AND RAYNAL, M. No Hot Spot Non-blocking Skip List. In *2013 IEEE 33rd International Conference on Distributed Computing Systems* (2013), IEEE, pp. 196–205.

[5] DA ROCHA PINTO, P., DINSDALE-YOUNG, T., DODDS, M., GARDNER, P., AND WHEELHOUSE, M. A Simple Abstraction for Complex Concurrent Indexes. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (New York, NY, USA, 2011), OOPSLA '11, Association for Computing Machinery, p. 845–864.

[6] DA ROCHA PINTO, P., DINSDALE-YOUNG, T., AND GARDNER, P. TaDA: A Logic for Time and Data Abstraction. In *ECOOP 2014 – Object-Oriented Programming* (Berlin, Heidelberg, 2014), R. Jones, Ed., vol. 8586 of *Lecture Notes in Computer Science*, Springer, pp. 207–231.

[7] DICK, I., FEKETE, A., AND GRAMOLI, V. A skip list for multicore. *Concurrency and Computation: Practice and Experience 29*, 4 (May 2017), e3876.

[8] DIETRICH, E. A beginner guide to Iris, Coq and separation logic, 2021.

[9] FRASER, K. Practical lock-freedom. Tech. Rep. UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, 2004.

[10] FRUMIN, D., KREBBERS, R., AND BIRKEDAL, L. Reloc: A mechanised relational logic for fine-grained concurrency. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (New York, NY, USA, 2018), LICS '18, Association for Computing Machinery, p. 442–451.

[11] HASLBECK, M. W., AND EBERL, M. Skip Lists. *Archive of Formal Proofs* (Jan. 2020). https://isa-afp.org/entries/Skip_Lists.html, Formal proof development.

[12] HELLER, S., HERLIHY, M., LUCHANGCO, V., MOIR, M., SCHERER, W. N., AND SHAVIT, N. A lazy concurrent list-based set algorithm. In *International Conference On Principles Of Distributed Systems* (Berlin, Heidelberg, 2005), J. H. Anderson, G. Prencipe, and R. Wattenhofer, Eds., vol. 3974 of *Lecture Notes in Computer Science*, Springer, pp. 3–16.

[13] HERLIHY, M., LEV, Y., LUCHANGCO, V., AND SHAVIT, N. A Simple Optimistic Skiplist Algorithm. In *International Colloquium on Structural Information and Communication Complexity* (Berlin, Heidelberg, 2007), G. Prencipe and S. Zaks, Eds., vol. 4474 of *Lecture Notes in Computer Science*, Springer, pp. 124–138.

[14] HERLIHY, M., SHAVIT, N., LUCHANGCO, V., AND SPEAR, M. *The art of multiprocessor programming*, 2nd ed. Newnes, 2020.

[15] JACOBS, B., AND PIESSENS, F. Expressive Modular Fine-Grained Concurrency Specification. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2011), POPL '11, Association for Computing Machinery, p. 271–282.

[16] JUNG, R., KREBBERS, R., BIRKEDAL, L., AND DREYER, D. Higher-Order Ghost State. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA, 2016), ICFP 2016, Association for Computing Machinery, p. 256–269.

[17] JUNG, R., KREBBERS, R., JOURDAN, J.-H., BIZJAK, A., BIRKEDAL, L., AND DREYER, D. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming 28* (Nov. 2018), e20.

[18] JUNG, R., SWASEY, D., SIECZKOWSKI, F., SVENDSEN, K., TURON, A., BIRKEDAL, L., AND DREYER, D. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2015), POPL '15, Association for Computing Machinery, p. 637–650.

[19] KREBBERS, R., JOURDAN, J.-H., JUNG, R., TASSAROTTI, J., KAISER, J.-O., TIMANY, A., CHARGUÉRAUD, A., AND DREYER, D. MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic. *Proceedings of the ACM on Programming Languages 2*, ICFP (July 2018).

[20] KREBBERS, R., JUNG, R., BIZJAK, A., JOURDAN, J.-H., DREYER, D., AND BIRKEDAL, L. The Essence of Higher-Order Concurrent Separation Logic. In *Programming Languages and Systems* (Berlin, Heidelberg, 2017), H. Yang, Ed., vol. 10201 of *Lecture Notes in Computer Science*, Springer, pp. 696–723.

[21] KREBBERS, R., TIMANY, A., AND BIRKEDAL, L. Interactive Proofs in Higher-Order Concurrent Separation Logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (New York, NY, USA, 2017), POPL '17, Association for Computing Machinery, p. 205–217.

[22] KRISHNA, S., PATEL, N., SHASHA, D., AND WIES, T. Verifying Concurrent Search Structure Templates. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2020), PLDI 2020, Association for Computing Machinery, p. 181–196.

[23] MÉVEL, G., AND JOURDAN, J.-H. Formal Verification of a Concurrent Bounded Queue in a Weak Memory Model. *Proceedings of the ACM on Programming Languages 5*, ICFP (Aug. 2021).

[24] O'HEARN, P., REYNOLDS, J., AND YANG, H. Local Reasoning about Programs that Alter Data Structures. In *Computer Science Logic* (Berlin, Heidelberg, 2001), L. Fribourg, Ed., vol. 2142 of *Lecture Notes in Computer Science*, Springer, pp. 1–19.

[25] PUGH, W. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM 33*, 6 (June 1990), 668–676.

[26] REYNOLDS, J. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science* (2002), IEEE, pp. 55–74.

[27] SÁNCHEZ, A., AND SÁNCHEZ, C. Formal Verification of Skiplists with Arbitrary Many Levels. In *Automated Technology for Verification and Analysis* (Cham, 2014), F. Cassez and J.-F. Raskin, Eds., vol. 8837 of *Lecture Notes in Computer Science*, Springer, pp. 314–329.

[28] TASSAROTTI, J., AND HARPER, R. A Separation Logic for Concurrent Randomized Programs. *Proceedings of the ACM on Programming Languages 3*, POPL (Jan. 2019).

[29] VINDUM, S. F., AND BIRKEDAL, L. Contextual Refinement of the Michael-Scott Queue (Proof Pearl). In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New York, NY, USA, 2021), CPP 2021, Association for Computing Machinery, p. 76–90.

[30] VINDUM, S. F., FRUMIN, D., AND BIRKEDAL, L. Mechanized Verification of a Fine-Grained Concurrent Queue from Meta's Folly Library. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs* (New York, NY, USA, 2022), CPP 2022, Association for Computing Machinery, p. 100–115.

[31] XIONG, S., PINTO, P. D. R., NTZIK, G., AND GARDNER, P. Abstract Specifications for Concurrent Maps. In *Programming Languages and Systems* (Berlin, Heidelberg, 2017), H. Yang, Ed., vol. 10201 of *Lecture Notes in Computer Science*, Springer, pp. 964–990.

[32] YEON, J., KIM, L., HAN, Y., LEE, H. G., LEE, E., AND KIM, B. S. JellyFish: A Fast Skip List with MVCC. In *Proceedings of the 21st International Middleware Conference* (New York, NY, USA, 2020), Middleware '20, Association for Computing Machinery, p. 134–148.