Verification of Concurrent Search Structures

by

Nisarg Patel

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

September, 2024

Professor Thomas Wies

Dedication

To my sister Kinjal, who is a bundle of pure joy to all.

Acknowledgements

They say it takes a village to raise a child, and I feel very fortunate with the village I found myself in. My advisor, Thomas Wies, has played the biggest role in my evolution as a researcher. He has treated me with kindness and patience, and has always made himself available to me. He gently nudged me towards becoming a better researcher and has consistently kept my best interests at heart. His attention to detail and commitment to high standards are qualities that I will carry with me throughout the rest of my career. I would also like to thank Dennis Shasha, who has been my mentor from the beginning of my PhD. His enthusiasm and energy is infectious, and he has been a great source of inspiration to me.

I would like to thank Joseph Tassarotti, Aurojit Panda and Constantin Enea for being part of my committee. Beyond that, Joe and Panda have helped me at various points with my research. Constantin gave me my first taste of research during a summer internship before I started my PhD.

I would also like to thank Kedar Namjoshi for giving me the opportunity to collaborate with him for two summers at Nokia Bell Labs. I greatly enjoyed the experience and had a lot of fun learning new topics beyond my dissertation. Finally, I would like to thank the faculty at Chennai Mathematical Institute for fostering my interest in Formal Methods and setting up a solid foundation for all my future research.

I am lucky to have made so many close friendships over the years. Naming all of them individually will take up the entire page, so I will name just the factions: my childhood friends, my Chennai friends, my New York friends and my Michigan friends. There is something I have learned from all of you, and I thank you all for being part of my life.

I would like to thank my parents for shaping me into the person I am today. From my mother, I learned how to deal with adversities in life, and my father instilled in me a deep sense of curiosity about the world from a young age. I credit my elder sister, Kinjal, for my joviality and childlike sense of humor. I would also like to thank the parents of my partner, Sumegha, who have welcomed me into their family with immense warmth.

Lastly, I would not have made it this far without all the love and support from Sumegha. She was always there for me when I needed her, even as she navigated her own challenging PhD journey. Without her, I certainly would have fallen into a bureaucratic pitfall or two. She brings a lot of joy and color to my life, and I look forward to us growing together!

Abstract

Concurrent search structures are a class of concurrent data structures that implement a key-value store. Concurrent search structures are integral components of modern software systems, yet they are notoriously difficult to design and implement. In the context of concurrency, linearizability is the accepted notion of correctness of a data structure. A concurrent data structure is said to be linearizable when each of its operation appears to execute instantaneously at some point between its invocation and return point. Typically, linearizability is established by identifying a *linearization point* for each operation, i.e., an atomic step when the operation appears to take effect. Verifying linearizability of concurrent search structures remains a formidable challenge due to the inherent complexity of the underlying algorithms. The dissertation addresses this challenge by verifying practical implementations of concurrent search structures, thereby bridging the gap between theory and practice in concurrent search structure verification.

Heretofore, verification of concurrent search structures has often lead to large, intricate proofs that are hard to comprehend and reuse. Hence, this dissertation develops verification techniques that aid modularity and enables proof reuse. The notion of a *template algorithm* has been used in the literature in order to make the proofs of correctness modular in terms of the algorithmic structure. A template algorithm allows one to capture certain common design patterns, while abstracting over others. For instance, a template algorithm may dictate how concurrent threads interact but abstract away from the concrete layout of nodes in memory. Once the template algorithm is verified, its proof can be instantiated on a variety of search structures. However, prior work exploring template algorithms has been limited to lock-based concurrent search structures.

The concrete contribution of the dissertation is developing and verifying new template algorithms that cover several variants of lock-free skiplists and lock-based log-structured merge (LSM) trees. The template algorithms capture concurrency mechanisms, but abstract away node-level details as well as the style and order of the maintenance operations.

The generalizable contribution of the dissertation is the advancement in the verification technology required to prove the new template algorithms. There are two key contributions here, first relating to *hindsight reasoning* and second to *keyset reasoning*.

Hindsight reasoning has been shown to be useful for dealing with future-dependent and external linearization points, a challenge that commonly arises when dealing with lock-free and multicopy data structures. Hindsight reasoning has been explored in the literature, but not in the context of a foundational program logic. This dissertation is the first to formalize

hindsight reasoning in a general purpose program logic with foundational correctness. In particular, the dissertation embeds hindsight reasoning in the concurrent separation logic Iris via prophecy variables. The dissertation also presents evidence of significant reduction in proof size when using hindsight reasoning in Iris compared to the standard prophecy-based reasoning.

Keyset reasoning is useful for lifting assertions on a node's contents to the global contents held by the structure. The dissertation develops a *keyset resource algebra*, an Iris resource algebra to enable keyset reasoning in Iris.

All of the techniques and proofs are mechanized in Iris/Coq. Verified search structures include in particular the Michael set, the Harris list, the Herlihy-Shavit skiplist and an LSM-tree implementation based on LevelDB. The dissertation also verifies a new variant of Herlihy-Shavit skiplist previously not considered in the literature. The verification effort represents a significant contribution as it is the first mechanized proof of linearizability for concurrent skiplists and LSM-trees.

Contents

List of Figures

List of Tables

1 Introduction

The digital world has become omnipresent in our day to day life. Almost all computer systems that manage the digital world, from giant databases to smartphones, are composed of multiple processors that operate on shared data for efficiency. Having n processors, however, does not immediately imply a factor of n increase in speed, because the individual processors also need to communicate and synchronize with each other over the shared data. The shared data is often organized into so called *concurrent data structures*, and the stored data is accessed by operations on these data structures. The algorithms that perform these operations must be designed carefully in order to function correctly. For instance, there must not be two processes adding contradictory information to the shared data.

Unfortunately, these algorithms are also among the most difficult software artifacts to develop correctly. Real-world systems are often affected by software bugs due to concurrency. The Northeast blackout of 2003 was a power outage in northeastern and midwestern parts of US and Canada affecting 55 million people in total [128]. The blackout was caused in part due to a concurrency bug in the alarm systems. A similar bug in the radiation therapy machine Therac-25 caused severe overdose and led to the death of at least six people [110]. The digital platforms of financial and government services, which cater to hundreds of thousands of people everyday, are also plagued by such problems. It is clear therefore that we need systematic and dependable techniques to reason about and ensure correctness of these complex algorithms.

Formal verification is the process of analyzing a system against a formal specification or property using mathematical techniques. In the context of software systems, formal verification techniques range from ensuring generic correctness properties (such as the absence of run-time errors) to proving full functional correctness. Functional correctness involves demonstrating that a program's behavior aligns with its expected behavior.

Techniques for checking generic correctness properties include static analysis and model checking. Static analysis tools such as Error Prone for Java [57], Infer [41], Clang Static Analyzer [156] and the ASTRÉE analyzer [14] are developed and integrated into the workflow at major companies in the software industry. These tools automatically analyze millions of lines of code to identify bugs. In addition to static analysis, Amazon has incorporated model-checking to verify the distributed protocols and authentication mechanisms underlying AWS cloud services [146, 165]. However, to achieve a high degree of scalability and automation, these tools either under-approximate program behaviors, potentially missing critical bugs, or over-approximate and generate spurious errors as false positives.

To provide the highest level of assurance for a system, one must prove its functional correctness. Notable projects that have established the functional correctness of real-world software systems include the verified optimizing C compiler CompCert [107], the verified OS microkernel seL4 [91], and the verified HTTPS replacement developed in Project Everest [12]. The downside of proving functional correctness is that it requires several person-years of work by verification experts.

In light of this context, this dissertation aims to prove the functional correctness of *concurrent search structures*. A search structure is a key-based store that implements a mutable set of keys, i.e. the `Set` abstract datatype (ADT) or, more generally, a mutable map of keys to values, i.e. the `Map` ADT. A search structure is also commonly known as a dictionary data structure. It provides five basic operations: (i) create an empty structure, (ii) insert a key-value pair, (iii) search for a key and return its value, (iv) delete the entry associated with a key, and (v) update the value associated with a particular key.

This dissertation presents a strategy designed to minimize the manual effort involved in proving the functional correctness of commonly used concurrent search structures. We accomplish this by enhancing the modularity of the verification process and enabling proof reuse across diverse data structure implementations.

The correctness criterion we target in this dissertation is that of *linearizability* [70]. A concurrent data structure is linearizable if for every concurrent execution history of its operations, each operation appears to take effect at a single atomic step between its invocation and return point.

We note that our focus is on establishing partial correctness, meaning we do not provide guarantees of progress, such as ensuring that a thread executing an operation on a search structure will always terminate. In this dissertation, we assume that all programs operate within a garbage-collected environment. This assumption is justified by the fact that issues related to manual memory reclamation can be handled separately. For example, [120, 121] propose a technique that separates the proof of data structure correctness from the underlying memory reclamation algorithm, allowing the correctness proof to be carried out under the assumption of garbage collection. Recent work has demonstrated how to perform such modular proofs within concurrent separation logics [81]. Finally, this dissertation also assumes sequential consistency as the underlying memory model for execution. We discuss this assumption and related works in Chapter 10.

1.1 State of the Art

The inherent complexity of concurrrent algorithms comes from the combination of managing concurrent interference while simultaneously organizing data in memory to optimize performance. As a result, the state of the art verification techniques for concurrent data structures have emphasized proof modularity. There are four main types of modular proof techniques: (i) Hoare logic [72] enables proofs to be compositional in terms of program structure; (ii) local reasoning techniques [10, 89, 99, 100, 125, 132, 144] allow proofs of programs to be decomposed in terms of the state they modify; (iii) thread modular techniques [70, 80, 136]

allow one to reason about each thread in isolation; and (iv) refinement techniques [7, 37, 73] enable reasoning about properties of a system at different levels of abstraction.

Separation Logics (SL) [18, 19, 33, 39, 40, 42, 48, 56, 86, 106, 127, 130, 157, 164] incorporating all of the above techniques have led to great progress in the verification of practical concurrent search structures.

A particular breakthrough came with [97] which demonstrated a modular proof methodology for verifying complex real-world data structures such as (lock-based) B-link trees, hash-tables and linked lists. The core ingredients of the methodology are the *template algorithms*, the *flow framework* [99, 100] and the concurrent separation logic Iris [82, 83, 86, 94–96]. We explain each ingredient in detail.

A template algorithm captures certain common design patterns, while abstracting over others. For instance, a template algorithm may dictate how concurrent threads interact (synchronization) but abstract away from the concrete layout of nodes in memory (memory representation). Once the template algorithm is verified, its proof can be instantiated on a variety of search structures. The template algorithms for concurrent search structures were first introduced by Shasha and Goodman [153], who also identified the invariants needed for decoupling reasoning about synchronization and memory representation for such data structures.

The flow framework provides an SL-based abstraction mechanism that allows one to reason about global inductive invariants of general graphs in a local manner. Using this framework, the methodology in [97] performs separation-logic-style reasoning about the correctness of a concurrent search structure template while abstracting from the specific low-level heap representation of the underlying data structure.

The final ingredient is the separation logic Iris. The approach in [97] shows how to capture the high-level proof idea in terms of Iris resource algebras, yielding a general methodology for modular verification of concurrent search structures.

1.2 Contributions

We categorize search structures as either *single-copy*, i.e., structures that contain at most one copy of a key at any time (like B-trees and linked-lists) or *multicopy*, i.e., structures that may contain multiple copies of a single key such as Log-Structured Merge (LSM) Trees [134]. Single-copy structures follow the *in-place* update design. For example, if key k has an associated value 5 and then this is changed to 17, an algorithm on a single-copy structure would find the node containing k and change the value from 5 to 17. By contrast, multicopy structure follow the *out-of-place* update design. Using the previous example, a new pair $(k, 17)$ would be prepended to the structure in a multicopy structure. A subsequent search on k would return the value associated with the most recent pair prepended to the structure, 17 in this case.

The pragmatic advantage of multicopy structures is that all modifications update the root node of the structure, so they can be processed very fast. The pragmatic disadvantage when compared to single-copy structures is that searches can take longer and the data structure

may grow larger.

The proof methodology in [97] targets lock-based single-copy structures. This dissertation extends the proof methodology to encompass lock-based multicopy structures and lock-free single-copy structures. There are significant challenges when extending the proof methodology to a broader class of concurrent search structures. This dissertation presents generalizable solutions to these challenges, advancing the verification technology in the process.

We next discuss the contributions in detail. All of the techniques and proofs are mechanized in Iris/Coq. Verified search structures include in particular the Michael set [122], the Harris list [62], the Herlihy-Shavit skiplist [68, § 14] and an LSM tree implementation based on LevelDB. The dissertation also verifies a new variant of the Herlihy-Shavit skiplist previously not considered in the literature. The verification effort represents a significant contribution as it is the first mechanized proof of linearizability for concurrent skiplists and LSM-trees.

1.2.1 Contribution 1: Novel Template Algorithms

The dissertation innovates on the template algorithms from [97, 153] in the following two ways:

1. We introduce (i) an LSM-DAG Template that encompasses a broad class of (lock-based) LSM Tree implementations; (ii) a Skiplist Template that encompasses lock-free linked-lists and skiplists.

2. The novel templates include maintenance operations in addition to the core operations (search, insert, delete, etc.). Thus, instantiations of our templates are closer to their real-world counterparts.

The LSM-DAG Template. The LSM Tree is a prominent example of a multicopy structure. The data structure consists of a root node stored in memory, and a linked list of nodes stored on disk. New updates (to a key) are added at the root node. When the root node becomes full, the data is *flushed* out to nodes on the disk. A search for a key k traverses the list starting from the root node and retrieves the value associated with the first copy of k that is encountered.

The LSM tree can be tuned by implementing workload- and hardware-specific data structures at the node level. In addition, research has been directed towards optimizing the layout of nodes and developing different strategies for the maintenance operations used to reorganize these data structures. This has resulted in a variety of implementations today (e.g. [35, 112, 143, 160, 169]). Despite the differences between these implementations, they generally follow the same high-level algorithms for the core search structure operations.

We construct a template algorithm for concurrent multicopy structures from the high-level descriptions of their operations and then prove the correctness of these operations. Notably our LSM DAG template generalizes the LSM tree so that the outer data structure

can be a DAG rather than just a list. The template also permits implementations that utilize the DAG structure of the underlying graph to organize the data, such as a data structure that combines the LSM tree and B-link tree.

The Skiplist Template. Locks ensure non-interference on portions of memory to guarantee that certain needed constraints hold in spite of concurrency. The disadvantage of locks is that if a thread holding a lock on some portion of memory p stops, then no other thread can get a conflicting lock on p. For that reason, some practical implementations such as Java's `ConcurrentSkipListMap` [135] use lock-free algorithms.

This dissertation shows how to capture multiple variants of concurrent lock-free skiplists and linked lists in the form of template algorithms. Thus, proving the correctness of such a template algorithm results in a proof that is applicable to many variants at once. Our template algorithms are parametric in the skiplist height and allow variations along the following three dimensions: (i) maintenance style (eager vs lazy) (ii) node implementations and (iii) the order of maintenance operations on the higher levels of the skiplists.

By instantiating our template algorithm with appropriate maintenance operations and node implementations, we obtain verified versions of existing (skip)list algorithms from the literature such as the Herlihy-Shavit skiplist algorithm, the Michael set, and the Harris list algorithm. We also obtain new concurrent skiplist algorithms that have not been considered before. These new algorithms are correct by construction thanks to our modular verification framework.

1.2.2 Contribution 2: Hindsight Reasoning in Iris

Linearizability in thread-modular logics such as Iris is defined via *(logically) atomic triples* [33, 85, 86]. Intuitively, an atomic triple $\langle x.\ P \rangle\ e\ \langle v.\ Q \rangle$ says that at some point during the execution of e, the resources described by the precondition P will be updated to satisfy the postcondition Q for return value v in one atomic step. The variable x can be thought of as the abstract state of the data structure before the update at the linearization point.

Linearizability of a data structure operation `op` can be expressed by an atomic triple of the form

$$\langle C.\ \mathsf{DS}(r, C) \rangle\ \mathtt{op}\ r\ \langle res.\ \exists\ C'.\ \mathsf{DS}(r, C') * \Psi_{\mathtt{op}}(C, C', res) \rangle. \qquad \text{(ClientSpec)}$$

Here, r is the pointer that provides access to the data structure. The predicate $\mathsf{DS}(r, C)$ is the *representation predicate* that relates the root pointer with the abstract state C of the structure. For instance, in the case of a lock-free linked list, the abstract state refers to the set of keys contained in the structure. The predicate $\Psi_{\mathtt{op}}(C, C', res)$ captures the sequential specification of the structure. The specification essentially says there is a single atomic step in `op` where the abstract state changes from C to C' according to the sequential specification $\Psi_{\mathtt{op}}(C, C', res)$. This step is `op`'s linearization point. We call (ClientSpec) the *client-level* atomic specification for the data structure under proof.

A particular challenge when proving the (ClientSpec) for the new templates is that their operations exhibit future-dependent and external linearization points. A linearization point

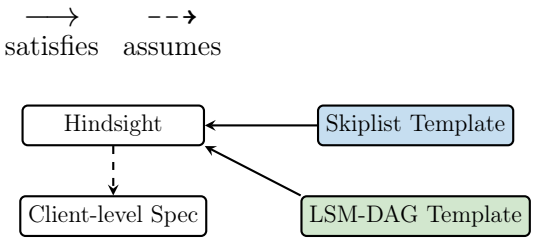


Figure 1.1: Proof Organization. Each box represents a collection of modules relevant to the label. The dashed arrows represent module dependence, i.e., assumption of specifications. The normal arrows represent implementation of the target module (fulfillment of the assumptions).

is said to be future-dependent if it cannot be determined at any fixed moment, but only at the end of the execution, once any interference of other concurrent operations has been accounted for. A linearization point of a thread is said to be external when it lies in a different concurrent thread. Such linearization points are typically difficult to determine, and reasoning about them in Iris requires *prophecy variables* [1, 85] and a *helping protocol* [81, 85].

Hindsight reasoning [46, 47, 108, 117, 118, 133] has emerged as a suitable technique to address the challenge of verifying data structures exhibiting future-dependent and external linearization points. It enables temporal reasoning about computations using a *past predicate* $\diamond p$, which expresses that proposition p held true at some prior state in the computation. A critical step when using hindsight reasoning is the *temporal interpolation* [118], i.e., a lemma of the form: if there existed a past state that satisfied property p and the current state satisfies q, then there must have existed an intermediate state that satisfied o. We set up temporal interpolation in a manner so that the inferred property o implies a linearization point. Hence, a thread which has knowledge about a past state that it witnessed and the current state that it observes can employ temporal interpolation to infer its linearization point in hindsight.

To our knowledge, this dissertation is the first to embed hindsight reasoning within a foundational program logic (Iris). The dissertation proposes a *hindsight framework* whose main feature is connecting linearizability in hindsight to the atomic triple linearizability. This connection is made via an intermediate specification, called the *hindsight specification*, and a proof that any data structure that satisfies the hindsight specification also satisfies its atomic triple specification. The proof involves a common helping protocol that is applicable to a broad class of data structures and is done once and for all.

From the perspective of a proof author using the hindsight framework to verify a concurrent structure, one has to establish only the hindsight specification to prove the atomic triple specification. The hindsight specification is a Hoare Triple specification [72]. In essence, it asks for a thread to establish linearizability in hindsight at the end of its execution. The proof author does not have to reason about the helping protocol directly in the verification process. Moreover, the framework also provides a mechanism for storing an abstraction of the history of computation so that the proof author can express properties about the past state using the $\diamond$ modality.

We use the hindsight framework to establish linearizability of the Skiplist and LSM-DAG Templates. Figure 1.1 provides an overview of our verification effort. The module "Hindsight" captures the assumptions regarding the hindsight specification. The module "Client-level Spec" relates the client-level specification expressed in terms of atomic triples to the hindsight specification used for the template-level proofs. The corresponding proof involves the reasoning about prophecies and the helping protocol, which is done once and for all and applicable to all data structures that fulfill the assumptions made in the "Hindsight" module. Afterwards, we show that the "Skiplist Template" and the "LSM-DAG Template" satisfy the requirements of the "Hindsight" module.

1.2.3 Contribution 3: Keyset Reasoning in Iris

The notion of keysets was introduced in [153] where it was integral to the correctness reasoning for single-copy search structures laid out in that paper. A keyset is a node-level quantity and intuitively, a keyset of a node refers to the set of keys that a node is *responsible* for. Or, in other words, if k is in the keyset of a node n, then n should contain k or k should not be in the structure.

In single-copy search structures, the keysets of all nodes partition the set of all keys and provide the crucial *Keyset Property*:

$$\forall\, n \in N, k \in \mathsf{K}.\ k \in \mathsf{ks}(n) \Rightarrow (k \in C(N) \Leftrightarrow k \in C(n)) \qquad (\mathsf{KeysetPr})$$

This property enables one to lift a proof of the specification at the node level to a proof of the sequential specification of the search structure. A particular situation where (KeysetPr) proves indispensable is when `search` fails to find the search key. Note that `search` observes only the nodes it visited, and hence has only a partial view of the structure. When `search` fails to find the key, the proof has to reconcile this partial view of the structure with the global view. In essence, if a concurrent invocation of `search` on key k fails to find the key, can we conclude that there was a point in time during its execution when k was in fact not present in the structure? Here, the property (KeysetPr) helps us reconcile facts gathered by search with the global state of the structure. Specifically, if `search` can determine a node n such that $k \in \mathsf{ks}(n)$ and $k \notin C(n)$, then we can immediately infer that k was not present in the structure at that point in time.

To capture the properties of keysets naturally in Iris, the dissertation introduces a *keyset resource algebra (RA)*, a form of ghost state in Iris. An element of the keyset RA is a pair of sets of keys (K, C) with $C \subseteq K$. We associate such pairs to regions in the data structure graph such that C is the union of the node contents in the region and K is the union of the nodes' keysets. The composition of such pairs is defined as (pair-wise) disjoint union. The validity and the composition operator of the keyset RA then implicitly capture the desired properties of keysets and ensure (KeysetPr).

The keyset RA has proven to be indispensable to the verification of single-copy search structures. The keyset RA was first introduced in [97]. However, this dissertation claims the intellectual contribution of the development of the keyset RA from that verification effort. The keyset RA is also used for the verification of the Skiplist Templates.

1.3 Outline

The dissertation is composed of three parts. The first part provides the necessary background beginning with some preliminary concepts in Chapter 2. This is followed by Chapter 3 which explains the core ingredients of the proof methodology of [97], i.e., the Edgeset Framework, the templates for (lock-based) single-copy structures and the Flow framework (Chapter 3). Chapter 4 provides basics of separation logic and Iris (Chapter 4) that is necessary for proof formalization.

The second part of the dissertation presents the contributions. Chapter 5 explains the keyset RA and Chapter 6 develops the Hindsight Framework. These techniques are then applied towards the verification of the the LSM-DAG Templates (Chapter 7) and the Skiplist Templates (Chapter 8). Chapter 9 offers the evaluation of the verification effort from Chapter 8 and Chapter 7.

The final part of the dissertation presents a discussion of related work (Chapter 10) and conclusion (Chapter 11).

Funding Acknowledgements

This work was funded in parts by NYU Wireless and by the United States National Science Foundation under grants CCF-2304758, 1840761, 2304758, and 25-74100-F1202. Further funding came from an Amazon Research Award Fall 2021. Any opinions, findings, and conclusions or recommendations expressed in this dissertation are those of the author and do not reflect the views of Amazon.

Publication Note

Parts of this dissertation have appeared (or will appear) in the following publications below.

Refereed Publications

1. Siddharth Krishna, Nisarg Patel, Dennis Shasha, and Thomas Wies. 2020. Verifying concurrent search structure templates. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020). Association for Computing Machinery, New York, NY, USA, 181-196.
   DOI : https://doi.org/10.1145/3385412.3386029

2. Nisarg Patel, Siddharth Krishna, Dennis Shasha, and Thomas Wies. 2021. Verifying concurrent multicopy search structures. Proc. ACM Program. Lang. 5, OOPSLA, Article 113 (October 2021), 32 pages.
   DOI : https://doi.org/10.1145/3485490
   Parts of texts in Chapter 1, Chapter 7, Chapter 9 and Chapter 10 originated from this publication.

3. Nisarg Patel, Dennis Shasha, and Thomas Wies. 2024. Verifying Lock-free Search Structure Templates. 2024.
   To appear at ECOOP 2024.
   Parts of texts in Chapter 1, Chapter 9 and Chapter 10 originated from this publication.
   Chapter 6 and Chapter 8 are reproduced from this publication.

Monograph

1. Siddharth Krishna, Nisarg Patel, Dennis Shasha, and Thomas Wies. 2021. Automated Verification of Concurrent Search Structures. Springer Nature Switzerland AG 2021.
   DOI : https://doi.org/10.1007/978-3-031-01806-0
   Parts of texts in Chapter 1 and Chapter 10 originated from this monograph.
   Chapter 2, Chapter 4, Chapter 3 and Chapter 5 are reproduced from this monograph with minor changes.

Part I: Background

2 | Preliminaries

This chapter provides technical background for some concepts used in this dissertation, namely, basic mathematical notation and the programming language we use.

2.1 Basics and Notation

We begin with some basic definitions and notation.

- The term $(b \,?\, t_1 : t_2)$ denotes t_1 if condition b holds and t_2 otherwise.
- We write $f \colon A \to B$ for a total function from A to B, and $f \colon A \rightharpoonup B$ for a partial function from A to B.
- For a partial function f, we write $f(x) = \bot$ if f is undefined at x.
- We use the lambda notation $(\lambda x.\ E)$ to denote a function that maps x to the expression E (typically containing x).
- If f is a (partial) function from A to B, we write $f[x \rightarrowtail y]$ to denote the function from $A \cup \{x\}$ defined by $f[x \rightarrowtail y](z) \coloneqq (z = x \,?\, y : f(z))$.
- We use $\{x_1 \rightarrowtail y_1, \ldots, x_n \rightarrowtail y_n\}$ for pairwise different x_i to denote the function $\epsilon[x_1 \rightarrowtail y_1] \cdots [x_n \rightarrowtail y_n]$, where ϵ is the function on an empty domain.
- Given functions $f_1 \colon A_1 \to B$ and $f_2 \colon A_2 \to B$, we write $f_1 \uplus f_2$ for the function $f \colon A_1 \uplus A_2 \to B$ that maps $x \in A_1$ to $f_1(x)$ and $x \in A_2$ to $f_2(x)$ (if A_1 and A_2 are not disjoint sets, $f_1 \uplus f_2$ is undefined).
- We also write $\lambda_0 \coloneqq (\lambda m.\ 0)$ for the identically zero function and $\lambda_{\mathsf{id}} \coloneqq (\lambda m.\ m)$ for the identity function.
- For functions f_1, f_2, we write $f_2 \circ f_1$ to denote function composition, i.e. $(f_2 \circ f_1)(x) = f_2(f_1(x))$, and use superscript notation f^p to denote the function composition of f with itself p times.

- For multisets, we use the standard set notation if it is clear from the context. We also write $\{x_1 \rightarrowtail i_1, \ldots, x_n \rightarrowtail i_n\}$ for the multiset containing i_1 occurrences of x_1, i_2 occurrences of x_2, etc. For a multiset S, we write $S(x)$ to denote the number of occurrences of x in S.

- We write _ for an anonymous variable, usually to bind a variable that is never used.

We now turn to introducing some basic algebraic concepts that will be used in the later chapters.

Definition 1 *A* partial monoid *is a set M, along with a partial binary operation $+ \colon M \times M \rightharpoonup M$, and a special zero element $0 \in M$, such that*

(1) $+$ is associative, i.e., $(m_1 + m_2) + m_3 = m_1 + (m_2 + m_3)$; and

(2) 0 is an identity element, i.e., $m + 0 = 0 + m = m$.

Here, equality means that either both sides are defined and equal, or both sides are undefined.

Partial monoids are the basis of ghost state, an important reasoning technique which will be introduced in Chapter 4. An example of a partial monoid is the set $\mathcal{P}(S)$ of all subsets of an arbitrary set S, together with disjoint union $\uplus$ (where $S_1 \uplus S_2$ is undefined if they are not disjoint) and the empty set $\emptyset$. We usually identify a partial monoid $(M, +, 0)$ with its support set M.

If $+$ is a total function, then we call M a monoid. For example, the set of natural numbers $\mathbb{N}$ together with addition $+$ and zero form a monoid. Let $m_1, m_2, m_3 \in M$ be arbitrary elements of the (partial) monoid in the following. Here is some terminology and notation associated with (partial) monoids:

- We call a (partial) monoid M *commutative* if $+$ is commutative, i.e., $m_1 + m_2 = m_2 + m_1$. $\mathbb{N}$ and $\mathbb{Z}$ are commutative monoids, while 2×2 matrices of natural numbers with matrix multiplication and the identity matrix form a non-commutative monoid.

- Similarly, a commutative (partial) monoid M is *cancellative* if $+$ is cancellative, i.e., if $m_1 + m_2 = m_1 + m_3$ is defined, then $m_2 = m_3$. For example, $\mathbb{N}$ is cancellative monoid while the monoid formed by sets of natural numbers under set union is not (as $\{1\} \cup \{1, 2\} = \{1\} \cup \{2\}$).

- We say M is *positive* if $m_1 + m_2 = 0$ implies that $m_1 = m_2 = 0$. As you might expect, $\mathbb{N}$ is a positive (partial) monoid, while $\mathbb{Z}$ is not positive.

- For a positive (partial) monoid M, we can define a partial order $\leq$ on its elements as $m_1 \leq m_2$ if and only if $\exists m_3.\ m_1 + m_3 = m_2$. Positivity also implies that every $m \in M$ satisfies $0 \leq m$. For $\mathbb{N}$, this order corresponds to the natural less-than-or-equal-to ordering on natural numbers.

2.2 Programming Language

The programming language that we use in this dissertation is an ML-like language with higher-order store, fork, and compare-and-set, whose grammar is given below:

$e \in \mathsf{Expr} ::=$	x	Variable
$\mid$	`()`	Unit constant
$\mid$	z	Integer constants
$\mid$	`true` $\mid$ `false`	Boolean constants
$\mid$	e_1 `+` e_2 $\mid$ e_1 `-` e_2 $\mid \ldots$	Arithmetic expressions
$\mid$	e_1 `==` e_2 $\mid$ e_1 `<=` e_2 $\mid \ldots$	Boolean expressions
$\mid$	(e_1, e_2)	Pair (tuple) expressions
$\mid$	**`if`** e_1 **`then`** e_2 **`else`** e_3	Conditional
$\mid$	$e_1\ e_2$	Function application
$\mid$	$(\mu\ f\ x.\ e)$	(Recursive) function
$\mid$	**`let`** x `=` e_1 **`in`** e_2	Let binding
$\mid$	$\texttt{Inj}_1\ e \mid \texttt{Inj}_2\ e$	Constructor expressions
$\mid$	**`match`** e **`with`** $\texttt{Inj}_1\ x$ `->` e_1 `|` $\texttt{Inj}_2\ x$ `->` e_2	Pattern matching
$\mid$	**`ref`**(e)	Reference creation
$\mid$	$!e$	Dereference
$\mid$	$e_1 \leftarrow e_2$	Reference assignment
$\mid$	**`CAS`**(e, e_1, e_2)	Compare-and-set
$\mid$	**`FAA`**(e, e_1)	Fetch-and-add
$\mid$	**`fork`** $\{e\}$	Fork

As is standard in languages based on the λ-calculus, all programs are expressions, the simplest kind of which is just a variable or constant value. The constant values in our language include the unit value (the return value of expressions that are evaluated only for their side-effects such as assignments), integers, and Booleans. Arithmetic and Boolean expressions use infix notation and conditional expressions are as expected.

Function application is written in standard functional programming style as `foo arg` instead of `foo(arg)`. It is also left-associative, i.e., a nested function application `foo arg1 arg2` should be read as `(foo arg1) arg2`. A function expression $(\mu\ f\ x.\ e)$ evaluates to the function that satisfies the (possibly recursive) equation $f(x) = e$. For example, the following expression evaluates to the factorial function

$$(\mu\ fac\ x.\ \textbf{`if`}\ x \texttt{ == } 0\ \textbf{`then`}\ 1\ \textbf{`else`}\ x \texttt{ * } fac\ (x \texttt{ - } 1))$$

A lambda abstraction expression $(\lambda x.\ e)$, which describes a function that maps argument x to the expression e, can be defined as a syntactic shorthand $(\lambda x.\ e) \coloneqq (\mu\ \_\ x.\ e)$.

A let binding expression **`let`** x `=` e_1 **`in`** e_2 binds the variable x to the result value of the expression e_1, and evaluates e_2 with this new binding. That is, e_2 in general would depend on x, so is evaluated assuming x is equal to e_1. The value obtained from e_2 is then also

the result value of the whole let binding expression. In the rest of the dissertation, we use standard syntactic shorthands, such as:

$$\begin{aligned} e_1;\ e_2 &\coloneqq \texttt{let}\ \_ = e_1\ \texttt{in}\ e_2 \\ \texttt{let}\ f\ x = e_1\ \texttt{in}\ e_2 &\coloneqq \texttt{let}\ f = (\lambda x.\ e_1)\ \texttt{in}\ e_2 \\ \texttt{let rec}\ f\ x = e_1\ \texttt{in}\ e_2 &\coloneqq \texttt{let}\ f = (\mu\ f\ x.\ e_1)\ \texttt{in}\ e_2 \end{aligned}$$

We also omit the `in` keyword when defining top-level functions.

The language provides constructors $\texttt{Inj}_1$ and $\texttt{Inj}_2$ to construct values of a generic disjoint sum type. If e evaluates to value v, then $\texttt{Inj}_i\ e$ evaluates to $\texttt{Inj}_i\ v$ which can be thought of as a pair consisting of v and a tag bit that encodes i. Such tagged values can be decomposed using pattern matching expressions such as:

$$\texttt{match}\ e_0\ \texttt{with}\ \texttt{Inj}_1\ x\ \texttt{->}\ e_1\ \texttt{|}\ \texttt{Inj}_2\ x\ \texttt{->}\ e_2$$

Here, if e_0 evaluates to $\texttt{Inj}_i\ v$, then the match expression binds x to v and continues evaluating e_i under this binding. The result value of the match expression is that of the chosen e_i. Algebraic data types, which are commonly supported in functional languages, can be encoded using disjoint sums. For instance, consider the ML type α `option` where α is a type parameter. This type has two constructors: `None` and `Some` e where e must evaluate to a value of type α. The type can be used, e.g., to turn a partially defined function returning values of type α into a total function returning values of type α `option` by using `None` to indicate the absence of a return value and `Some` x to indicate that the return value is x. A caller of the function can then distinguish these two cases by pattern matching on the constructor of the return value. We can encode this type by letting `None` $\coloneqq \texttt{Inj}_1$ `()` and `Some` $x \coloneqq \texttt{Inj}_2\ x$.

The reference creation expression $\textbf{ref}(e)$ evaluates to the address of a newly allocated heap location, whose value is set to the result of evaluating the expression e. Heap locations, or references, can be read (or *dereferenced*) by using the $!e$ command, where e evaluates to a heap location. A heap location obtained by evaluating an expression e_1 can be updated to the value of e_2 using an assignment $e_1 \leftarrow e_2$, which returns the unit value `()`. The compare-and-set expression $\textbf{CAS}(e, e_1, e_2)$ is an instruction that checks if the value at heap location ℓ obtained by evaluating e is equal to the value of e_1; if so, it sets ℓ's value to the value of e_2 and returns `true`, otherwise it does nothing and returns `false`. The compare-and-set is *atomic* once the argument expressions are evaluated, which means that during the execution of the comparison and update, no other thread can read from or write to ℓ (in other words, they appear to take place instantaneously). The fetch-and-add expression $\textbf{FAA}(e, e_1)$ increments the value at heap location resulting from evaluation of e by the value of e_1. Like compare-and-set, fetch-and-add is also atomic. The fork command $\textbf{fork}\ \{e\}$ creates a new thread that evaluates expression e in parallel with the forking thread. The expression returns the unit value `()` without waiting for the new thread to complete its execution.

3 Ingredients for modular verification

This chapter introduces the *edgeset framework* that allows one to view a single-copy search structure as an abstract graph whose nodes are labelled by their contents and edges are labelled by *edgesets*. Such an abstract view allows us to define template algorithms for concurrent search structures that fix a concurrent technique but can be instantiated to multiple concrete data structures. The edgeset formulation we use, and the template algorithms based on them, comes from Shasha and Goodman [153].

We first present the intuition behind the fundamental concept of an edgeset and show that it applies across existing search structures. Next we describe the B-link tree data structure, a highly-efficient and popular algorithm that uses the *link* technique of synchronization. We then apply the edgeset notion to derive a template algorithm that can be instantiated to any concurrent search structure algorithm that supports a link-based redirection for operations that arrive at an incorrect node, including the B-link tree. We end the chapter with a discussion of the Flow Framework that allows reasoning about global graph properties in a local manner. The Flow Framework makes it possible to reason about edgesets and related quantities in a local manner.

3.1 An Intuitive Introduction to Edgesets

Every search structure supports the notion of navigation. This implies that searches (and the search portion of inserts, deletes, and updates) follow edges (normally in the form of either explicit or implicit pointers) as they traverse a path to an appropriate destination node. For example if a search on a binary search tree for key 5 arrives at a node n having value 7, then the search will proceed to the left child of n. Similarly a search for key 10 on a hash structure characterized by hash function h will proceed to the node $h(10)$.

In this framework, we associate each edge (n, n') of a search structure with a set of key values called the *edgeset*, written $\mathsf{es}(n, n')$. If a key k belongs to $\mathsf{es}(n, n')$, then a search that arrives at n will proceed to n'. In a sorted linked list (Figure 3.1, left), if node n has a key 6, then $\mathsf{es}(n, n.\texttt{next})$ consists of all values greater than 6. In a binary search tree (Figure 3.1, middle), if node n has the key value 2, then $\mathsf{es}(n, n.\texttt{left})$ consists of all values less than 2. Similarly, for a hash structure having hash function h and a bucket i, the edgeset from the

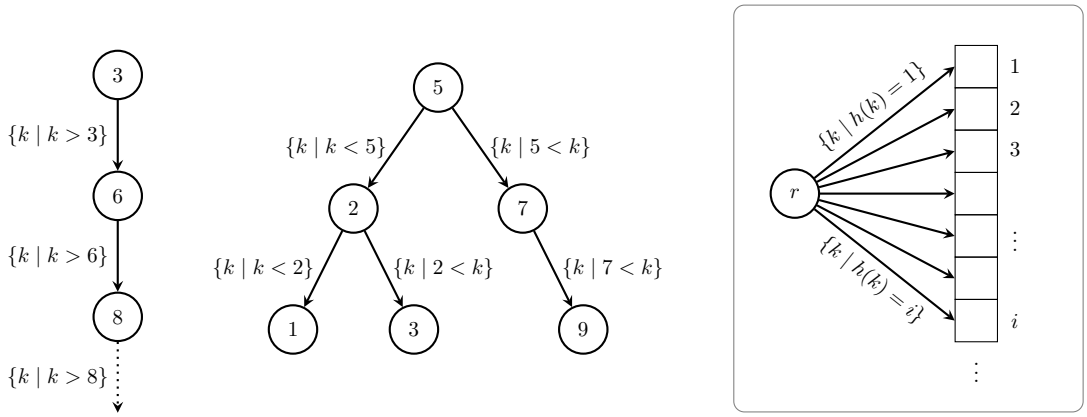


Figure 3.1: Examples of edgesets shown as edge labels. Left: sorted linked list, middle: binary search tree, right: a hash structure.

root of the hash structure to i consists of $\{k \mid h(k) = i\}$ (Figure 3.1, right).

Thus, the edgeset concept gives us the means to express a search/insert/update/delete algorithm that applies to any search structure. A search for key k starts at a root (it is fine for search structures to have many roots) and stops at node n if no edgeset leaving n contains k.

On well-formed search structures, edgesets have the following two properties:

1. For every node n in a search structure, the edgesets leaving n are mutually exclusive. This ensures that searches have a unique next node to navigate to.

2. The set of keys in n, denoted $\mathsf{contents}(n)$, will be disjoint with the edgesets leaving n. This ensures that a search for k will not leave node n when k is in node n.

In a binary search tree, for example, the edgesets of the left and right children of node n are disjoint and are of course disjoint from the key in node n.

3.2 B-link Trees

The B-link tree (Figure 3.2) is an implementation of a concurrent search structure based on the B-tree. A B-tree is a generalization of a binary search tree, in that a node can have more than two children. In a binary search tree, each node contains a key k_0 and up to two pointers y_l and y_r. An operation on k takes the left branch if $k < k_0$ and the right branch otherwise. A B-tree generalizes this by having l sorted keys $k_0, \ldots, k_{l-1}$ and $l + 1$ pointers $y_0, \ldots, y_l$ at each node, such that $B \leq l + 1 < 2B$ for some constant B. At internal nodes, an operation on k takes the branch y_i if $k_{i-1} \leq k < k_i$.

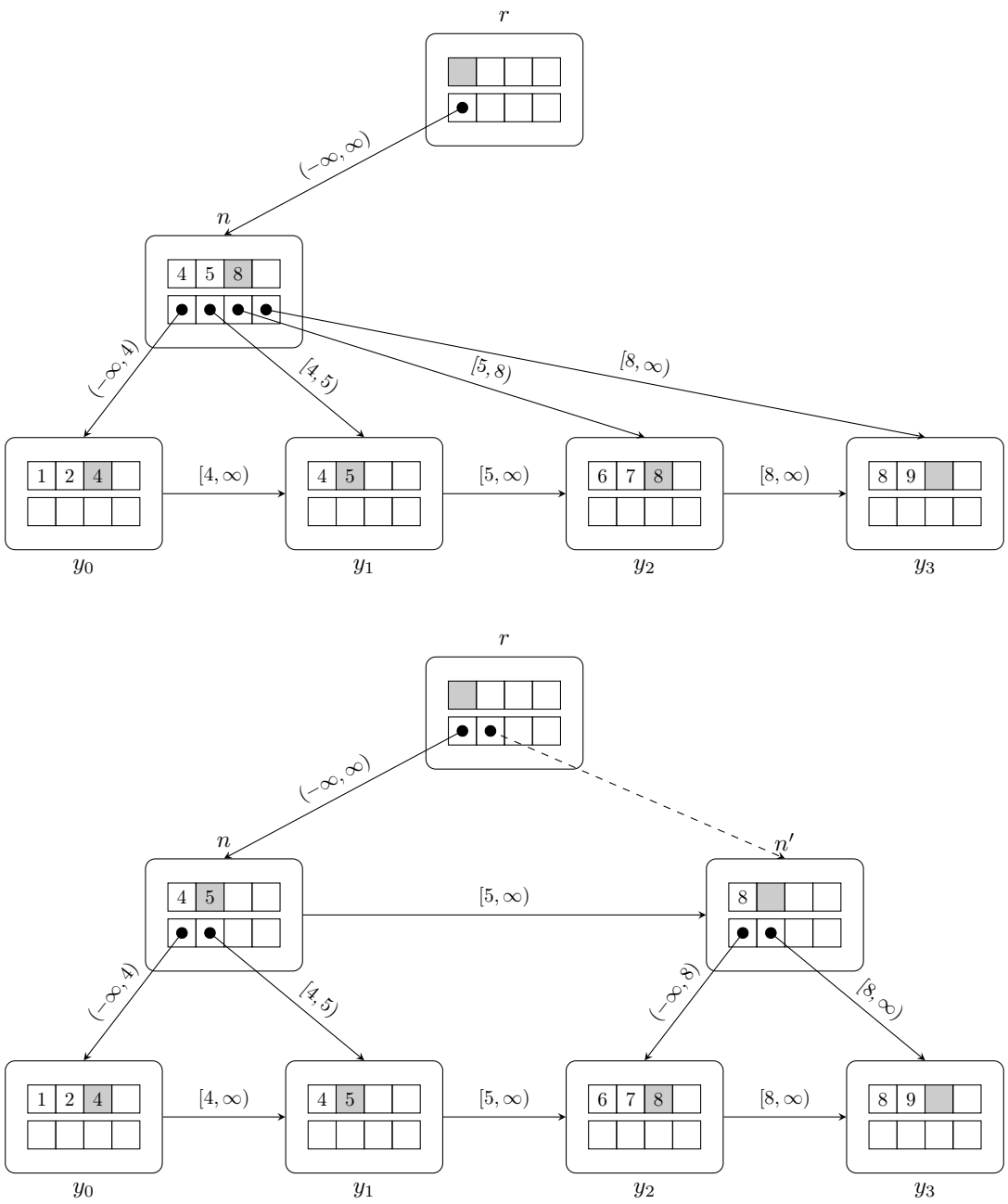


Figure 3.2: B-link tree half-split operation. A B-link tree before (top) and after (bottom) a half-split on node n, which was full, that transferred children y_2 and y_3 to a new node n'. A subsequent complete-split will add n' to the parent r (the dashed edge). Each node contains an array of keys (top array) and an array of pointers (bottom array), and a separator value in a gray box directing operations on larger keys to follow the link edge to the right-neighboring node. Edges are labeled by their edgeset (§3.3).

In the most common implementations of B-trees (called B+ trees), the keys are stored only in leaf nodes; internal nodes contain "separator" keys for the purpose of routing only, and therefore are not part of the contents of the structure. For example, the search structure depicted in Figure 3.2 (bottom) has key contents $\{1, 2, 4, 6, 7, 8, 9\}$. When an operation arrives at a leaf node n, it proceeds to insert, delete, or search for its operation key in the keys of n. To avoid interference, each node has a lock that must be held by an operation before it reads from or writes to the node.

When a node n becomes full, a separate maintenance thread performs a split operation by transferring half its keys (and pointers, if it is an internal node) into a new node n', and adding a link to n' from the parent of n. A concurrent algorithm needs to ensure that this operation does not cause concurrent operations at n looking for a key k that was transferred to n' to conclude that k is not in the structure. The B-link tree solves this problem by linking n to n' and storing a key k' (the key in the gray box in the figure) that indicates to concurrent operations that the key k can be reached by following the link edge if $k > k'$. (That is what we mean by the intuitive notion of redirection above.)

To reduce the time the parent node is locked, this split is performed in two steps: (i) a half-split step that locks n, transfers half the keys to n', and adds a link from n to n' and (ii) a complete-split performed by a separate thread that takes a half-split node n, locks the parent of n, and adds a pointer to n'.

Figure 3.2 (top) shows the state of a B-link tree where node n has become full. We thus perform a half-split that moves its children $\{y_2, y_3\}$ to a new node n' and adds a link edge from n to n'. The key 5 in the gray box in n directs operations on keys $k \geq 5$ via the link edge to n'. The bottom figure shows the state after this half-split but before the complete-split when the pointer of n' will be added to r (shown using a dotted edge in Figure 3.2).

3.3 Abstracting Search Structures using Edgesets

The link technique is not restricted to B-trees: consider a hash table implemented as an array of pointers, where the ith entry refers to a bucket node that contains an array of keys $k_0, \ldots, k_l$ that all hash to i. When a node n gets full, it is locked, its keys are moved to a new node n' with twice the space, and n is linked to n'. Again, a separate operation locks the main array entry and updates it from n to n'. Thus, B-link trees and hash-link structures follow the same principles of forward pointer-based thread redirection.

While these data structures look completely different, the main operations of search, insert, and delete follow the same abstract algorithm. In both B-link trees and hash-link structures, there is some local rule by which operations are routed from one node to the next, and both introduce link edges when keys are moved to ensure that no other operation loses its way.

Reprising the discussion of §3.1, we view the state of a search structure abstractly as a mathematical graph. Each node in this graph can represent anything from two adjacent heap cells (in the case of a singly-linked list) to a collection of arrays and fields (in the case of a B-tree), and this mapping is determined by the specific implementation under consideration.

```
let create () =
  let r = allocRoot () in
  r

let rec traverse n k =
  lockNode n;
  match findNext n k with
  | None -> n
  | Some n' ->
      unlockNode n;
      traverse n' k
```

```
let rec cssOp ω r k =
  let n = traverse r k in
  match decisiveOp ω n k with
  | None -> unlockNode n;
      cssOp ω r k
  | Some res -> unlockNode n;
      res
```

Figure 3.3: The link template algorithm. The cssOp method is the main method, and represents the core search structure operations (search, insert, and delete) via the parameter ω. It uses an auxiliary method `traverse` that recursively traverses the search structure until it finds the node upon which to operate (the node containing k in its keyset, as described in §3.5). This template can be instantiated to the B-link tree algorithm by providing implementations of helper functions `findNext` and `decisiveOp`. `findNext` $n\,k$ returns Some n' if $k \in \mathsf{es}(n, n')$ and None if there exists no such n'. `decisiveOp` $n\,k$ performs the operation ω (either search, insert, or delete) on k at node n.

We then define the *edgeset* of an edge (n, n'), written $\mathsf{es}(n, n')$, to be the set of keys for which an operation on one of those keys arriving at a node n traverses (n, n').

The B-link tree in Figure 3.2 labels each edge with its edgeset; the edgeset of (n, y_1) is $[4, 5)$ and the edgeset of the link edge (y_0, y_1) is $[4, \infty)$. Note that 4 is in the edgeset of (y_0, y_1) even though an operation on 4 would not normally reach y_0. This is deliberate. In order to make edgeset a local quantity, we say $k \in \mathsf{es}(n, n')$ if an operation on k would traverse (n, n') assuming it somehow found itself at n.

In the hash table, assuming there exists a global root node, the edgeset from the root to the ith array entry is $\{k \mid h(k) = i\}$, i.e., all the key values for which a search would go to the node of the ith array entry. By contrast, the edgeset from an array entry to the bucket node is the set of all keys $\mathbb{K}$, as is the edgeset from a deleted bucket node to its replacement. The reason is that once we arrive at an array entry (or a deleted node), we follow the outgoing edge no matter which key we are looking for.

3.4 The Link Template

Figure 3.3 lists the link template algorithm [153] that uses edgesets to describe the algorithm used by all core operations for both B-link trees and hash tables in a uniform manner. The algorithm is described in the ML-like language that we use throughout the dissertation, and is described in more detail in §2.2. The algorithm assumes that an implementation provides certain primitives or helper functions that satisfy certain properties. For example, `findNext` is a helper function that finds the next node to visit given a current node n and an operation key k, by looking for an edge (n, n') with $k \in \mathsf{es}(n, n')$. For the B-link tree, `findNext` does a binary search on the keys in a node to find the appropriate pointer to follow.

For the hash table, when at the root, `findNext` returns the edge to the array element indexed by the hash of the key; when at a bucket node n, `findNext` returns the link edge if it exists and k is not in n. The function `cssOp` can be used to build implementations of all three search structure operations by implementing the helper function `decisiveOp` to perform the desired operation (read, add, or remove) of key k on the node n.

An operation on key k starts at the root r, and calls a function `traverse` on line 13 to find the node on which it should operate. `traverse` is a recursive function that works by following edges whose edgesets contain k (using the helper function `findNext` on line 7) until the operation reaches a node n with no outgoing edge having an edgeset containing k. Note that the operation locks a node only during the call to `findNext`, and holds no locks when moving between nodes. `traverse` terminates when `findNext` does not find any n' such that $k \in \mathsf{es}(n, n')$. In the B-link tree example, this corresponds to finding the appropriate leaf.

At this point, the thread performs the decisive operation on n (line 14). Note that for the link template we assume `decisiveOp` returns an optional Boolean value so that it can signal when it fails. If it is not possible for the decisive operation to be completed because, say, an insert operation encounters a full node, `decisiveOp` returns `None` and the algorithm unlocks n, gives up, and starts from the root again. Note that non-completion does not imply the existence of race conditions or null pointer exceptions. Our proofs show that such errors are impossible. If the decisive operation can be completed (i.e., it succeeds), then `decisiveOp` returns `Some` *res* and the algorithm unlocks n and returns *res*.

If we can verify this link template algorithm with a proof that is parameterized by the helper functions, then we can reuse the proof across diverse search structures.

3.5 From Edgesets to Keysets

In this section, we describe the high-level argument behind the correctness of single-copy structures. The crux of the argument relies on tying the *local view* of the thread with the *global* state of the search structure. For instance, when a thread executing `delete` for succeeds in deleting key k from a node, we must establish that k is deleted from the search structure. In other words, no additional copies of k were present in other nodes.

The edgeset framework helps the proof argument above by defining a *keyset* for each node. Intuitively, a keyset of a node, written as $\mathsf{ks}(n)$, represents all keys that node is responsible for. To be precise, if a key k belongs to $\mathsf{ks}(n)$, but k is not contained in n, then it will not be contained in any other node. Below, we provide the technical definition of $\mathsf{ks}(n)$.

We once again work on the abstract graph view of the search structure from §3.1. Let the union of edgesets leaving a node n be the *outset* $\mathsf{outset}(n)$. Suppose we denote the intersection of edgesets along the path from the root leading to a given node n as the *inset* of n, denoted $\mathsf{inset}(n)$. We can then define the *keyset* of n as:

$$\mathsf{ks}(n) \coloneqq \mathsf{inset}(n) \setminus \mathsf{outset}(n)$$

For example, if a node n in a binary search tree has key 7 (see Figure 3.1), the parent of n (which is also the root) has key 5, and n has a right child but no left child, then $\mathsf{ks}(n)$ is

the set of keys greater than 5 and less than or equal to 7, $\{k \mid 5 < k \leq 7\}$. Of course, the contents of n, namely 7, is in that set.

Sequential search structure algorithms will ensure the following two invariants:

1. For every node n, the contents $\mathsf{contents}(n)$ are a subset of the keyset $\mathsf{ks}(n)$.

2. The keysets of every pair of distinct nodes are disjoint.

We call these the *keyset invariants*.[1] In Iris, we encode these invariants via Iris's notion of ghost state, known as *resource algebra* (RA, in short). The algebraic properties of the resource algebra will ensure that these invariants are maintained, and the edgeset framework can be used to relate the ghost state to the concrete contents of the search structure. We provide a brief introduction to resource algebras in Section 4.4, and discuss the resource algebras specifically for keysets in Chapter 5.

Many common single-copy search structures maintain the keyset invariants. A consequence of these invariants is that a search for key k can examine the node n such that $k \in \mathsf{ks}(n)$ to determine whether k is anywhere in the search structure (because the keyset invariants implies that k cannot be anywhere else). That examination of n is called the *decisive operation* of the search. For many structures of interest, the decisive operations of insert, delete, and update of key k would also apply to the node n having k in its keyset. We revisit these ideas in Chapter 8.

3.6 The Flow Framework

The notion of keyset defined in the previous section is a global quantity defined over the entire graph. We give a brief introduction to the flow framework by explaining how the keyset quantity can be expressed as a node-local quantity by using flows.

The flow framework enables separation-logic-style reasoning about recursive functions on graphs. Certain restrictions apply. The function must be of the form $\mathit{fl} \colon N \to M$ where N is the set of nodes of the graph and $(M, +, 0)$ is a commutative cancellative monoid, called the *flow domain*. Further, fl must satisfy the *flow equation*:

$$\forall n \in N.\, \mathit{fl}(n) = \mathit{in}(n) + \sum_{n' \in N} e(n', n)(\mathit{fl}(n')) \qquad (\mathsf{FlowEqn})$$

Intuitively, this equation states that fl can be computed by assigning every node an initial value according to the *inflow function* $\mathit{in} \colon N \to M$ and then propagating these values along the edges of the graph using the *edge function* $e \colon N \times N \to M \to M$ to reach a fixpoint. At each node n, the values propagated from predecessor nodes n' are aggregated using the monoid operation $+$. A function fl that satisfies the flow equation is called a *flow* and a graph equipped with a flow is a *flow graph*. The flow framework then enables us to

[1] Note that we do not require the invariant that the union of keysets of all nodes cover the keyspace $\mathbb{K}$; as discussed in §5.2, this is only needed to prove termination.

reason compositionally about invariants of flow graphs expressed as node-local conditions that depend on a node's flow.

As an example, consider the quantity inset discussed in the previous section. Intuitively, $\mathsf{inset}(n)$ represents the set of keys which are contained in the path from the root node to n. Its formal definition is the following fixpoint equation

$$\forall n \in N.\ \mathsf{inset}(n) = in(n) \cup \bigcup_{n' \in N} \mathsf{es}(n', n) \cap \mathsf{inset}(n')$$

where $in(n) \coloneqq (n = r\ ?\ \mathbb{K} : \emptyset)$. The inset of a node n is thus $\mathbb{K}$ if n equals the root r, else the set of keys k that are in the inset of a predecessor n' such that $k \in \mathsf{es}(n', n)$.

Due to its fixpoint definition, the inset can be expressed as an instance of (FlowEqn) using sets and set operations, and edge functions that take the intersection with the appropriate edgeset. This means we can define a flow domain where the flow at each node is the inset of that node. Since keyset of a node is defined in terms of its inset, we have expressed keysets via the flow framework indirectly.

Once the inset is defined as a flow, we can use the notion of a *flow interface* to prove locally that an update to the graph does not change the flow of any nodes outside the modified region. The flow interface of a region consists of its outflow and inflow, maps that intuitively capture the contribution of this region to the flow of the rest of the world and the contribution of the outside world to this region's flow, respectively. If the interface of a modified region is preserved, then the framework guarantees that the flow of the rest of the graph is unchanged.

Technically, this kind of reasoning is enabled by the separation algebra structure of flow graphs (in particular the definition of flow graph composition), which extends the composition of partial graphs in standard separation logic so that the frame rule also preserves flow values of nodes in the frame. Instead of performing an explicit induction over the entire graph structure to prove that inset values continue to satisfy desired invariants, the necessary induction is hidden away inside the definition of flow graph composition (for more details see [100]).

The flow interfaces form a resource algebra in Iris, and hence, we can use them as ghost values in Iris proofs. We use them extensively in this dissertation in order to reason about global graph quantities such as keysets. This dissertation uses the formalization of the flow resource algebra from [97].

4 | Separation Logic

In this dissertation, we use *separation logic* to specify and verify concurrent data structures. Separation logic (SL), is an extension of Hoare logic [72] that is tailored to perform modular reasoning about programs that manipulate mutable resources. In other words, SL is a language that allows one to succinctly and modularly describe states of a program. Each sentence in this language is called a *proposition* (or *formula*, *assertion*), and describes a set of states. We say a state *satisfies* a proposition when the state is described by the proposition. Separation logic also gives us a set of proof rules that can be used to prove that states of interest (such as the set of resulting states after a program executes) satisfy a particular proposition.

There are many incarnations of SL, each tailored to reasoning about a particular class of programs. In this dissertation, we use *Iris*, a mechanized higher-order concurrent separation logic framework. The distinguishing feature of Iris is its generality: it is designed as a small set of core primitives and proof rules that can be used to encode a large variety of common constructs and techniques for reasoning about concurrent programs. In particular, Iris is easily extendable with user-defined resources via its ghost state mechanism, which we will describe in Section 4.4.

The price paid for this generality is that the core Iris logic is very abstract. To make this dissertation accessible to a wider audience, we present many of the derived features as though they are primitive Iris features, and avoid talking about the formal semantic model altogether. We refer the interested reader to a paper by Jung et al. [83] for a more detailed introduction to Iris, the Iris tutorial at POPL 2021 [22] for a gentle introduction, and to the documentation [74] for the full details. We discuss other SLs, as well as alternate approaches to verifying data structures, in Chapter 11.

4.1 Separation Logic by Example

Consider the following program expression that reads the value stored at heap location x into a variable ($v = !x$) and then writes $v + 1$ back into the location:

$$e_{\mathsf{inc}} \coloneqq \textbf{let}\ v = !x\ \textbf{in}\ x \leftarrow (v + 1)$$

Informally, e_{inc} is a program that increments the value stored at the heap location x.

We can formalize this specification in SL by using a *Hoare triple* [72], which is an assertion of the form $\{P\}\ e\ \{v.\, Q\}$, where P and Q are propositions. If $\{P\}\ e\ \{v.\, Q\}$ holds, then for every state σ that satisfies P we have (1) the program e does not reach an error state when run from σ (for example, by trying to read unallocated memory), and (2) that if e terminates then it returns some value v and the new state is some σ' that satisfies Q. We call P the *precondition* and Q the *postcondition* of e, and we write $\{P\}\ e\ \{Q\}$ in the case where Q does not mention the return value v.

Here is the desired specification of our example program e_{inc}:[1]

$$\{x \mapsto n\}\ e_{\mathsf{inc}}\ \{x \mapsto n+1\}$$

The precondition here uses a *points-to* predicate $x \mapsto n$, a primitive proposition asserting that the program state contains a heap cell at address x containing value n. The postcondition uses a similar points-to predicate, and, in simple words, the triple says: if the program e_{inc} is run on a state that contains a heap cell at address x with value n, then it results in a state where the cell x contains $n+1$. It also implicitly asserts that e_{inc} does not crash, by, e.g., attempting to dereference memory that is not allocated or divide by zero.

To prove that e_{inc} meets its specification, we use the proof rules for Hoare triples shown in Figure 4.1. In the figure, we write $e[x \rightarrowtail v]$ to denote the expression e after substituting all occurrences of the variable x with the term v. The function $\mathit{free}(Q)$ computes free variables in Q. The rules for Hoare triples can be used in the same manner as proof rules in classical proof systems. We next discuss the core feature of the separation logic, the *separating conjunction*.

Consider the program expression below that writes to two heap cells in two parallel threads:

$$e_{\mathsf{par}} \coloneqq (x \leftarrow 1 \parallel y \leftarrow 2)$$

In the above program we use the commonly used parallel composition syntax $(e_1 \parallel e_2)$ which denotes forking two threads that run e_1 and e_2 respectively and waiting until they complete. Parallel composition can be defined in terms of **`fork`** and **`CAS`**. Note that this program's behavior can be described precisely only if x and y are *distinct* heap locations; if they are equal, then the two threads would race to write to the same heap cell. As a result, the two writes on x could happen in either order, and the final value stored at x cannot be determined.

To specify this program, we use the separating conjunction $*$. Unlike standard conjunction $\wedge$, separating conjunction conjoins two propositions that describe *disjoint* portions of program state. For instance, $x \mapsto \_ \wedge y \mapsto \_$ asserts that x is a heap cell and y is a heap cell (but they could be the same heap cell), while $x \mapsto \_ \ * \ y \mapsto \_$ asserts that x is a heap cell *and separately* y is a heap cell.

Formally, a state σ satisfies $P * Q$ if it can be broken up into two disjoint states $\sigma = \sigma_1 \circledcirc \sigma_2$ such that σ_1 satisfies P and σ_2 satisfies Q. When P and Q are propositions denoting heaps, this means that they talk about disjoint regions of the heap, i.e. that they do not have any

[1] In all such triples that we use as specifications in this dissertation, free variables such as n are implicitly universally quantified.

$$\textsc{hoare-ret}\ \{\mathsf{True}\}\ w\ \{v.\, v = w\} \qquad \textsc{hoare-false}\ \{\mathsf{False}\}\ e\ \{v.\, P\}$$

$$\textsc{hoare-alloc}\ \{\mathsf{True}\}\ \mathbf{ref}(v)\ \{\ell.\, \ell \mapsto v\} \qquad \textsc{hoare-load}\ \{\ell \mapsto v\}\ !v\ \{w.\, \ell \mapsto v\ *\ w = v\}$$

$$\textsc{hoare-store}\ \{\ell \mapsto v\}\ \ell \leftarrow w\ \{\ell \mapsto w\}$$

$$\textsc{hoare-cas-suc}\ \{\ell \mapsto v\}\ \mathbf{CAS}(\ell, v, w)\ \{b.\, b = \mathit{true}\ *\ \ell \mapsto w\}$$

$$\textsc{hoare-cas-fail}\ \frac{v \neq v'}{\{\ell \mapsto v\}\ \mathbf{CAS}(\ell, v', w)\ \{b.\, b = \mathit{false}\ *\ \ell \mapsto v\}}$$

$$\textsc{hoare-lam}\ \frac{\{P\}\ e[x \rightarrowtail v]\ \{w.\, Q\}}{\{P\}\ (\lambda x.\, e)\ v\ \{w.\, Q\}} \qquad \textsc{hoare-fork}\ \frac{\{P\}\ e\ \{\mathsf{True}\}}{\{P\}\ \mathbf{fork}\ \{e\}\ \{\mathsf{True}\}}$$

$$\textsc{hoare-let}\ \frac{\{P\}\ e_1\ \{w.\, R\} \qquad \forall w.\, \{R\}\ e_2[x \rightarrowtail w]\ \{v.\, Q\}}{\{P\}\ \mathbf{let}\ x\ \texttt{=}\ e_1\ \mathbf{in}\ e_2\ \{v.\, Q\}}$$

$$\textsc{hoare-csq}\ \frac{P \Rrightarrow P' \qquad \{P'\}\ e\ \{v.\, Q'\} \qquad \forall v.\, Q' \Rrightarrow Q}{\{P\}\ e\ \{v.\, Q\}}$$

$$\textsc{hoare-frame}\ \frac{\{P\}\ e\ \{v.\, Q\}}{\{P * R\}\ e\ \{v.\, Q * R\}} \qquad \textsc{hoare-disj}\ \frac{\{P\}\ e\ \{v.\, R\} \qquad \{Q\}\ e\ \{v.\, R\}}{\{P \vee Q\}\ e\ \{v.\, R\}}$$

$$\textsc{hoare-exist}\ \frac{\forall x.\, \{P\}\ e\ \{v.\, Q\} \qquad x \notin \mathit{free}(Q)}{\{\exists x.\, P\}\ e\ \{v.\, Q\}}$$

Figure 4.1: Proof rules for establishing Hoare triples.

heap addresses in common. In particular, this means that $x \mapsto \_ \ * \ y \mapsto \_$ implies that $x \neq y$, while the proposition $x \mapsto y \ * \ x \mapsto z$ is unsatisfiable as it is not possible to split a heap into two disjoint portions both of which contain the same address x.

We can thus specify e_{par} as follows:

$$\{x \mapsto \_ \ * \ y \mapsto \_\} \ (x \leftarrow 1 \parallel y \leftarrow 2) \ \{x \mapsto 1 \ * \ y \mapsto 2\}$$

To prove this specification, we use the following parallel composition rule (which can be derived from HOARE-FORK):

$$\textsc{hoare-par} \ \frac{\{P_1\} \ e_1 \ \{Q_1\} \qquad \{P_2\} \ e_2 \ \{Q_2\}}{\{P_1 * P_2\} \ (e_1 \parallel e_2) \ \{Q_1 * Q_2\}}$$

HOARE-PAR tells us that executing two threads in parallel is safe if both threads operate on disjoint portions of the state.

Every proposition in SL can be thought of as a *resource*. It may be natural to think of the predicate $x \mapsto n$ as a resource, as it describes a heap cell or a part of the program state. In the concurrent setting we can think of each thread *owning* the resources in its precondition. HOARE-PAR then tells us that a thread can subdivide its resources and distribute them among the subthreads that it spawns. Primitive resources such as points-to predicates $x \mapsto \_$ are not subdividable, so there is no way to use HOARE-PAR alone to prove that the program $(x \leftarrow 1 \parallel x \leftarrow 2)$ is safe. In such situations, we think of resources such as $x \mapsto \_$ as being exclusively owned by a single thread.

Given this, it may seem counter-intuitive that the rules HOARE-CAS-SUC and HOARE-CAS-FAIL require exclusive ownership of the location ℓ despite the fact that the main application of CAS is fine-grained concurrent programs where multiple threads access the same heap cell. However, we will see in §4.5.1 proof rules for atomic commands such as CAS (compare-and-set) that allow one to take exclusive ownership of shared resources for the duration of an atomic command (since no other thread can interfere).

4.2 Iris Propositions

Iris propositions describe the *resources* owned by a thread. These resources can be part of a concrete program state, for example, a set of heap cells, which captures the situations in which a thread has exclusive ownership over these cells (because, say, it has locked them). To reason about fine-grained concurrency and more complex concurrency patterns, these resources can also capture shared ownership and partial knowledge of shared program state. In order to build intuition, we focus on the simple case where propositions describe concrete program states, in the form of subsets of the heap, and defer the discussion of advanced resources to Section 4.4. For a formal definition of program states and the satisfaction relation in Iris, see the Iris documentation [74].

The grammar of the subset of Iris propositions that we use throughout the dissertation is shown in Figure 4.2, and includes the following constructs:

$$
\begin{aligned}
P, Q, R \coloneqq\ & \mathsf{True} \mid \mathsf{False} \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \\
& \mid \exists x.\ P \mid \forall x.\ P \\
& \mid x \mapsto v \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \mathop{\Large *}_{x \in X} P \\
& \mid \boxed{P}^{\mathcal{N}} \mid \triangleright P \mid \boxed{a}^{\gamma} \mid P \Rrightarrow Q \\
& \mid \{P\}\ e\ \{v.\, Q\} \mid \langle x.\ P \rangle\ e\ \langle v.\, Q \rangle \mid \mathsf{AU}_{x.P,Q}(\Phi)
\end{aligned}
$$

Figure 4.2: The grammar of Iris propositions used in this dissertation.

- The first line consists of standard propositional constructs: the propositions True and False[2], then conjunction, disjunction, and implication.

- The second line introduces quantification. Note that since Iris is a *higher-order* logic, x can range over any type, including that of propositions and (higher-order) predicates.

- We have already seen the points-to proposition and separating conjunction on the third line. Note that Iris is an affine logic, which means $P * Q \vdash P$[3] holds for any propositions P and Q. In particular, this implies that if a state σ satisfies $x \mapsto v$, then σ can possibly contain more than just the heap cell at address x.

- We also have an iterated version of separating conjunction: $\mathop{\Large *}_{x \in X} P$, where the bound variable x ranges over a finite set X. For example, $\mathop{\Large *}_{x \in X} x \mapsto 13$ denotes states that contain at least the set of heap cells with addresses in X all of whom store the value 13.

- The *separating implication* connective $\mathrel{-\!\!*}$, also known as the *magic wand*, is defined as: a state σ satisfies $P \mathrel{-\!\!*} Q$ if for every state σ_1 disjoint from σ that satisfies P, the combined state $\sigma \circledcirc \sigma_1$ satisfies Q. The best way to understand $\mathrel{-\!\!*}$ is to think of $*$ and $\mathrel{-\!\!*}$ as separation logic analogues of $\wedge$ and $\Rightarrow$ from first-order logic. For instance, $P * Q$ means you have both P *and* Q (and that they are disjoint, or more generally, composable). Similarly, $P \mathrel{-\!\!*} Q$ describes a state such that if you conjoin it with a state satisfying P, then you get a state satisfying Q.

  This property, $P * (P \mathrel{-\!\!*} Q) \vdash Q$, is the SL analogue of *modus ponens* in first-order logic ($P \wedge (P \Rightarrow Q) \vdash Q$).

- The fourth line contains the invariant proposition $\boxed{P}^{\mathcal{N}}$ and later modality $\triangleright P$ (both explained in §4.3), as well as the ghost state proposition $\boxed{a}^{\gamma}$ and the view shift $P \Rrightarrow Q$ (both explained in Section 4.4).

[2]The propositions True and False are semantically distinct from the Boolean constants *true* and *false* which are values of our programming language. Intuitively, propositions can be thought of as describing sets of program states, where a program state maps program variables to values.

[3]$\vdash$ is the *provable entailment* relation; $P \vdash Q$ means that for any state in which P holds, Q also holds.

- Finally, the last line contains Hoare and atomic triples (explained in §4.5).

4.3 Invariants

Consider the following concurrent program [4]:

$$e_{\mathsf{even}} \coloneqq \mathbf{let}\ x\ =\ \mathbf{ref}(0)\ \mathbf{in}\ (x \leftarrow \mathbf{FAA}(x,2)\ \parallel\ x \leftarrow \mathbf{FAA}(x,2));\ !x$$

And say we want to prove the specification

$$\{\mathsf{True}\}\ e_{\mathsf{even}}\ \{v.\ \mathsf{Even}(v)\},$$

where $\mathsf{Even}(n) \coloneqq (n\%2 = 0)$ says that the program returns an even number. Unlike e_{par}, this program manipulates the *same* heap location in two parallel threads. This means we cannot use the parallel composition rule HOARE-PAR, as there is no way we can duplicate the heap cell $x \mapsto 0$ in order to give it to both of the threads:

Nevertheless, the program is safe, and no matter which thread writes to x first, the returned value will be even. The program does not crash because the threads are using atomic instructions to store a value at x, so despite the fact that x is a shared heap cell, both stores are safe. Moreover, since both threads increment by an even number to x, the final result is also even. We need a proof mechanism that allows us to share state between threads and formalize the above reasoning.

The solution is to use invariants. An *invariant* in Iris is a proposition of the form $\boxed{P}^{\mathcal{N}}$, where P is an arbitrary Iris proposition. Invariants provide a mechanism to reason about ownership of resources describing shared state that can be concurrently accessed by many threads. Intuitively, an invariant is a property that, once established, will remain true forever. It is therefore a duplicable resource and can be freely shared with any thread.

However, in order to ensure that the invariant indeed remains valid once it has been established, Iris's proof rules for invariants impose restrictions on how the resources contained in an invariant can be accessed and manipulated. At any point in time, a thread can *open* an invariant $\boxed{P}^{\mathcal{N}}$ and gain ownership of the contained resources P. These resources can then be used in the proof of a single atomic step of the thread's execution. After the thread has performed an atomic step with an open invariant, the invariant must be *closed*, which amounts to proving that P has been reestablished. Otherwise, the proof cannot succeed.

The proof rules for invariants are given in Figure 4.3. The $\triangleright$ symbol is called the "later" modality, and is needed to ensure soundness in complex cases when Hoare triples or invariants themselves are stored inside invariants; we can ignore it for now. INV-ALLOC is a rule that allows us to take a resource R that we own and turn it into an invariant. In most proofs, this signifies the point at which some local state is shared with other threads for the first time. INV-DUP allows invariants to be freely duplicated; this will allow us to share them among

[4] This example is adapted from [22].

$$\text{INV-ALLOC}\ \frac{\left\{\boxed{R}^{\mathcal{N}} * P\right\}\ e\ \{Q\}_{\mathcal{E}}}{\left\{\triangleright R * P\right\}\ e\ \{Q\}} \qquad \text{INV-DUP}\ \boxed{R}^{\mathcal{N}} \vdash \boxed{R}^{\mathcal{N}} * \boxed{R}^{\mathcal{N}}$$

$$\text{INV-OPEN}\ \frac{\left\{\triangleright R * P\right\}\ e\ \left\{\triangleright R * Q\right\}_{\mathcal{E}} \qquad e\ \text{atomic}}{\left\{\boxed{R}^{\mathcal{N}} * P\right\}\ e\ \left\{\boxed{R}^{\mathcal{N}} * Q\right\}_{\mathcal{E} \uplus \mathcal{N}}}$$

Figure 4.3: Proof rules for Iris invariants.

threads. Finally, INV-OPEN allows us to open an invariant and gain ownership of its contents for the duration of an atomic step.

The $\mathcal{N}$ in $\boxed{P}^{\mathcal{N}}$ refers to the *namespace* of the invariant. Namespaces are part of the mechanism used in Iris to keep track of invariants that are currently open and need to be closed before the next atomic step. This is necessary to avoid issues of re-entrancy in case of nested invariants, which would lead to logical inconsistencies. The set of namespaces that one is allowed to open is kept as an annotation (subscript) to the Hoare triple in question; for instance, note that INV-OPEN ensures that the invariant named $\mathcal{N}$ cannot be opened again. Since most proofs in this dissertation use only a single invariant, we also omit these namespace annotations from Hoare triples.

We can now prove that e_{even} returns an even number, using an invariant as follows:

$\{\mathsf{True}\}$
let x = **ref**(0) **in**
$\{x \mapsto 0\}$
(* Using INV-ALLOC *)
$\left\{\boxed{\exists n.\, x \mapsto n\ *\ \mathsf{Even}(n)}\right\}$
(* Using INV-DUP *)
$\left\{\begin{array}{l}\boxed{\exists n.\, x \mapsto n\ *\ \mathsf{Even}(n)} \\ *\ \boxed{\exists n.\, x \mapsto n\ *\ \mathsf{Even}(n)}\end{array}\right\}$
(* Using HOARE-PAR *)
$(x \leftarrow \mathbf{FAA}(x, 2)\ \|\ x \leftarrow \mathbf{FAA}(x, 2))$
$\left\{\begin{array}{l}\boxed{\exists n.\, x \mapsto n\ *\ \mathsf{Even}(n)} \\ *\ \boxed{\exists n.\, x \mapsto n\ *\ \mathsf{Even}(n)}\end{array}\right\}$
$\left\{\boxed{\exists n.\, x \mapsto n\ *\ \mathsf{Even}(n)}\right\}$
$!x$
$\{v.\, x \mapsto v\ *\ \mathsf{Even}(v)\}$

$\left\{\boxed{\exists n.\, x \mapsto n\ *\ \mathsf{Even}(n)}\right\}$
(* Using INV-OPEN *)
$\{x \mapsto n'\ *\ \mathsf{Even}(n')\}$
$x \leftarrow \mathbf{FAA}(x, 2)$
$\{x \mapsto n' + 2\ *\ \mathsf{Even}(n' + 2)\}$
(* Using INV-CLOSE *)
$\left\{\boxed{\exists n.\, x \mapsto n\ *\ \mathsf{Even}(n)}\right\}$

Here we use INV-ALLOC at the beginning to transfer the newly-created heap cell $x \mapsto 0$ into an invariant[5]. We then use INV-DUP to duplicate it, and then share it among the two

[5]We omit the namespace of the invariant when it is irrelevant to the context.

threads using HOARE-PAR. Each thread then uses INV-OPEN to open the invariant around the atomic store operation, after which we show that the store preserves the invariant that x contains an even number in each case. After the threads join, we once again open the invariant to read the contents of x, and the invariant tells us that whatever the value is, it is even.[6]

4.4 Ghost State

Ghost state, originally called auxiliary variables [136], is a formal technique where the prover adds state (variables or resources) to a program that capture knowledge about the history of a computation not present in the state of the original program in order to verify it. As long as the added ghost state, and the ghost commands that modify it, have no effect on the run-time behavior of the program, then a so-called *erasure* theorem states that a proof of the augmented program can be transformed into a proof of the program with all ghost state removed (i.e., the original program). In Iris, ghost state is purely logical and ghost commands are represented as proof rules, which gives us such an erasure theorem for free. The ghost state concept has shown itself to be an invaluable tool in the verifier's toolbox, and has been used to encode many common reasoning techniques including permissions, tokens, capabilities, and protocols.

Our treatment of ghost state in Iris is restricted to RAs that have units, as that is sufficient for the proofs in this dissertation. Iris supports more general and powerful kinds of ghost state, so-called *cameras*, and we refer interested readers to the introduction to cameras by Jung et al. [83]. We also motivate ghost state in this chapter as a means to verifying template search structures, but ghost state has a wide variety of applications in verifying many kinds of algorithms and properties. The Iris tutorial at POPL 2021 [22] contains motivating examples that demonstrate other uses of ghost state.

In this section, we start by motivating the need for ghost state using the e_{even} as an example. We then see how Iris supports ghost state via the notion of resource algebras (RAs), and use a fractional RA to verify a stronger specification for e_{even}.

4.4.1 Motivation

In Section 4.3, we proved the following specification for e_{even}:

$$\{\mathsf{True}\}\ e_{\mathsf{even}}\ \{v.\, \mathsf{Even}(v)\}$$

However, the above specification does not fully capture the behavior of e_{even}. A stronger specification for e_{even} would establish that the value of x read at the end will be 4. That is, the following Hoare triple:

$$\{\mathsf{True}\}\ e_{\mathsf{even}}\ \{v.\, v = 4\}$$

[6] We throw away the invariant at the end because we no longer need it, which we can do because Iris is affine.

The invariant used earlier, $\boxed{\exists n.\, x \mapsto n \;*\; \mathsf{Even}(n)}$, is not enough to prove the above specification. The reason is the existential quantification over the value stored at x. In order to prove the stronger specification, we need to encode the knowledge that there are exactly two threads that increment the value at x by 2. We introduce ghost locations γ_1 and γ_2 to store the contribution to the value at x by each thread. Additionally, the knowledge about the contributions must be shared between the invariant and the thread.

Iris expresses ownership of ghost state by the proposition $\boxed{a}^{\gamma}$ which asserts ownership of a piece a of the ghost location γ. The values a stored in ghost state belong to a resource algebra (RA, defined formally below). For example, a *fractional* RA over integers consists of elements of the form (q, n), where $q \in (0, 1]$ is a rational number and n is an integer, as well as a unit element ε and a special element $\lightning$. As we will see, a fractional ghost location can be split and shared among many threads to permit shared reads, but threads can only write to the ghost location if they own the full ghost location. This means that they have the following properties:

$$\mathsf{True} \Rrightarrow \boxed{(1, n)}^{\gamma} \tag{4.1}$$

$$\boxed{(q, n)}^{\gamma} \;*\; q = q_1 + q_2 \;\Rrightarrow\; \boxed{(q_1, n)}^{\gamma} \;*\; \boxed{(q_2, n)}^{\gamma} \tag{4.2}$$

$$\boxed{(q_1, n_1)}^{\gamma} \;*\; \boxed{(q_2, n_2)}^{\gamma} \;\vdash\; n_1 = n_2 \tag{4.3}$$

$$\boxed{(1, n)}^{\gamma} \Rrightarrow \boxed{(1, n')}^{\gamma} \tag{4.4}$$

Here, the first property allows creating a new ghost location with full permission. The second property tells us that if we own $\boxed{(q, n)}^{\gamma}$ then we can split it into smaller fractions (via a ghost update $\Rrightarrow$, explained below). The ghost update implicitly assumes $q_1, q_2 \in (0, 1]$. The third property tells us that any two fractional states must agree on the set of keys. The final property allows changing the value at the ghost location if full permission is owned.

With the fractional RA, we can now prove the stronger specification for e_{even} via the following invariant:

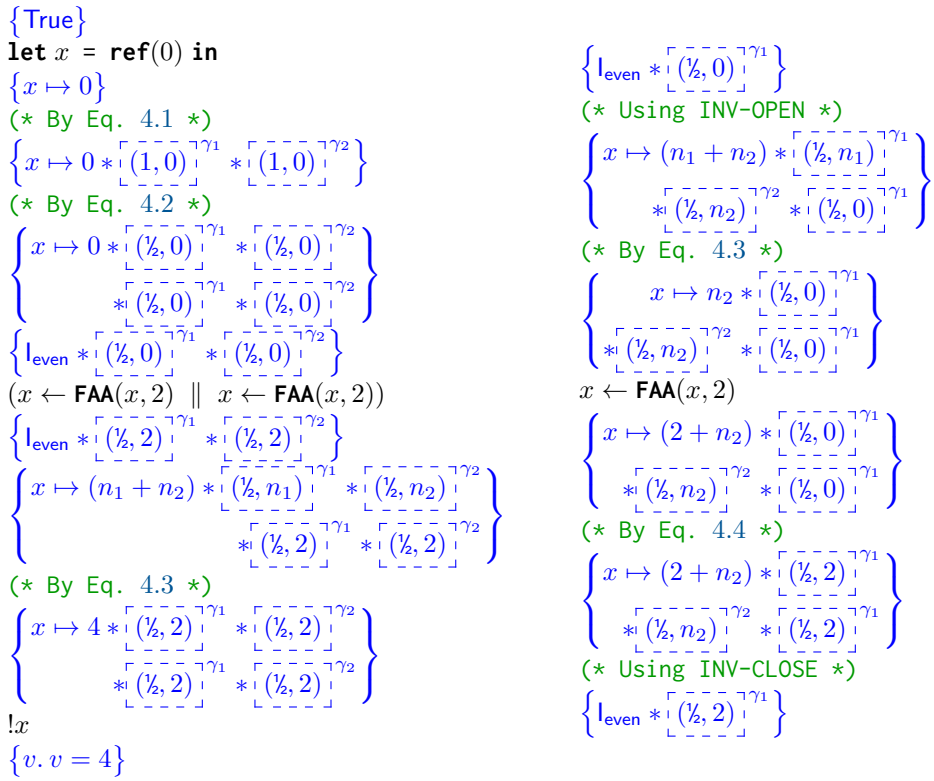
$$\mathsf{I}_{\mathsf{even}} \coloneqq \exists\, n_1\, n_2,\; x \mapsto (n_1 + n_2) * \boxed{(\tfrac{1}{2}, n_1)}^{\gamma_1} * \boxed{(\tfrac{1}{2}, n_2)}^{\gamma_2}$$

Consider the proof for the stronger specification for e_{even} below:

A *resource algebra* is a tuple $(M, \mathcal{V}\colon M \to \mathit{Prop}, (\cdot)\colon M \times M \to M, \varepsilon \in M)$ satisfying:

$$\forall a, b, c.\ (a \cdot b) \cdot c = a \cdot (b \cdot c) \qquad \text{(RA-ASSOC)}$$
$$\forall a, b.\ a \cdot b = b \cdot a \qquad \text{(RA-COMM)}$$
$$\forall a.\ \varepsilon \cdot a = a \qquad \text{(RA-ID)}$$
$$\mathcal{V}(\varepsilon) \qquad \text{(RA-VALID-ID)}$$
$$\forall a, b.\ \mathcal{V}(a \cdot b) \Rightarrow \mathcal{V}(a) \qquad \text{(RA-VALID-OP)}$$

Figure 4.4: The definition of a (unital) resource algebra (RA).



The ghost locations are initialized with value 0 as it denotes the contribution to each thread initially. The invariant $\mathsf{I}_{\mathsf{even}}$ is allocated by splitting the ownership over both ghost locations into halves. The right hand side shows the proof of one thread updating its contribution from intial value 0 to 2. The proof solely relies on the properties assumed for the fractional RA. Coming back to the proof on the left, we can establish that each thread has contributed 2 to the value of x, resulting in the final value of 4.

$$\textsc{ghost-alloc}\ \frac{\mathcal{V}(a)}{\mathsf{True} \Rrightarrow \exists \gamma. \boxed{a}^{\gamma}} \qquad \textsc{ghost-op}\ \boxed{a \cdot b}^{\gamma} \Leftrightarrow \boxed{a}^{\gamma} * \boxed{b}^{\gamma}$$

$$\textsc{ghost-valid}\ \boxed{a}^{\gamma} \Rrightarrow \mathcal{V}(a)$$

$$\textsc{ghost-update}\ \frac{a \rightsquigarrow B}{\boxed{a}^{\gamma} \Rrightarrow \exists b \in B. \boxed{b}^{\gamma}} \qquad \textsc{vs-trans}\ \frac{P \Rrightarrow Q \qquad Q \Rrightarrow R}{P \Rrightarrow R}$$

$$\textsc{vs-frame}\ \frac{P \Rrightarrow Q}{P * R \Rrightarrow Q * R}$$

Figure 4.5: Proof rules for manipulating ghost resources and view shifts.

4.4.2 Resource Algebras

Formally, a resource algebra (RA) consists of a set M, a *validity* predicate $\mathcal{V}(-)$, and a binary operation $(\cdot)\colon M \times M \to M$ that satisfy the axioms in Figure 4.4 (*Prop* is the type of propositions of the meta-logic (e.g., Coq)). RAs are a generalization of the partial commutative monoid (PCM) algebra commonly used by separation logics.[7]

The two important mechanisms for using ghost state are: (1) a ghost location can be split and combined using the rule: $\boxed{a}^{\gamma} * \boxed{b}^{\gamma} \dashv\vdash \boxed{a \cdot b}^{\gamma}$; and (2) at any point, if a thread owns a resource $\boxed{c}^{\gamma}$, then the value c is valid, i.e. $\mathcal{V}(c)$. This means that an RA's composition operator $\cdot$ must be associative and commutative.[8] The axiom RA-VALID-OP disallows taking an invalid element and composing it with another element to make it valid; since Iris maintains an invariant that the composition of all values in a ghost location is valid, this axiom implies that any sub-resource in that location is also valid. RA-ID makes ε an identity or unit element with respect to composition, and RA-VALID-ID says the unit must be valid.

One can think of the ghost state proposition $\boxed{a}^{\gamma}$ as the ghost analogue of the points-to predicate $x \mapsto v$ that asserts that the (real) location x contains value v.[9] However, $\boxed{a}^{\gamma}$ asserts only that γ contains a value *one of whose parts* is a (as we saw with the fractional ghost state example above). This means ghost state can be split and combined according to

[7] Iris actually uses *cameras* as the structure underlying resources, but as we do not use higher-order resources (i.e. state which can embed propositions) in this dissertation, we restrict our attention to resource algebras, a stronger, but simpler, structure. Furthermore, RAs technically have a *core* function $|-|$ that maps each element to a (potentially different) unit, but since all the RAs we use have a gobal unit (i.e., $|\_| = \varepsilon$), we omit the core function in our presentation and restrict our attention to unital RAs.

[8] Readers familiar with separation algebras will notice that the composition operator is not partial; cases where composition used to be undefined can be encoded by sending them to an invalid element.

[9] In fact, Iris's core logic makes no distinction between ghost state and non-ghost state – the heap is represented using special ghost state, and the points-to predicate $x \mapsto v$ is defined using the ghost location predicate $\boxed{a}^{\gamma}$. However, we continue to use the $\mapsto$ symbol for expressing constraints on heap locations as it is widely-used and will be familiar to readers who have studied SL before.

the composition operator of the underlying RA.

Example 1 *Given a set S, we define the fractional RA over values of S as:*

$$M \coloneqq (q, s) \in (\mathbb{Q} \cap (0, 1]) \times S \mid \varepsilon \mid \lightning \qquad \mathcal{V}(a) \coloneqq a \neq \lightning \qquad \varepsilon \cdot a \coloneqq a \cdot \varepsilon \coloneqq a$$

$$(q_1, s_1) \cdot (q_2, s_2) \coloneqq \begin{cases} (q_1 + q_2, s_1) & \text{if } q_1 + q_2 \leq 1 \land s_1 = s_2 \\ \lightning & \text{otherwise} \end{cases} \qquad \lightning \cdot \_ \coloneqq \_ \cdot \lightning \coloneqq \lightning$$

The fractional RA we used above in the proof of e_{even} can be obtained by instantiating the set S in Example 1 with the set of all integers. The definition of composition and validity in the fractional RA ensure that elements must agree on their second element and have compatible fractions. This implies that ghost locations containing elements of the fractional RA enjoy the properties (4.2) and (4.3).

Figure 4.5 provides proof rules for manipulating ghost resources. The rule GHOST-ALLOC is used to allocate a new ghost resource. The rule restricts the allocation to only those resources that are valid. The remaining rules dictate how a ghost resource can be updated using the view shift modality $\Rrightarrow$, which we explain next through the example of the fractional RA.

Ghost state by itself is not very useful unless it can be updated. However, unlike physical state, which can be modified at any point to any value, ghost state updates are restricted since Iris maintains the invariant that the composition of all the pieces of ghost state at a particular location is valid (as given by $\mathcal{V}$). Iris allows only *frame-preserving updates* $a \rightsquigarrow b$, defined below.

Definition 2 *A* frame-preserving update *is a relation between an element $a \in M$ and a set $B \subseteq M$, written $a \rightsquigarrow B$, such that*

$$\forall a_{\mathrm{f}} \in M.\ \mathcal{V}(a \cdot a_{\mathrm{f}}) \Rightarrow \exists b \in B.\ \mathcal{V}(b \cdot a_{\mathrm{f}}).$$

We write $a \rightsquigarrow b$ if $a \rightsquigarrow \{b\}$.

Intuitively, $a \rightsquigarrow b$ says that every *frame* a_{f} that is compatible with a should also be compatible with b. Thus, changing a thread's fragment of the ghost state from a to some b will not invalidate assumptions about a_{f} made by any other thread. The fractional RA has the following frame-preserving update:

$$\textsc{frac-upd}\ (1, s) \rightsquigarrow (1, s')$$

Note that the element $(1, s)$ has no frame (no non-unit element can compose with it), thus the frame-preserving update condition holds trivially.

This allows us to change the value stored at a ghost location holding a fractional RA value as long as we own all the pieces of that location. Correspondingly, we also note that there are no frame-preserving updates from (q, s) to any (q, s') when $q < 1$, which means no thread can change the value s unless that thread holds all the fragments (i.e., $q = 1$).

The rule GHOST-UPDATE in Figure 4.5 lifts frame-preserving updates on RA elements to ghost updates on Iris propositions, which are captured by Iris's view shift modality $\Rrightarrow$. The intuitive meaning of $P \Rrightarrow Q$ is that if we have the resource P, then we can perform one or more frame-preserving updates in order to transform the resource P to Q. Alternatively, we can regard $P \Rrightarrow Q$ as a Hoare triple, with P as precondition and Q as postcondition, but no program code, as P must be transformed to Q solely by manipulating ghost resources. The rule VS-TRANS enables combining multiple view shifts into one, while VS-FRAME allows *framing* out resources from the update, similar to Hoare triples. The rule HOARE-CSQ (introduced in Section 4.1) allows one to use view shifts to perform ghost updates on the pre- and postcondition of a Hoare triple.

Over the course of this dissertation, we will encounter RAs in many different contexts. In subsequent chapters, we define and explain them on as needed basis.

4.5 Atomic Triples

In this section we introduce the concept of *logical atomicity*, which is used to specify programs that execute in multiple atomic steps but whose effect appears to take place in a single point in time. This concept will be useful for specifying concurrent search structures in a way that can be used to verify concurrent client programs.

We begin by explaining why Hoare triples are not sufficient for specifying concurrent data structures. For example, consider a concurrent search structure with `cssOp` function (for concurrent search structure operation) as its operation (subsuming all the core search structure operations via parameter `op`). Recall that search structures are data structures implementing a mathematical set data type, so `op` is either search, insert, or delete. Suppose we tried to specify the behavior of a concurrent search structure using a Hoare triple as follows :

$$\{\mathsf{CSS}(r, C)\}\ \texttt{cssOp op}\ r\ k\ \{res.\ \mathsf{CSS}(r, C')\ *\ \Psi_{\mathsf{op}}(k, C, C', res)\}, \tag{4.5}$$

where $\mathsf{CSS}(r, C)$ (for concurrent search structure) is a predicate describing a search structure at location r with contents C. While we might be able to prove that a concurrent search structure satisfies this specification, this specification will not be helpful when reasoning about concurrent programs that use the concurrent search structure.

To see why, consider the client program:

$$e_{\mathsf{client}} \coloneqq (\texttt{cssOp insert}\ r\ k_1 \quad \| \quad \texttt{cssOp insert}\ r\ k_2)$$

This is a program that manipulates a concurrent search structure rooted at r by calling the insert operation on two keys k_1 and k_2 on two parallel threads.

A simple specification that captures the fact that e_{client} is memory-safe in a concurrent context is:

$$\{\mathsf{CSS}(r, C)\}\ e_{\mathsf{client}}\ \{\mathsf{True}\}$$

One might try to prove this specification the same way we proved that e_{even} returned an even number, by creating a new invariant containing $\mathsf{CSS}(r, C)$ and sharing it among the

two threads. However, there is a catch: we cannot use INV-OPEN to open the invariant when we need to get $\mathsf{CSS}(r, C)$ to satisfy the precondition of `cssOp` in (4.5). The reason is that INV-OPEN can be applied only to programs that are *physically atomic*, i.e. that they execute in a single machine instruction (e.g., $!x$, or a **CAS**). Here, on the other hand, we need to reason about `cssOp`, which for most realistic search structures executes in multiple physical steps.

On the other hand, a good concurrent search structure is designed to be used in precisely such conditions, by using locks or other concurrent protocols to ensure that concurrent invocations of `cssOp` from multiple threads is safe. We thus need a stronger specification for `cssOp`, one that captures the fact that in concurrent settings `cssOp` *behaves* as though it is atomic.

We can specify the concurrent behavior of such programs using *atomic triples* [33, 85, 86]. An atomic triple $\langle\, x.\ P \,\rangle\ e\ \langle\, v.\ Q \,\rangle$ is made up of the precondition P (that can refer to the *pseudo-quantified* variable x, as explained below), return value v, postcondition Q (that can refer to v and x), and a program e. Such a triple means that e, despite executing in potentially many atomic steps, appears to operate atomically on the shared state and transforms it from a state satisfying P to one satisfying Q.

Atomic triples are strongly related to the well-known *linearizability* [49, 70] criterion for concurrent algorithms. Intuitively, there is a point in time during the course of the execution of e, known as the *linearization point*, where e updates P to Q. For the example of an insert operation on a search structure, this will be the point of time when the inserted value is visible to other threads. Linearizability requires that a concurrent set of operations produces the same final state and returns the same values as a sequential execution of the operations where the ordering is the order of the linearization points. In other literature [11], linearizability is known as order-preserving serializability.

The specification we want to prove for concurrent search structures is the following:

$$\langle\, C.\ \mathsf{CSS}(r, C) \,\rangle\ \texttt{cssOp op}\ r\ k\ \langle\, res.\ \mathsf{CSS}(r, C')\ *\ \Psi_{\texttt{op}}(k, C, C', res) \,\rangle \tag{4.6}$$

The binder on C in the precondition is a special *pseudo-quantifier* that captures the fact that during the execution of `op`, the value of C can change (e.g., by concurrent operations). At the linearization point however, `cssOp` changes $\mathsf{CSS}(r, C)$ to $\mathsf{CSS}(r, C')$ in an atomic step. The new set of keys C' and the eventual return value *res* satisfy the predicate $\Psi_{\texttt{op}}(k, C, C', res)$. Note that the C in the postcondition is bound in the precondition, i.e. to the contents *just before* the linearization point. The goal is that clients of the search structure can pretend that they are using a serial or sequential implementation with specification $\Psi_{\texttt{op}}$.

We call operations that satisfy atomic triples as being *logically atomic*. Coming back to our motivating example e_{client}, once we have proved that `cssOp` is logically atomic, we can use the following rule to open an invariant around it:

$$\text{LOGATOM-INV}\ \frac{\langle\, R * P \,\rangle\ e\ \langle\, v.\ R * Q(v) \,\rangle}{\boxed{R}^{\mathcal{N}} \vdash \langle\, P \,\rangle\ e\ \langle\, v.\ Q(v) \,\rangle}$$

This allows us to complete the proof of e_{client} using an invariant and 4.6:

Note that the invariant that we used in the above proof was

$$\boxed{\exists C.\ \mathsf{CSS}(r, C)}^{\mathcal{N}},$$

which existentially quantifies over the search structure contents C. This means that each time we open the invariant, the search structure can potentially have a different set of contents. This also means that when we close the invariant, we "forget" any changes we made to the contents, for example, that the left thread added k_1 to the contents. This weak invariant (which does not place any constraints on the contents) is sufficient because the postcondition here is simply True. If we wanted to prove something stronger, for example that the contents at the end of e_{client} will be $C \cup \{k_1, k_2\}$, then we would need a more complex invariant. In fact, we would need some way of keeping track of the updates the two threads make to the structure, for which we need to use ghost state (similar to the discussion in Chapter 4.4).

4.5.1 Proving Atomic Triples

Let us now turn to the question of how to prove that a program satisfies an atomic triple specification.

Recall that an atomic triple $\langle\, x.\ P \,\rangle\ e\ \langle\, v.\ Q \,\rangle$ means there is a single physical step during the execution of e when the shared state is transformed from P to Q. Thus, while proving $\langle\, x.\ P \,\rangle\ e\ \langle\, v.\ Q \,\rangle$, we cannot treat P and Q as the pre- and postconditions of e as a whole (remember, e is potentially a complex program consisting of multiple atomic steps). It is more accurate to think of P and Q as the pre- and postcondition to e's linearization point. Unlike proofs of Hoare triples, where we are given ownership to the resources in P at the beginning of e's execution and are under an obligation to transform them to Q by the end, in proofs of atomic triples we can read or modify the resources in the precondition only during atomic steps. Furthermore, our obligation is that all atomic steps accessing P either make a modification that preserves P, except for (exactly) one step, which has to transform it into Q.

Figure 4.6 contains the proof rules that help us execute the above proof argument. Most proofs of atomic triples start by using the rule LOGATOM-INTRO, which converts the atomic triple into a standard Hoare triple. The precondition contains an *atomic update token* $\mathsf{AU}_{x.P,Q}(\Phi)$, which records the fact that we are proving an atomic triple with precondition $x.\ P$ (recall, x is bound by a pseudo-quantifier) and postcondition Q. As we will see, this token gives us the *right* to use the resources in the precondition P when executing atomic instructions, but the token also records our *obligation* to transform P to Q before execution completes. One way to use the resources in P is to use the AU-ABORT rule, which gives us access to the precondition P if the expression e is atomic: i.e. if we can prove that e atomically transforms a global (shared) precondition P and some local precondition P' into a local postcondition Q' while leaving P unchanged. This rule is useful when a program has some initial operations that modify the shared state in a way that does not change the abstract state (for instance, by locking or performing maintenance on a node). At some point, however, the program must update the shared state to the postcondition Q. LOGATOM-INTRO

$$\textsc{logatom-intro}\ \frac{\forall \Phi.\ \left\{\mathsf{AU}_{x.P,Q}(\Phi)\right\}\ e\ \left\{v.\ \Phi(v)\right\}}{\left\langle\, x.\ P\,\right\rangle\ e\ \left\langle\, v.\ Q\,\right\rangle}$$

$$\textsc{logatom-atom}\ \frac{\forall x.\ \left\{P\right\}\ e\ \left\{v.\ Q\right\} \qquad e\ \text{atomic}}{\left\langle\, x.\ P\,\right\rangle\ e\ \left\langle\, v.\ Q\,\right\rangle}$$

$$\textsc{au-abort}\ \frac{\left\langle\, x.\ P * P'\,\right\rangle\ e\ \left\langle\, v.\ P * Q'\,\right\rangle}{\left\{\mathsf{AU}_{x.P,Q}(\Phi) * P'\right\}\ e\ \left\{v.\ \mathsf{AU}_{x.P,Q}(\Phi) * Q'\right\}}$$

$$\textsc{au-commit}\ \frac{\left\langle\, x.\ P * P'\,\right\rangle\ e\ \left\langle\, v.\ Q * Q'\,\right\rangle}{\left\{\mathsf{AU}_{x.P,Q}(\Phi) * P'\right\}\ e\ \left\{v.\ \Phi(v) * Q'\right\}}$$

$$\textsc{logatom-frame}\ \frac{\left\langle\, x.\ P\,\right\rangle\ e\ \left\langle\, v.\ Q\,\right\rangle}{\left\langle\, x.\ P * R\,\right\rangle\ e\ \left\langle\, v.\ Q * R\,\right\rangle}$$

Figure 4.6: Proof rules for establishing atomic triples.

enforces this obligation by using an unknown, universally quantified proposition $\Phi(v)$, in the postcondition of the Hoare triple. One can think of $\Phi(v)$ as the precondition for the continuation of the computation performed after e terminates, by a larger program containing e. The only way to prove $\Phi(v)$ is to use rule AU-COMMIT, which lets us exchange the atomic update token for $\Phi(v)$ if we can prove that the program e transforms the global state from P to Q in an atomic step. As with AU-ABORT, the rule allows for some extra local resources P' and Q' in the pre- and postcondition of e.

Part II: Contributions

5 | Keyset Reasoning in Iris

We have seen how to use ghost state in order to enable complex proof arguments in Chapter 4. In this chapter, we define a keyset resource algebra (RA) that can be used for many single-copy search structures, and demonstrate it by verifying a simple two-node template.

5.1 A Two-Node Template

```
1 let create () =
2   let n1, n2 = allocNodes () in
3   (n1, n2)
4
5 let cssOp op n1 n2 k =
6   let n = findNode n1 n2 k in
7   lockNode n;
8   let res = decisiveOp op n k in
9   unlockNode n;
10  res
```

Figure 5.1: A template algorithm for a two-node search structure.

In this section we present an example of a template algorithm for the simple search structure containing exactly two nodes (Figure 5.1, left). The `create` function creates a new search structure, by calling a `allocNodes` helper function, and returns the address of the root of the newly created structure.

The `cssOp` function (for concurrent search structure operation) stands for any one of the three core search structure operations, by means of the parameter `op`. Recall that search structures are data structures implementing a mathematical set data type, so `op` is either search, insert, or delete. Since there are two nodes, the first step in this algorithm is to find the node in which to search for, insert, or delete the given key. This is done via a new helper function `findNode`. Once the appropriate node n is found, the algorithm proceeds by locking n. Thereafter, it calls a function `decisiveOp` on the locked node, before unlocking the node and returning the result. The Boolean value returned indicates if the operation modified the search structure. For example, an insertion returns true if the given key was not already present in the structure.

$$\langle\, b\, R.\ \mathsf{L}(b, x, R)\,\rangle\ \texttt{lockNode}\ x\ \langle\, \mathsf{L}(\mathit{true}, x, R)\ *\ R\,\rangle$$

$$\langle\, R.\ \mathsf{L}(\mathit{true}, x, R)\ *\ R\,\rangle\ \texttt{unlockNode}\ x\ \langle\, \mathsf{L}(\mathit{false}, x, R)\,\rangle$$

Figure 5.2: Abstract specification for `lockNode` and `unlockNode`.

Implementations of this template choose not only how to store the keys in a node (e.g., as an array of keys or a list of keys) but also how to divide keys between nodes. For instance, one possible implementation would be to send the odd keys to n_1 and the even keys to n_2. We represent this choice in the template proof via an abstract function[1] $\mathsf{ks}(n)$ that maps a node n to a set of keys we call the *keyset*. Intuitively, we expect the implementation to define the keyset of a node n as the set of keys $\mathsf{ks}(n)$ that, if present in the structure, must be in n. In the above example, $\mathsf{ks}(n_1)$ is the set of odd numbers, and $\mathsf{ks}(n_2)$ is the set of even numbers. The proof of the template can use this keyset function to specify the behavior it expects from the `findNode` helper function:

$$\{\mathsf{True}\}\ \texttt{findNode}\ n_1\ n_2\ k\ \{n.\ \mathsf{InFP}(n_1, n_2, n) * k \in \mathsf{ks}(n)\}$$

Here, $\mathsf{InFP}(n_1, n_2, n) \coloneqq (n = n_1 \vee n = n_2)$ is a predicate that captures the fact that n is in the *footprint* of the data structure, i.e., that it is one of the nodes in the data structure.[2]

Before we move on to template algorithms for multi-node structures, we take a brief aside and present an abstract specification for the `lockNode` and `unlockNode` methods so that we can reuse the part of the proof dealing with the locking mechanism.

We define an abstract higher-order predicate $\mathsf{L}(b, x, R)$ that captures a *lock region*:

$$\mathsf{L}(b, x, R) \coloneqq \mathsf{lk}(x) \mapsto b\ *\ (b\ ?\ \mathsf{True} : R)$$

Here, R is a proposition that denotes an arbitrary resource protected by the lock with lock location $\mathsf{lk}(x)$ and lock bit b. The Boolean b indicates whether the lock is (un)locked. We express the specification for `lockNode` and `unlockNode` using the predicate $\mathsf{L}(b, x, R)$ as shown in Figure 5.2. Note that a thread can call `lockNode` or `unlockNode` only on a node x for which it owns the heap cell $\mathsf{lk}(x)$ – this is where the $\mathsf{InFP}(n_1, n_2, n)$ predicate will be used.

The challenge is in providing a suitable specification for `decisiveOp`. At the point when `decisiveOp` is called, only one of the two nodes in the structure is locked by the current thread, and hence any specification for `decisiveOp` can speak only about the node n. A natural first-attempt would be:

$$\{\mathsf{Node}(n, C_n)\}\ \texttt{decisiveOp}\ \mathsf{op}\ n\ k\ \{\mathit{res}.\ \mathsf{Node}(n, C_n') * \Psi_{\mathsf{op}}(k, C_n, C_n', \mathit{res})\},$$

[1] Abstract functions are like abstract predicates in that the template proof is done without knowing their definition; instead, the proof relies on certain assumptions about them.

[2] While this is a trivial definition for the two-node template, we will use the same predicate to simplify the more complex proofs in later chapters.

$$\Psi_{\mathsf{op}}(k, C, C', res) \coloneqq \begin{cases} C' = C \wedge (res \iff k \in C) & \mathsf{op} = \texttt{search} \\ C' = C \cup \{k\} \wedge (res \iff k \notin C) & \mathsf{op} = \texttt{insert} \\ C' = C \setminus \{k\} \wedge (res \iff k \in C) & \mathsf{op} = \texttt{delete} \end{cases}$$

Figure 5.3: Sequential specification of a search structure as a Set ADT. k refers to the operation key, C and C' to the abstract state before and after operation op, respectively, and res is the return value of op.

This spec says `decisiveOp` converts node n with contents C_n ($\mathsf{Node}(n, C_n)$) into node n with updated contents C'_n ($\mathsf{Node}(n, C'_n)$) such that the search structure specification predicate $\Psi_{\mathsf{op}}(k, C_n, C'_n, res)$ holds. In this chapter, we assume that predicate $\Psi_{\mathsf{op}}(k, C_n, C'_n, res)$ represents the Set ADT specification shown in Figure 5.3.

Note that the postcondition of `cssOp` requires us to show that the contents of the entire search structure are modified from some C to C' such that $\Psi_{\mathsf{op}}(k, C, C', res)$ holds. To complete the proof, we need to show that $\Psi_{\mathsf{op}}(k, C_n, C'_n, res) \Rightarrow \Psi_{\mathsf{op}}(k, C, C', res)$.

This is not true of arbitrary sets $C_n \subseteq C$ and $C'_n \subseteq C'$. Consider the case where node n_1 has contents $\{1, 3, 8\}$, and n_2 has contents $\{2, 4, 8\}$ and `decisiveOp` removes key 8 from n_1. Here $C_n = \{1, 3, 8\}$ and $C'_n = \{1, 3\}$, but $C = C' = \{1, 2, 3, 4, 8\}$.

So, we need more constraints. Our example implementation assigned each node a distinct set of keys (n_1 got the odd keys and n_2 got even keys). The missing piece of the proof is the property that the keysets of any two nodes are *disjoint*. If we have a data structure where all keysets are disjoint and the contents of each node n are a subset of the keyset of n, then we can show that it is sufficient for `decisiveOp` to ensure that Ψ holds on some node n such that $k \in \mathsf{ks}(n)$. We next show how to encode this argument in separation logic using an appropriate resource algebra.

5.2 Disjoint Keysets and the Keyset RA

We define an RA that we use to keep track of the keyset and contents of each node simultaneously:

Definition 3 *Given a key space* $\mathbb{K}$*, the* keyset *RA is defined as:*

$$\mathit{KEYSET} \coloneqq (\mathbb{K} \times \mathbb{K}) \mid \lightning \qquad \mathcal{V}((K, C)) \coloneqq (C \subseteq K) \qquad \mathcal{V}(\lightning) \coloneqq \mathit{False}$$

$$(K_1, C_1) \cdot (K_2, C_2) \coloneqq \begin{cases} (K_1 \cup K_2, C_1 \cup C_2) & \text{if } C_1 \subseteq K_1 \wedge C_2 \subseteq K_2 \wedge K_1 \cap K_2 = \emptyset \\ \lightning & \text{otherwise} \end{cases}$$

$$\lightning \cdot \_ \coloneqq \_ \cdot \lightning \coloneqq \lightning$$

The unit of this RA is the element $(\emptyset, \emptyset)$.

This is an RA where elements are pairs of sets of keys, where the first set represents the keyset and the second represents the contents of a node (or, more generally, a set of nodes). We also have a special element $\lightning$ representing invalid compositions. The validity predicate checks if the contents are a subset of the keyset, and composition is only defined between valid elements whose keysets are disjoint.

In order to use the keyset RA in our proofs, we will need a standard RA construction useful to reason about shared ownership of a logical value. The *authoritative* RA $\textsc{Auth}(M)$ [74, 83], constructed from an arbitrary RA M, is used to model situations where there exists an authoritative element a of M, and threads own fragments b of a such that $b \preccurlyeq a \coloneqq \exists c.\ a = b \cdot c$.

Definition 4 *Given an RA M with unit ε, the* authoritative *RA $\textsc{Auth}(M)$ is defined as:*

$$\textsc{Auth}(M) \coloneqq (\mathsf{ex}(M) \mid \lightning) \times M \qquad \mathcal{V}((x, b)) \coloneqq (\exists a.\ x = \mathsf{ex}(a) \land b \preccurlyeq a \land \mathcal{V}(a))$$

$$(x_1, b_1) \cdot (x_2, b_2) \coloneqq \begin{cases} (x_1, b_1) & \text{if } x_2 = \mathsf{ex}(\varepsilon) \\ (x_2, b_1) & \text{if } x_1 = \mathsf{ex}(\varepsilon) \\ (\lightning, b_1 \cdot b_2) & \text{otherwise} \end{cases}$$

The unit of $\textsc{Auth}(M)$ is the element $(\mathsf{ex}(\varepsilon), \varepsilon)$.

Let $a, b \in M$. When using the $\textsc{Auth}(M)$ RA, we write $\bullet\, a$ for ownership of an authoritative element $(\mathsf{ex}(a), \varepsilon)$ and $\circ\, b$ for fragmental ownership $(\mathsf{ex}(\varepsilon), b)$ and $\bullet\, a, \circ\, b$ for combined ownership $(\mathsf{ex}(a), b)$. The composition operator is defined so that only one authoritative element can be owned, as $(\bullet\, a_1, \circ\, b_1) \cdot (\bullet\, a_2, \circ\, b_2) = (\lightning, b_1 \cdot b_2)$ which is invalid. However, multiple fragmental elements can be owned simultaneously, and they compose according to the composition of the underlying RA:

$$\textsc{auth-frag-op}\ (\circ\, a) \cdot (\circ\, b) = \circ\, (a \cdot b)$$

An important property of authoritative RAs is that if one owns both an authoritative element $\bullet\, a$ and a fragment $\circ\, b$, then by the definition of validity, we know that the fragment is a part of the authoritative element, i.e., $b \preccurlyeq a$.

In our proofs, we will be using $\textsc{Auth}(\textsc{Keyset})$, the authoritative keyset RA. Using $\textsc{Auth}(\textsc{Keyset})$, we add the formula $\boxed{\bullet(\mathbb{K}, C)}^{\gamma}$ to the definition of CSS to represent the abstract state of the search structure as one whose keyset is the entire key space $\mathbb{K}$ and contains the keys C. Similarly, we represent the local abstract state of a node n by the formula $\boxed{\circ(K_n, C_n)}^{\gamma}$, where K_n and C_n are the keyset and contents, respectively, of n. By the definition of the authoritative RA, the assertion

$$\boxed{\bullet(\mathbb{K}, C)}^{\gamma} * \mathop{\Large *}_{n \in N} \boxed{\circ(K_n, C_n)}^{\gamma}$$

expresses that the sets K_n for each $n \in N$ are disjoint and their union is included in $\mathbb{K}$.[3] Moreover, $C_n \subseteq K_n$ and similarly the C_n sets are disjoint and are included in C. If we can associate each C_n and K_n to the contents and keyset, respectively, of n, then an assertion like the one above gives us the desired disjoint decomposition of the abstract state into local states.

The Auth(Keyset) RA has frame-preserving updates such as the following, which we will use to update the ghost state when we insert or delete a key k:

$$\text{KS-INS}\ \frac{k \notin K_n}{\bullet(K, C), \circ(K_n, C_n) \rightsquigarrow \bullet(K, C \cup \{k\}), \circ(K_n, C_n \cup \{k\})}$$

$$\text{KS-DEL}\ \frac{k \in K_n}{\bullet(K, C), \circ(K_n, C_n) \rightsquigarrow \bullet(K, C \setminus \{k\}), \circ(K_n, C_n \setminus \{k\})}$$

For example, KS-DEL says that if $\boxed{\bullet(K, C)}^{\gamma}$ and $\boxed{\circ(K_n, C_n)}^{\gamma}$ are valid resources such that $k \in K_n$ then we can update the fragment to $(K_n, C_n \setminus \{k\})$ (for instance when we remove k from the contents of a node n) and the authoritative resource to $(K, C \setminus \{k\})$ (meaning k is also removed from the global contents). Combining this with KS-INS for insertions and case-analysis on `op`, we get the following lemma:

$$\text{KS-UPD}\ \begin{array}{l} \boxed{\bullet(K, C)}^{\gamma} * \boxed{\circ(K_n, C_n)}^{\gamma} * k \in K_n * \Psi_{\mathtt{op}}(k, C_n, C_n', res) \\ \Rrightarrow \exists C'.\ \boxed{\bullet(K, C')}^{\gamma} * \boxed{\circ(K_n, C_n')}^{\gamma} * \Psi_{\mathtt{op}}(k, C, C', res) \end{array}$$

The lemma KS-UPD captures the intuition that changes made locally to C_n percolate through the global contents C due to disjointness of keysets and the fact that a node's contents is always a subset of its keyset. The lemma is proved from KS-INS and KS-DEL by case analysis on the operation `op` and application of rules GHOST-UPDATE, VS-TRANS and VS-FRAME.

5.3 Proof of the Two-Node Template

We can now prove the two-node template (Figure 5.5). The definition of CSS has been extended to account for two nodes and a lock region for each. It also contains the authoritative version of the keyset and global contents: $\boxed{\bullet(\mathbb{K}, C)}^{\gamma}$. Each node is represented by the node

[3] We cannot use the keyset RA to encode the invariant that the union of the sets K_n *cover* the key space $\mathbb{K}$ because the authoritative RA's validity predicate tells us only that fragmental elements are included in the authoritative element. While practical concurrent search structure implementations will ensure that keysets cover the key space, we do not need to maintain such an invariant in our proofs because we prove only partial correctness, and not termination. For instance, suppose we had a two-node structure where the keysets did not cover the key space. Given a k not in the keyset of either node, the only way for `findNode` n_1 n_2 k to satisfy its specification is for it to not terminate.

$$\{\mathsf{True}\}\ \texttt{allocNodes}\ ()\ \{n_1, n_2.\ \mathsf{Node}(n_1, \emptyset) * \mathsf{lk}(n_1) \mapsto \mathit{false} * \mathsf{Node}(n_2, \emptyset) * \mathsf{lk}(n_2) \mapsto \mathit{false}\}$$

$$\{\mathsf{True}\}\ \texttt{findNode}\ n_1\ n_2\ k\ \{n.\ \mathsf{InFP}(n_1, n_2, n) * k \in \mathsf{ks}(n)\}$$

$$\{\mathsf{Node}(n, C_n)\}\ \texttt{decisiveOp}\ \mathsf{op}\ n\ k\ \{\mathit{res}.\ \mathsf{Node}(n, C'_n) * \Psi_{\mathsf{op}}(k, C_n, C'_n, \mathit{res})\}$$

$$\mathsf{Node}(n, C_n) * \mathsf{Node}(n, C'_n) \twoheadrightarrow \mathsf{False}$$

Figure 5.4: The assumptions made by the two-node template on implementations.

predicate $\mathsf{N}(n)$, which contains the abstract predicate $\mathsf{Node}(n, C_n)$ that is implementation-specific as well as the fragment containing n's keyset and contents $\boxed{\circ(\mathsf{ks}(n), C_n)}^{\gamma}$.

We first describe the proof of the `create` method that constructs the search structure. The specification of `create` is a Hoare triple because the search structure is created before the concurrent context begins or any threads are created. We use the helper function `allocNodes` that allocates the nodes $\mathsf{Node}(n_1, \emptyset)$ and $\mathsf{Node}(n_2, \emptyset)$ with empty contents. It also creates the lock bit for each node. We then use GHOST-ALLOC to allocate a ghost location γ with contents $\bullet\,(\mathbb{K}, \emptyset) \cdot \circ\,(\mathsf{ks}(n_1), \emptyset) \cdot \circ\,(\mathsf{ks}(n_2, \emptyset))$. This ghost state is valid because the keysets of the two nodes are disjoint and included in the key space $\mathbb{K}$. We then use GHOST-OP to split this into the authoritative version and two fragments, which we then fold into the predicates $\mathsf{N}(n_1)$ and $\mathsf{N}(n_2)$. We can then combine these with the lock locations to get the proof context shown in line 11. By definition of CSS, this gives us the desired postcondition, which is a search structure with empty contents.

Moving to `cssOp`, the call to `findNode` is handled as explained previously, using the specification given in Figure 5.4. To prove the precondition of `lockNode`, we open the precondition and use the predicate $\mathsf{InFP}(n_1, n_2, n)$ that we obtained from `findNode` to show that we own $\mathsf{L}(b, n, \mathsf{N}(n))$. After `lockNode`, we can move the predicate $\mathsf{N}(n)$ from the shared state into our local state as before. We then use the specification of `decisiveOp` to get a modified node predicate $\mathsf{Node}(n, C'_n)$ and $\Psi_{\mathsf{op}}(k, C_n, C'_n, \mathit{res})$.

The linearization point for the two-node template is at the call to `unlockNode`. We use the rule AU-COMMIT to open the precondition and get access to the shared state, obtaining the resources shown in the intermediate assertion on line 28. We then use KS-UPD to update both the node's fragment of the keyset RA as well as the authoritative element to the new contents, and obtain the resource $\Psi_{\mathsf{op}}(k, C, C', \mathit{res})$. This step corresponds to the reasoning that since the decisive operation was performed on a node n such that $k \in \mathsf{ks}(n)$, the global contents also change appropriately. We can then apply `unlockNode`'s specification to change the lock location of n, and return $\mathsf{N}(n)$ to the shared state, obtaining the postcondition $\mathsf{CSS}(n_1, n_2, C') * \Psi_{\mathsf{op}}(k, C, C', \mathit{res})$.

$\mathsf{InFP}(n_1, n_2, n) \coloneqq (n = n_1 \vee n = n_2)$

$\mathsf{N}(n) \coloneqq \exists C_n.\ \mathsf{Node}(n, C_n) * \boxed{\circ(\mathsf{ks}(n), C_n)}^{\gamma}$

$\mathsf{CSS}(n_1, n_2, C) \coloneqq \exists b_1, b_2.\ \boxed{\bullet(\mathbb{K}, C)}^{\gamma} * \mathsf{L}(b_1, n_1, \mathsf{N}(n_1)) * \mathsf{L}(b_2, n_2, \mathsf{N}(n_2))$

$\{\mathsf{True}\}$
let create () =
  $\{\mathsf{True}\}$
  let n_1, n_2 = allocNodes () **in**
  $\{\mathsf{Node}(n_1, \emptyset) * \mathsf{lk}(n_1) \mapsto \mathit{false} * \mathsf{Node}(n_2, \emptyset) * \mathsf{lk}(n_2) \mapsto \mathit{false}\}$
  $\left\{\mathsf{Node}(n_1, \emptyset) * \mathsf{lk}(n_1) \mapsto \mathit{false} * \mathsf{Node}(n_2, \emptyset) * \mathsf{lk}(n_2) \mapsto \mathit{false} * \boxed{\bullet(\mathbb{K}, \emptyset) \cdot \circ(\mathsf{ks}(n_1), \emptyset) \cdot \circ(\mathsf{ks}(n_2), \emptyset))}^{\gamma}\right\}$
  $\left\{\boxed{\bullet(\mathbb{K}, \emptyset)}^{\gamma} * \mathsf{L}(\mathit{false}, n_1, \mathsf{N}(n_1)) * \mathsf{L}(\mathit{false}, n_2, \mathsf{N}(n_2))\right\}$
  $\{\mathsf{CSS}(n_1, n_2, \emptyset)\}$
  (n_1, n_2)
$\{\mathsf{CSS}(n_1, n_2, \emptyset)\}$

$\langle\, C.\ \mathsf{CSS}(n_1, n_2, C)\,\rangle$
let cssOp op n_1 n_2 k =
  $\{\mathsf{True}\}$
  let n = findNode n_1 n_2 k **in**
  $\{\mathsf{InFP}(n_1, n_2, n) * k \in \mathsf{ks}(n)\}$
    $\langle\, \mathsf{CSS}(n_1, n_2, C) * \mathsf{InFP}(n_1, n_2, n) * k \in \mathsf{ks}(n)\,\rangle$
    lockNode n;
    $\langle\, \mathsf{CSS}(n_1, n_2, C) * \mathsf{N}(n) * k \in \mathsf{ks}(n)\,\rangle$
  $\{\mathsf{N}(n) * k \in \mathsf{ks}(n)\}$
  $\left\{\mathsf{Node}(n, C_n) * \boxed{\circ(\mathsf{ks}(n), C_n)}^{\gamma} * k \in \mathsf{ks}(n)\right\}$
  let res = decisiveOp op n k **in**
  $\left\{\mathsf{Node}(n, C_n') * \boxed{\circ(\mathsf{ks}(n), C_n)}^{\gamma} * \Psi_{\mathsf{op}}(k, C_n, C_n', res) * k \in \mathsf{ks}(n)\right\}$
    $\left\langle\, \mathsf{Node}(n, C_n') * \boxed{\circ(\mathsf{ks}(n), C_n)}^{\gamma} * \Psi_{\mathsf{op}}(k, C_n, C_n', res) * \boxed{\bullet(\mathbb{K}, C)}^{\gamma} * k \in \mathsf{ks}(n) * \cdots\,\right\rangle$
    $\left\langle\, \mathsf{Node}(n, C_n') * \boxed{\circ(\mathsf{ks}(n), C_n')}^{\gamma} * \Psi_{\mathsf{op}}(k, C, C', res) * \boxed{\bullet(\mathbb{K}, C')}^{\gamma} * \cdots\,\right\rangle$ (* By KS-UPD *)
    unlockNode n;
    $\langle\, \mathsf{CSS}(n_1, n_2, C') * \Psi_{\mathsf{op}}(k, C, C', res)\,\rangle$
  $\{\mathsf{True}\}$
  res
$\langle\, v.\ \mathsf{CSS}(n_1, n_2, C') * \Psi_{\mathsf{op}}(k, C, C', v)\,\rangle$

Figure 5.5: Proof of the two-node template algorithm.

6 | Hindsight Reasoning

In this chapter, we explain our hindsight framework. We motivate the basics of *Hindsight Reasoning* [46, 47, 108, 117, 118, 133] via a simplified distributed counter data structure from [118]. This is followed by a high-level overview of our framework in Iris. As part of the overview, we provide the perspective of a proof author using our framework to verify linearizability of a data structure in Iris. Finally, we close the chapter by providing the technical details underlying our framework.

6.1 A Distributed Counter

A distributed counter object abstractly represents a natural number n with two operations, `read` which returns the current value of the counter and `incr` which increments the value of the counter by 1. The counter is composed of two memory locations each storing a natural number. The abstract value of the counter is the sum of the numbers stored in the two memory locations.

```
1  let mk_counter #() =
2    let l1 = ref 0 in
3    let l2 = ref 0 in
4    (l1, l2)
5
6  let read c =
7    let x = !(fst c) in
8    let y = !(snd c) in
9    x + y
10 let incr c =
11   if (nondet ()) then
12     FAA (fst c) 1
13   else
14     FAA (snd c) 1
```

Figure 6.1: Implementation of a distributed counter. Command `ref` allocates a new memory location. Commands `fst` and snd evaluate a pair of values to its first and second component respectively. Function nondet returns a boolean value chosen non-deterministically. Command FAA stands for the atomic *fetch-and-add* operation.

Figure 6.1 provides an implementation of the distributed counter. Operation `mk_counter` creates a new distributed counter object by allocating two memory locations with value 0 stored in both locations. Operation `read` reads the values stored in the two locations successively and returns their sum. Operation `incr` non-deterministically picks one of the locations and increments it by 1 atomically via a fetch-and-add operation.

6.2 Linearizability of the Distributed Counter

$$\Psi_{\mathsf{op}}(n, n', res) \coloneqq \begin{cases} n' = n \land res = n & \mathsf{op} = \texttt{read} \\ n' = n + 1 \land res = () & \mathsf{op} = \texttt{incr} \end{cases}$$

Figure 6.2: Sequential specification of the distributed counter. Parameters n and n' refer to the abstract state of the distributed counter before and after operation op, respectively, and res is the return value of op.

Let us focus on the correctness reasoning for the distributed counter. In order to establish linearizability of the distributed counter, we must determine linearization points for each of its operations. That is, an atomic step where the effect of the operation takes place according to the sequential specification of the counter. The sequential specification of a counter is shown in Figure 6.2. It establishes that `read` does not change the abstract state and returns the current value of the counter. On the other hand, `incr` increments the counter value by one and returns a unit value.

We refer to a linearization point as *modifying* if the operation changes the abstract state of the data structure (like `incr`) and otherwise refer to it as *unmodifying* (like `read`). The modifying linearization points of a data structure are typically easier to reason about. For instance, the linearization point for `incr` is when the fetch-and-add `FAA` (in Line 12 or Line 14) takes effect. However, it is not easy to determine the unmodifying linearization points of an operation because they may be future-dependent. For instance, the linearization point of `read` cannot be determined at any fixed moment, but only at the end of the execution, once any interference of other concurrent operations has been accounted for. In other words, no atomic step in `read` can be confirmed as its linearization until the `read` has terminated. Let us explain this in further detail.

We introduce an abstract predicate $\mathsf{counter}(n_1, n_2)$ to represent the state of the counter where the first location stores value n_1 and the second n_2. Let T_r be a thread executing `read`, while concurrently there are two threads T_1 and T_2 executing `incr`. Let the initial state of the counter be $\mathsf{counter}(0, 0)$. Consider the following interleaving steps of the concurrent execution of the three threads :

(S1) T_r reads the first location to be 0.

(S2) T_1 increments the first location : $\mathsf{counter}(0, 0) \rightsquigarrow \mathsf{counter}(1, 0)$.

(S3) T_2 increments the second location : $\mathsf{counter}(1, 0) \rightsquigarrow \mathsf{counter}(1, 1)$.

(S4) Finally, T_r reads the second location and returns with final value 1.

In summary, T_r observes the first location with value 0 and second with value 1, although the counter never attained the state $\mathsf{counter}(0, 1)$. Nevertheless, this execution of `read` is

linearizable, with the linearization point being the step (S2) when T_1 increments the first location. To emphasize, the linearization point for thread T_r is an atomic step of thread T_1. Contrast this execution trace with one where steps (S3) and (S4) are swapped. That is, T_r reads the second location with value 0 before T_2 increments it. In this case, T_r returns 0 and its linearization point is step (S1). This is why the linearization point of `read` is future-dependent.

The intuitive argument for linearizability of the `read` operation relies on the following two invariants of the distributed counter:

Invariant 1 The values in the two locations of the distributed counter are monotonically non-decreasing.

Invariant 2 The abstract value of the distributed counter (i.e. the sum of the values in the two locations) increases by at most one for every atomic step.

Invariant 1 ensures that if a thread observes a state of the counter as $\mathsf{counter}(a_1, a_2)$ first and then $\mathsf{counter}(b_1, b_2)$ at some later point in time, then we can infer $a1 \leq b1$ and $a2 \leq b2$. In a similar vein, if a thread observes the abstract value of the counter to be n_1 first and then n_2 at some later point in time, then Invariant 2 allows us to infer: (i) $n1 \leq n2$; and (ii) the abstract state of the counter must have attained all values between n_1 and n_2.

Equipped with the two invariants, we can now present the intuitive argument for the linearizability of the `read` operation. From the perspective of a thread executing `read`, let the state of the counter be $\mathsf{counter}(a_1, a_2)$ when the first location is read (Line 7). The value read is a_1. Let the state of the counter be $\mathsf{counter}(b_1, b_2)$ when the second location is read (Line 8). The value read is b_2. From Invariant 1 above, the thread can infer that $a_1 + a_2 \leq a_1 + b_2 \leq b_1 + b_2$. Additionally, by Invariant 2, the thread can establish that in the time period between the two reads, the counter must have attained a state with abstract value $a_1 + b_2$ at some point. This point becomes its linearization point, completing the intuitive proof argument.

Hindsight reasoning [117, 118] is designed to formalize proof arguments like one for the `read` operation above. It enables temporal reasoning about computations using a *past predicate* $\diamond\!\!\cdot p$, which expresses that proposition p held true at some prior state in the computation (up to the current state). For instance, thread T_r can establish $\diamond\!\!\cdot(\mathsf{counter}(0, 0))$ at step (S4) even though the current state of the counter at that point is $\mathsf{counter}(1, 1)$. The reason is that $\mathsf{counter}(0, 0)$ was true at an earlier point in time, namely at (S1). Note that the past operator $\diamond\!\!\cdot$ abstracts away the exact time point when the predicate held true. Note also that a past predicate is not affected by concurrent interferences, as it merely records some fact about a past state.

There are two ways to establish a past predicate that are relevant for our proofs. The first is to establish the predicate in the current state directly. That is, $\diamond\!\!\cdot p$ holds if p holds in the current state. As an example, thread T_r observes the state $\mathsf{counter}(0, 0)$ at step (S1). Thus, for all subsequent steps (S2-S4), T_r can establish $\diamond\!\!\cdot(\mathsf{counter}(0, 0))$. The second way to establish a past predicate is through the use of *temporal interpolation* [118]. That is, one

proves a lemma of the form: if there existed a past state that satisfied property p and the current state satisfies q, then there must have existed an intermediate state that satisfied o. As an example, thread T_r observes the counter with state $\mathsf{counter}(1,1)$ at step (S4), while also holding the knowledge $\diamond\!\!\cdot(\mathsf{counter}(0,0))$. Invariant 2 allows thread T_r to infer via temporal interpolation that there must have been an intermediate state where the abstract value of the counter was 1 (namely, at the end of step (S2)). Note that the temporal interpolation does not establish that state $\mathsf{counter}(1,0)$ existed in particular. It only establishes that at some point, the abstract state must have obtained value 1.

6.3 Hindsight Reasoning in Iris

Linearizability in Iris is defined via *(logically) atomic triples* [33, 86] described in Section 4.5. Linearizability of a data structure operation op can be expressed by an atomic triple of the form

$$\boxed{\mathsf{Inv}(r)} \twoheadrightarrow \langle\, C.\ \mathsf{DS}(r,C) \,\rangle\ \mathsf{op}\, r\ \langle\, res.\ \exists\, C'.\ \mathsf{DS}(r,C') \ast \Psi_{\mathsf{op}}(C,C',res) \,\rangle. \qquad \text{(ClientSpec)}$$

Here, r is the pointer that provides access to the data structure. The predicate $\mathsf{DS}(r,C)$ is the *representation predicate* that relates the head pointer with the abstract state C of the structure. For instance, in the case of the distributed counter the abstract state refers to the sum of the two locations of the counter. The predicate $\mathsf{Inv}(r)$ is the shared data structure invariant. It can be thought of as a thread-local precondition of the atomic triple, which we express using a separating implication. The invariant ties $\mathsf{DS}(r,C)$ to the data structure's physical representation and may contain other resources necessary for proving the atomic triple. The predicate $\Psi_{\mathsf{op}}(C,C',res)$ captures the sequential specification of the structure. The specification essentially says there is a single atomic step in op where the abstract state changes from C to C' according to the sequential specification $\Psi_{\mathsf{op}}(C,C',res)$ (Figure 6.2 for the distributed counter). This step is op's linearization point. We call (ClientSpec) the *client-level* atomic specification for the data structure under proof.

Proving atomic triples. The proof of establishing an atomic triple involves a *linearizability obligation* that must be discharged directly at the linearization point. However, it can be challenging to determine the linearization point precisely and to discharge the linearizability obligation exactly at that point. When the program execution reaches a potential linearization point that depends on future interferences by other threads, then the proof will fail if it is unable to determine whether the linearizability obligation should be discharged now or later. In Iris, this challenge is overcome using *prophecy variables* [85], which enable the proof to reason about the remainder of the computation that has not yet been executed.

Another challenge is that the linearization point of an operation may be an atomic step of another operation that is executed by a different thread (like thread T_r executing read in Section 6.2). Data structures that demonstrate such behavior are said to deploy *helping*. This behavior complicates thread modular reasoning. The conventional solution to this challenge in Iris is to use a *helping protocol* [81, 85, 138]. The helping protocol is specified as part

of the shared data structure invariant and consists of a registry that tracks which threads are expected to be linearized by other threads as well as conditional logic that governs the correct transfer and discharge of the associated linearizability obligations.

Both the use of prophecy variables and the helping protocol need to be tailored to the specific data structure at hand, which adds considerable overhead to the proof. To reduce this overhead, we present an alternative proof method that enables linearizability proofs based on hindsight arguments in Iris. Rather than identifying the linearization point precisely, the proof can establish linearizability in hindsight using temporal interpolation in the style of the intuitive proof argument for the distributed counter presented in Section 6.2.

Hindsight specification. Our proof method offers an intermediate specification, a Hoare triple specification, which in essence expresses that linearizability has been established in hindsight. In our Iris formalization, we show that any data structure whose operations satisfy the hindsight specification also satisfy the client-level atomic specification. This proof relates the two specifications via prophecy variables and a helping protocol. However, the helping protocol is data structure agnostic, making our proof method applicable to a broad class of structures exhibiting future-dependent unmodifying linearization points.

From the perspective of a proof author using our method to prove linearizability of some structure, one has to only establish the hindsight specification to obtain the proof of the client-level atomic specification. To this end, our method provides further guidance to the proof author.

In order to use hindsight reasoning, one has to have the history of computation at hand. Here, we offer a shared state invariant with a mechanism to store the history. The shared state invariant has three main components: a mechanism to store the history, the helping protocol, and finally, an abstract predicate that can be instantiated with invariants specific to the structure at hand. The first two components are data structure agnostic. The proof author only needs to specify the data structure-specific invariant and what information about the data structure state should be tracked by the history.

In the rest of this section, we discuss our method in detail. We begin with the hindsight specification, followed by a discussion of the shared state invariant and how to use it.

6.3.1 Linearizability in Hindsight

We motivate the hindsight specification using the challenges we face when proving the client-level atomic specification for the operations of the distributed counter. Let us recall the intuitive proof argument from Section 6.2. A thread executing `incr` will modify the structure and it can fulfill its linearizability obligation when the structure is modified. On the other hand, a thread executing `read` exhibits an unmodifying linearization point, which requires helping.

Prophecy reasoning. An important detail of our proof method is how it determines whether a thread requires helping. Whether a thread requires helping or not is dependent on whether it modifies the structure. In the following, we refer to the operation that may potentially change the abstract state of the structure as its *decisive operation*. In `incr`, these

are FAA operations on the locations (Line 12 or Line 14). Operation read on the other hand has no decisive operation. Across data structures, a decisive operation can take many forms. It may be physically atomic (like FAA) or only *logically* atomic (as we will see with the skiplist data structure).

In order to determine in advance whether a thread requires helping, our proof method attaches a prophecy to each thread. A prophecy in Iris can predict a sequence of values and is treated as a resource that can be owned by a thread. Ownership of a prophecy p is captured by the predicate $\mathsf{Proph}(p, pvs)$, where pvs is the list of predicted values. The predicate signifies the right to resolve p when the thread makes a physical step that produces some result value v. The resolution of p establishes equality between v and the head of the list pvs (i.e., the next value predicted by p). The resolution step yields the updated predicate $\mathsf{Proph}(p, pvs')$ where pvs' is the tail of pvs. This mechanism enables the proof to do a case analysis on the predicted values pvs before these values have been observed in the program execution[1].

The prophecy attached to a thread predicts the results of the thread's decisive operation. Note that an operation may have multiple decisive operations. It is also possible that a decisive operation fails and the operation restarts to attempt it again. Therefore, the prophecy needs to predict a sequence of result values, one for each attempted call to the thread's decisive operation.

For the purpose of this discussion, we assume that the prophecy predicts a sequence of Success or Failure values. If the sequence contains a Success value, then the decisive operation will succeed and the thread will modify the structure. Otherwise, the thread's linearization point is unmodifying. Let predicate $\mathsf{Upd}(pvs)$ hold when pvs contains at least one Success value.

The proof author needs only to identify the decisive operations that potentially change the abstract state of the structure (like incr as discussed above) by resolving the prophecy around these decisive calls.

Hindsight specification. Before we can present the hindsight specification, we recall the details regarding atomic triples in Iris from Chapter 4. An atomic triple $\langle\, x.\ P \,\rangle\ e\ \langle\, v.\, Q \,\rangle$ is defined in terms of standard Hoare triples of the form $\forall\ \Phi.\ \{\mathsf{AU}_{x.P,Q}(\Phi)\}\ e\ \{v.\, \Phi(v)\}$. The predicate $\mathsf{AU}_{x.P,Q}(\Phi)$ is the *atomic update token* and represents the linearizability obligation of the atomic triple. At each atomic step, it offers the thread a choice to linearize by *committing* the atomic update. Once committed, the atomic update transforms to $\Phi(v)$, which serves as a receipt of linearization.

We also introduce two auxiliary predicates:

- $\mathsf{Thread}(tid, t_0)$: this predicate is used to *register* the thread with identifier tid in the shared invariant. The argument t_0 denotes the time when thread tid began its execution.

- $\mathsf{PastLin}(\mathsf{op}, res, t_0)$: this predicate holds if there was a past state in the history between time t_0 and the point when this predicate is evaluated for which the sequential

[1]For further details on prophecies in Iris, we refer to [85].

specification Ψ_{op} held with result res. It essentially captures whether the sequential specification was true for any point after time t_0.

We now have all the ingredients to present the hindsight specification:

$$\begin{array}{l}\forall\ tid\ t_0\ pvs.\ \boxed{\mathsf{Inv}(r)} \mathrel{-\!\!*} \mathsf{Thread}(tid, t_0) \mathrel{-\!\!*} \\ \left\{\mathsf{Proph}(p, pvs)\ *\ (\mathsf{Upd}(pvs) \mathrel{-\!\!*} \mathsf{AU}_{\mathsf{op}}(\Phi))\right\}\ \mathsf{op}\ r\ k \\ \left\{\begin{array}{r} res.\ \exists pvs'.\ \mathsf{Proph}(p, pvs')\ *\ pvs = (\_\ @\ pvs') \\ *\ (\mathsf{Upd}(pvs) \mathrel{-\!\!*} \Phi(res)) \\ *\ (\neg\mathsf{Upd}(pvs) \mathrel{-\!\!*} \mathsf{PastLin}(\mathsf{op}, res, t_0)) \end{array}\right\}\end{array} \qquad \text{(HindSpec)}$$

We explain the specification piece by piece. The local precondition $\mathsf{Thread}(tid, t_0)$ ties the thread to its identifier tid and provides knowledge that tid begins executing at time t_0. The Hoare triple can be best understood by observing how prophecy resources are allowed to change (highlighted in brown) and what are the obligations when $\mathsf{Upd}(pvs)$ holds (in teal) versus when it does not hold (in magenta). Let us look at each of these in detail. First, the prophecy resource $\mathsf{Proph}(p, pvs)$ in the precondition changes to $\mathsf{Proph}(p, pvs')$ in the postcondition where pvs' is a suffix of pvs. It basically says that operation op is allowed to resolve the prophecy p as many times as necessary and then return the remaining resource at the end.

Now let us consider the case when $\mathsf{Upd}(pvs)$ holds. The precondition here provides the atomic update token $\mathsf{AU}_{\mathsf{op}}(\Phi)$ to op, expecting the receipt of linearization $\Phi(res)$ in return. Thus, the responsibility of linearization is delegated to op when $\mathsf{Upd}(pvs)$ holds. We can gain better insight by relating this situation to the distributed counter as before. This case corresponds to verifying the `incr` operation. The point when `FAA` is called becomes the linearization point and so the thread does not require help from other threads to linearize. The hindsight specification simply asks for the receipt from linearization $\Phi(res)$ at the end.

Next, let us consider the case when $\mathsf{Upd}(pvs)$ does not hold. The precondition provides no additional resources here, while the postcondition requires the predicate $\mathsf{PastLin}(\mathsf{op}, res, t_0)$. In simple terms, this means that if $\mathsf{Upd}(pvs)$ is not true, i.e., the prophecy says the thread is not going to modify the structure, then the hindsight specification allows exhibiting a past state from history when the sequential specification was true. Relating again to the distributed counter, this is applicable when verifying the `read` operation. According to the specification, the thread can look at the history of the structure and exhibit precisely the point when its return value matches the abstract state of the counter.

The proof argument for establishing the hindsight specification is significantly simpler than if one were to attempt a direct proof of the client-level atomic specification[2]. In particular, the proof author does not need to reason about helping and atomic update tokens in last case discussed above. Instead, they need only to reason about the structure-specific history invariant.

[2]See Section 9.2 for a detailed comparison of the two approaches.

6.4 Soundness of the hindsight specification

We establish the soundness of the hindsight specification by proving that any structure that satisfies the hindsight specification (HindSpec) must also satisfy its client-level specification (ClientSpec). Our proof that relates the two specifications involves a helping protocol stored as part of the shared state invariant. We begin by providing an overview of the helping protocol first. This is followed by details that make precies the role of prophecy variables. Finally, we provide the description of the shared state invariant encoding the helping protocol and an overview of the proof relating (HindSpec) to (ClientSpec).

Before op begins executing, the proof creates the prophecy resource $\mathsf{Proph}(p, pvs)$ assumed in the precondition of the hindsight specification. If the prophecy determines that the thread requires helping, then its client-level atomic triple is registered to a predicate which encodes the helping protocol as part of the shared state invariant $\mathsf{Inv}(r)$. The registered atomic triple serves as an obligation for the helping thread to commit the atomic triple. This obligation will be discharged by the appropriate concurrent operation determined by the op's sequential specification Ψ_{op}. The proof then uses the hindsight specification to conclude that it can collect the committed triple from the shared predicate. The committed triple serves as a receipt that the obligation to linearize has been fulfilled.

To govern the transfer of linearizability obligations and fulfillment receipts between threads via the shared invariant, the helping protocol tracks a *registry* of thread IDs with unmodifying linearization points that require helping from other concurrent threads. Each thread registered for helping is in either the *pending* state or the *done* state, depending on whether the thread has already been linearized. A thread registered for helping must be able to determine its current protocol state in order to be able to extract its committed atomic triple from the registry. For this purpose, the helping protocol includes a *linearization condition* that holds iff a registered thread *tid* has linearized (and is, hence, in *done* state).

From the point of view of a thread which *does* the helping, the linearization condition forces its proof to scan over the pool of uncommitted triples registered in the helping protocol and identify those that need to be linearized at its linearization point, changing their protocol state from *pending* to *done*. This step involves a proof obligation for the helping thread to show that the sequential specification of *tid*'s operation is indeed satisfied at the linearization point.

One crucial innovation in our helping protocol is that we have formulated a linearization condition that is parametric in the sequential specification of the data structure operations, making the soundness proof for the hindsight specification applicable to many structures at once. In particular, we deal with the aspect of scanning and updating the registry in the proof of the helping thread, the proof author simply invokes a lemma provided by our method at the identified linearization points. Therefore, the helping protocol mechanism remains fully opaque to the proof author.

```
1 let op̄ r =
2   let tid = NewThreadID in
3   let p = NewProph in
4   let v = op p r in
5   Resolve p to END(v); v
```

PROPHECY-CREATION
$\{\mathsf{True}\}\ \mathsf{NewProph}\ \{p.\ \exists\ pvs.\ \mathsf{Proph}(p, pvs)\}$

PROPHECY-RESOLUTION
$\{\mathsf{Proph}(p, pvs)\}\ \mathsf{Resolve}\ p\ \mathsf{to}\ v$
$\{\exists\ pvs'.\ \mathsf{Proph}(p, pvs')\ *\ pvs = v :: pvs'\}$

Figure 6.3: Wrapper augmenting op with prophecy-related ghost code, whose specification is given on the right.

6.4.1 Augmenting op with Prophecies

In light of the discussion in Section 6.3.1, we must augment op with auxiliary ghost code that creates and resolves the relevant prophecies. We do this by defining the wrapper function $\overline{\mathsf{op}}$ given in Figure 6.3. Let us first briefly discuss how prophecy variables can be used in Iris. The right side of Figure 6.3 shows the specifications of the two functions related to manipulating prophecies. The function NewProph returns a fresh prophecy p that predicts a sequence of values pvs, captured by the resource $\mathsf{Proph}(p, pvs)$. This resource can be owned and shared among threads via a shared invariant. The resource is also non-duplicable.

The values contained in the sequence pvs are determined by how p is resolved using the Resolve command. The rule PROPHECY-RESOLUTION ties the observed value (v) to the next prophesied value (the head of pvs). It also updates $\mathsf{Proph}(p, pvs)$ to the tail of pvs for the remaining prophesied values that are yet to be observed. In our context, we want the prophecy to predict whether the decisive operations (i.e. FAA in incr) will succeed as well as their return value. Hence, we augment incr by wrapping its decisive operations inside a Resolve command. The return value is captured by the resolution on Line 5. For our purposes, we assume that pvs is a sequence of Success or Failure values (i.e., the result of each attempted call to the decisive operation) followed by a special value $\mathsf{END}(res)$ indicating that no further future calls are expected, and that res will be the final return value[3]. Given the prophesied sequence of values pvs, a thread will exhibit an unmodifying linearization point iff pvs does not contain any Success values.

Let us now turn to $\overline{\mathsf{op}}$, which starts by generating a unique thread identifier using NewThreadID on Line 2. We also implement NewThreadID using the NewProph construct. There are two benefits to this: (i) Iris makes sure that prophecy identifiers are unique out of the box, and (ii) now we can use Iris's erasure theorem for prophecies to argue that the augmented code $\overline{\mathsf{op}}$ has the same behavior as op. After, $\overline{\mathsf{op}}$ creates a prophecy p as described above, it invokes the underlying operation op, and finally, terminates after resolving p to the result of op.

We can now finally present the client-level specification with $\overline{\mathsf{op}}$:

$$\boxed{\mathsf{Inv}(r)} \twoheadrightarrow \langle\, C.\ \mathsf{DS}(r, C)\,\rangle\ \overline{\mathsf{op}}\ p\ r\ \langle\, res\ C'.\ \mathsf{DS}(r, C')\ *\ \Psi_{\mathsf{op}}(C, C', res)\,\rangle.$$

[3]We have to also consider the case where pvs is not of this form, but that is a trivial case to handle and thus we do not consider it in this discussion.

The next section provides details on our choice of $\mathsf{Inv}(r)$ and the proof of $\overline{\mathsf{op}}$.

6.4.2 Invariant for Hindsight Reasoning

Tracking the history of computation. Hindsight arguments involve reasoning about past program states. Our encoding therefore tracks information about past states using *computation histories*. We define computation histories as finite partial maps from *timestamps*, $\mathbb{N}$, to *snapshots*, $\mathbb{S}$. A snapshot describes an abstract view of a program state. It is a parameter of our method. For instance, a snapshot may capture the physical memory representation of the data structure under proof, while abstracting from the remainder of the program state. Another parameter is a function $|\cdot|$ that computes the abstract state of the data structure from a given snapshot.

In order to track the history of computation, we rely on resource algebras. To reduce the burden on the proof author, Iris provides a library of predefined parameterized cameras. One such camera that we will use below is the *agreement camera* $\mathsf{Ag}(X)$ which is defined for any set X. The elements take the form $\mathsf{agree}(x)$ for $x \in X$ and composition is only defined in the case of identity: $\mathsf{agree}(x) \cdot \mathsf{agree}(x) = \mathsf{agree}(x)$. Another predefined camera we will use is that of finite partial maps from some set X to some camera R, $\mathsf{Map}(X, R)$, with composition defined by lifting the composition on R pointwise to maps. We also use the authoritative RA defined in Section 5.2. the camera $\mathsf{Auth}(\mathsf{MaxNat})$ is very useful to establish lower bounds of a monotonically non-decreasing quantity. The definitions of $\mathsf{Auth}(\mathsf{MaxNat})$ yields the valid entailment: $\boxed{\bullet\, \mathsf{Max}(n)}^{\gamma} * \boxed{\circ\, \mathsf{Max}(n')}^{\gamma} \vdash n' \leq n$. The permitted frame-preserving updates are shown below:

$$\textsc{auth-max-upd}\ \frac{n \leq n'}{\boxed{\bullet \mathsf{Max}(n)}^{\gamma} \Rrightarrow \boxed{\bullet \mathsf{Max}(n')}^{\gamma}}$$

$$\textsc{auth-max-snap}\ \boxed{\bullet \mathsf{Max}(n)}^{\gamma} \Rrightarrow \boxed{\bullet \mathsf{Max}(n)}^{\gamma} * \boxed{\circ \mathsf{Max}(n)}^{\gamma}$$

The rule AUTH-MAX-UPD requires that the update only increases the authoritative value to guarantee that the new authoritative value n' still composes with all fragmental values.

The predicate $\mathsf{Hist}(H, T)$ is then defined as

$$\begin{aligned}\mathsf{Hist}(H, T) \coloneqq \boxed{\bullet\, \mathsf{Max}(T)}^{\gamma_t} * \boxed{\bullet\, \mathsf{agree}(H)}^{\gamma_m} \\ * \operatorname{dom}(H) = \{0 \ldots T\} * \forall t < T.\, H(t) \neq H(t+1)\ .\end{aligned}$$

Here, we abuse notation and write $\mathsf{agree}(H)$ for the function that takes the history H to an element of $\mathsf{Auth}(\mathsf{Map}(\mathbb{N}, \mathsf{Ag}(\mathbb{S})))$ in the expected way. This resource allows us to define a *past predicate* $\diamondsuit_{s,t_0}(q)$ with the intuitive meaning that the history contains state s recorded after (or at) time t_0 for which proposition q holds true. The second last conjunct ensures that the authoritative history contains no gaps. The purpose of the last conjunct is to ensure that a thread appends to the history whenever any change to the program state is made that affects the snapshot (see discussion in Section 6.4.2).

$$
\begin{aligned}
\mathsf{Inv}(r) &\coloneqq \exists\, H\, T\, C.\ \overline{\mathsf{DS}}(r, C) * |H(T)| = C \\
&\quad * \mathsf{Hist}(H, T) \ * \ \mathsf{Inv}_{help}(H, T) \ * \ \mathsf{Inv}_{tpl}(r, H, T) \\
\mathsf{Inv}_{tpl}(r, H, T) &\coloneqq \mathsf{resources}(r, H(T)) \\
&\quad * (\forall t,\ 0 \leqslant t \leqslant T \Rightarrow \mathsf{per\_snapshot}(H(t))) \\
&\quad * (\forall t,\ 0 \leqslant t < T \Rightarrow \mathsf{transition\_inv}(H(t), H(t+1))) \\
\mathsf{Inv}_{help}(H, T) &\coloneqq \exists\, R.\ \boxed{\bullet\, R}^{\gamma_r} \\
&\quad * \mathop{\Large *}_{tid \in R} \ \exists\, \mathsf{op}\, v_p\, t_0\, \Phi\, \mathsf{Tok}.\ \mathsf{Reg}(tid, \mathsf{op}, v_p, t_0, \Phi, \mathsf{Tok}) \\
\mathsf{Reg}(tid, \mathsf{op}, v_p, t_0, \Phi, \mathsf{Tok}) &\coloneqq \mathsf{Thread}(tid, t_0) * \boxed{\mathsf{State}(\mathsf{op}, v_p, t_0, \Phi, \mathsf{Tok})} \\
\mathsf{State}(\mathsf{op}, v_p, t_0, \Phi, \mathsf{Tok}) &\coloneqq \mathsf{Pending}(\mathsf{op}, v_p, t_0, \Phi) \lor \mathsf{Done}(\mathsf{op}, v_p, t_0, \Phi, \mathsf{Tok}) \\
\mathsf{Pending}(\mathsf{op}, v_p, t_0, \Phi) &\coloneqq \mathsf{AU}_{\mathsf{op}}(\Phi) * \neg \mathsf{PastLin}(\mathsf{op}, v_p, t_0) \\
\mathsf{Done}(\mathsf{op}, v_p, t_0, \Phi, \mathsf{Tok}) &\coloneqq (\Phi(v_p) \ \lor \ \mathsf{Tok}) * \mathsf{PastLin}(\mathsf{op}, v_p, t_0)
\end{aligned}
$$

Figure 6.4: Definition of the shared state invariant encoding the hindsight reasoning. Variable H represents the history, T the current timestamp in use and C the abstract state of the structure.

Invariant. Figure 6.4 shows a simplified definition of the invariant that encodes the hindsight reasoning[4]. The conjunct $|H(T)| = C$ and the predicate $\overline{\mathsf{DS}}(r, C)$ together tie the abstract state C of the data structure to the latest snapshot in the history. The predicate $\overline{\mathsf{DS}}(r, C)$ is the dual of the representation predicate $\mathsf{DS}(r, C)$ used in the client-level atomic specification. Both represent one half of an ownership over the abstract state of the structure, keeping the abstract state defined by $\mathsf{Inv}(r)$ synchronized with the representation predicate $\mathsf{DS}(r, C)$.

The helping protocol predicate $\mathsf{Inv}_{help}(H, T)$ contains a *registry* $\boxed{\bullet\, R}^{\gamma_r}$ of thread IDs with unmodifying linearization points that require helping from other concurrent threads. For each thread ID *tid* in the registry, the shared state contains $\mathsf{Thread}(tid, \_)$ along with the state of *tid*, which is either Pending or Done. Pending captures an uncommitted atomic triple, and Done describes the operation after it has been committed. Note that we use the notation $\mathsf{AU}_{\mathsf{op}}(\Phi)$ to refer to the atomic update token of op and write just $\mathsf{AU}(\phi)$ when op is clear.

The predicate $\mathsf{Inv}_{tpl}(r, H, T)$ captures invariants particular to the data structure under proof. It is further composed of three abstract predicates that are meant to be instantiated with the structure-specific invariants. The three predicates serve the following purpose. The first predicate $\mathsf{resources}(r, H(T))$ ties the current snapshot to the physical representation of the structure. As an example, when verifying the distributed counter, this predicate

[4]For presentation purposes, the proof outline presented here abstracts from some technical details of the actual proof in Iris.

holds the points-to resources providing access to read and write over the two locations. The predicate $\mathsf{Hist}(H, T)$ contains a conjunct $(\forall t, t < T \Rightarrow H(t) \neq H(t+1))$. Together with the predicate resources, this conjunct forces a thread to update the history whenever the structure is modified. For this reason, the hindsight framework makes it mandatory to define this predicate. Other predicates in Inv_{tpl}, like per_snapshot and transition_inv explained below, are not mandatory but are typically useful.

The predicate $\mathsf{per\_snapshot}(H(T))$ captures the structural invariants that hold for any given snapshot. For instance with the distributed counter, this predicate holds facts such as that the abstract value of the counter is the sum of the values contained in the two locations. The predicate $\mathsf{transition\_inv}(s, s')$ captures a transition invariant on snapshots observed in the history. That is, it constrains how certain quantities evolve over time. Again as an example from the distributed counter, the facts about how values stored in two locations evolve according to Invariants 1 and 2 are stored here. Crucially, the facts in $\mathsf{transition\_inv}(s, s')$ allow temporal interpolation required to establish facts about past states in the history (like in Section 6.2).

To summarize, the proof author defines the snapshot of the structure, the function $|\cdot|$, and instantiates the three abstract predicates in Inv_{tpl} appropriately. The resulting shared state invariant then tracks the history and handles the helping protocol without requiring further fine-tuning to the data structure at hand.

6.4.3 Proof of $\overline{\mathtt{op}}$

We are now finally ready to prove the client-level specification for $\overline{\mathtt{op}}$. The proof outline for $\overline{\mathtt{op}}$ is shown in Figure 6.5. The proof begins by creating a thread identifier *tid* and prophecy p. As alluded to earlier, NewThreadID is also just NewProph in disguise, hence we obtain two prophecy resources, one each for *tid* and p. Before we can invoke op, we must record the start time of this thread and perform a case split on $\mathsf{Upd}(pvs)$ to determine if this thread requires helping. For this, we open the invariant at Line 7. To record the start time, we give up the resource $\mathsf{Proph}(tid, \_)$ to gain a duplicable resource $\mathsf{Thread}(tid, T_0)$ in exchange (T_0 is the current timestamp). The exact details of this exchange are not relevant, so we continue with the case split on $\mathsf{Upd}(pvs)$. First consider the case that $\mathsf{Upd}(pvs)$ holds. Then, thread *tid* does not require helping and the hindsight specification can ensures that op linearizes correctly. This is quite straightforward, so we focus on the other case, i.e. $\mathsf{Upd}(pvs)$ does not hold.

In order to register the thread for helping, we must establish $\mathsf{PastLin}(\mathtt{op}, v_p, T_0)$ or its negation. Here, v_p is the return value obtained by scanning for the end marker in *pvs*. We proceed by first checking if the thread can be linearized at the current timestamp T_0. If so, then we can safely linearize the thread right away. So let us focus on the more interesting case where we cannot linearize right away (Line 14). Here, the thread registers itself with the helping protocol by updating R to $R \cup \{tid\}$ using an update rule for the authoritative set camera (in Line 18). As part of registering for helping, it first establishes $\mathsf{Pending}(\mathtt{op}, v_p, T_0, \Phi)$ by transferring its obligation to linearize to the shared state, captured by the update token $\mathsf{AU}(\Phi)$. The condition $\mathsf{PastLin}(\mathtt{op}, v_p, T_0)$ follows from the fact that

$\boxed{\mathsf{Inv}(r)} * \langle\, C.\ \mathsf{DS}(r, C) \,\rangle$

let $\overline{\mathsf{op}}$ r =

$\{\mathsf{AU}(\Phi)\}$

let tid = NewThreadID

$\{\mathsf{AU}(\Phi) * \mathsf{Proph}(tid, \_)\}$

let p = NewProph **in**

$\{\mathsf{AU}(\Phi) * \mathsf{Proph}(tid, \_) * \mathsf{Proph}(p, pvs) * \boxed{\mathsf{Inv}(r)}\}$

(* Open invariant *)

$\{\mathsf{AU}(\Phi) * \mathsf{Thread}(tid, T_0) * \mathsf{Proph}(p, pvs) * \overline{\mathsf{DS}}(r, C_0) * \mathsf{Inv}_{help}(H_0, T_0) * \ldots\}$

(* Case analysis on $\mathsf{Upd}(pvs)$: only showing $\neg\mathsf{Upd}(pvs)$ *)

$\{\mathsf{AU}(\Phi) * \mathsf{Thread}(tid, T_0) * \mathsf{Proph}(p, pvs) * \overline{\mathsf{DS}}(r, C_0) * \boxed{\bullet\, R}^{\gamma_r} * tid \notin R * \ldots\}$

(* Let v_p such that pvs contains $\mathsf{END}(v_p)$ *)

(* Case analysis on $\Psi_{\mathsf{op}}(|H_0(T_0)|, |H_0(T_0)|, v_p)$: only showing $\neg\Psi_{\mathsf{op}}(|H_0(T_0)|, |H_0(T_0)|, v_p)$ *)

$\{\mathsf{Thread}(tid, T_0) * \mathsf{Proph}(p, pvs) * \overline{\mathsf{DS}}(r, C_0) * \boxed{\bullet\, R}^{\gamma_r} * tid \notin R\}$

$\{\cdots * \mathsf{AU}(\Phi) * \mathsf{Thread}(tid, T_0) * \neg\mathsf{PastLin}(\mathsf{op}, v_p, T_0)\}$

$\{\cdots * \mathsf{AU}(\Phi) * \mathsf{Thread}(tid, T_0) * \mathsf{Pending}(\mathsf{op}, v_p, T_0, \Phi) * \mathsf{Tok}\}$

$\{\cdots * \mathsf{State}(\mathsf{op}, v_p, T_0, \Phi, \mathsf{Tok}) * \mathsf{Tok}\}$

(* Ghost update: $\boxed{\bullet\, R}^{\gamma_r} \Rrightarrow \boxed{\bullet\, R \cup \{tid\}}^{\gamma_r}$ *)

$\{\mathsf{Thread}(tid, T_0) * \mathsf{Proph}(p, pvs) * \boxed{\circ\, \{tid\}}^{\gamma_r} * \mathsf{Tok} * \overline{\mathsf{DS}}(r, C_0) * \boxed{\bullet\, R \cup \{tid\}}^{\gamma_r} * \ldots\}$

$\{\mathsf{Thread}(tid, T_0) * \mathsf{Proph}(p, pvs) * \boxed{\circ\, \{tid\}}^{\gamma_r} * \mathsf{Tok} * \overline{\mathsf{DS}}(r, C_0) * \mathsf{Inv}_{help}(H_0, T_0) * \ldots\}$

(* Close invariant *)

$\{\mathsf{Thread}(tid, T_0) * \mathsf{Proph}(p, pvs) * \boxed{\circ\, \{tid\}}^{\gamma_r} * \mathsf{Tok}\}$

let v = op p r **in**

$\{\mathsf{Thread}(tid, T_0) * \boxed{\circ\, \{tid\}}^{\gamma_r} * \mathsf{Tok} * \mathsf{Proph}(p, pvs') * (pvs = prf \mathbin{+\!\!+} pvs') * (\mathsf{Upd}(pvs) \mathrel{-\!\!*} \Phi(v)) * (\neg\mathsf{Upd}(pvs) \mathrel{-\!\!*} \mathsf{PastLin}(\mathsf{op}, v_p, T_0))\}$

$\{\mathsf{Thread}(tid, T_0) * \boxed{\circ\, \{tid\}}^{\gamma_r} * \mathsf{Tok} * \mathsf{Proph}(p, pvs') * \mathsf{PastLin}(\mathsf{op}, v_p, T_0)\}$

Resolve p to $\mathsf{END}(v)$;

$\{\mathsf{Thread}(tid, T_0) * \boxed{\circ\, \{tid\}}^{\gamma_r} * \mathsf{Tok} * \mathsf{Proph}(p, []) * (v_p = v) * \mathsf{PastLin}(\mathsf{op}, v_p, T_0)\}$

(* Open invariant *)

$\{\mathsf{Thread}(tid, T_0) * \boxed{\circ\, \{tid\}}^{\gamma_r} * \mathsf{Tok} * \mathsf{PastLin}(\mathsf{op}, v, T_0) * \mathsf{DS}(r, C_1) * \mathsf{Inv}_{help}(H_1, T_1) * \ldots\}$

$\{\cdots * \mathsf{Tok} * \boxed{\circ\, \{tid\}}^{\gamma_r} * \mathsf{PastLin}(\mathsf{op}, v, T_0) * \mathsf{State}(\mathsf{op}, v, T_0, \Phi, \mathsf{Tok})\}$

$\{\cdots * \mathsf{Tok} * \mathsf{PastLin}(\mathsf{op}, v, T_0) * \mathsf{Done}(\mathsf{op}, v, T_0, \Phi, \mathsf{Tok})\}$

$\{\cdots * \mathsf{Tok} * \mathsf{PastLin}(\mathsf{op}, v, T_0) * (\Phi(v) \vee \mathsf{Tok})\}$

$\{\cdots * \Phi(v) * \mathsf{PastLin}(\mathsf{op}, v, T_0) * (\Phi(v) \vee \mathsf{Tok})\}$

$\{\cdots * \Phi(v) * \mathsf{PastLin}(\mathsf{op}, v, T_0) * \mathsf{Done}(\mathsf{op}, v, T_0, \Phi, \mathsf{Tok})\}$

$\{\Phi(v) * \mathsf{DS}(r, C_1) * \mathsf{Inv}_{help}(H_1, T_1) * \ldots\}$

(* Close invariant *)

$\{\Phi(v)\}$

v

$\langle\, res\ C'.\ \mathsf{DS}(r, C') * \Psi_{\mathsf{op}}(C, C', res) \,\rangle$

Figure 6.5: Outline for the proof of the client-level specification for op.

$\Psi_{\mathsf{op}}(k, |H_0(T_0)|, |H_0(T_0)|, v_p)$ does not hold. The thread also creates a fresh non-duplicable token Tok that it will later trade in for the receipt $\Phi(v_p)$.

We are now at the point of invocation for op. Before continuing further, let us briefly switch to the role played by the concurrent thread that must linearize thread *tid*. The helping thread must update the structure, and as per the invariant, update the history M as well as the helping protocol $\mathsf{Inv}_{help}(H, T)$. In particular, while updating the helping protocol, it scans over all threads registered for helping so far, moving them from state Pending to state Done as per the prophesied return value v_p.

Let us now return to the proof thread *tid* at Line 23. As precondition of (HindSpec), we give away $\mathsf{Proph}(p, pvs)$ and obtain the postcondition (Line 24). Simplifying the post-condition of (HindSpec) for our case $\neg\mathsf{Upd}(pvs)$, we obtain the predicate $\mathsf{PastLin}(\mathsf{op}, v_p, T_0)$. Next, we make a final resolution of the prophecy on Line 26. Since, pvs' is the suffix of pvs, $\mathsf{END}(v_p)$ must also be the final item in pvs'. After resolution, we obtain $\mathsf{END}(v_p) = \mathsf{END}(v)$ and by injectivity of the end marker, $v_p = v$.

We now have the predicate $\mathsf{PastLin}(\mathsf{op}, v, T_0)$ which we use to obtain the receipt of linearization from the shared invariant. Because $\mathsf{PastLin}(\mathsf{op}, v, T_0)$ holds now, we know thread *tid* cannot be in the Pending state. Hence, we know that thread *tid* must be in the Done state. Since the thread owns the unique token Tok, it trades it in to obtain $\Phi(v)$, which lets it complete the proof of its client-level specification.

7 | Multicopy Structures

This chapter introduces the LSM-DAG template that covers lock-based multicopy structures like LSM Trees. We provide the intuitive proof of correctness for the template using hindsight reasoning. Following this, we show how to use the hindsight framework from Chapter 6 to prove linearizability of the LSM-DAG Template.

7.1 Introduction

Single-copy structures achieve high performance for reads. However, some applications, such as event logging, require high write performance, possibly at the cost of decreased read speed and increased memory overhead. This demand is met by data structures that store upserts (inserts, deletes or updates) to a key k *out-of-place* at a new node instead of overwriting a previous copy of k that was already present in some other node. Performing out-of-place upserts can be done in constant time (e.g., always at the head of a list). A consequence of this design is that the same key k can now be present multiple times simultaneously in the data structure. Hence, we refer to these structures as *multicopy (search) structures*.

Examples of multicopy structures include the differential file structure [152], the log-structured merge (LSM) tree [134], and the Bw-tree [109]. These concurrent data structures are widely used in practice, including in state-of-the-art database systems such as Apache Cassandra [4] and Google LevelDB [58].

Like the verification method proposed by Krishna et al. [97], we aim to prove that the concurrent search structure of interest is linearizable [69], i.e., each of its operations appears to take effect atomically at a *linearization point* and behaves according to a sequential specification. For multicopy structures, the sequential specification is that of a (partial) mathematical map that maps a key to the last value that was upserted for that key. The framework proposed in [97, 153] does not extend to multicopy structures as it critically relies on the fact that every key is present in at most one node of the data structure at a time. Moreover, searches in multicopy structures exhibit dynamic non-local linearization points (i.e., the linearization point of a search is determined by and may be present during the execution of concurrently executing upserts). This introduces a technical challenge that is not addressed by this prior work. We discuss further related work in Chapter 10.

In this chapter, we introduce template algorithms for multicopy structures. We analyse them by first providing intuitive proof of their linearizability. This is followed by connecting

the intuitive proof to the hindsight framework from Chapter 6.

7.2 Motivation and Overview

From a client's perspective, a multicopy structure implements a partial mathematical map $M\colon \mathbb{K} \rightharpoonup \mathsf{V}$ of keys $k \in \mathbb{K}$ to values $v \in \mathsf{V}$. We refer to M as the *logical contents* of the structure. The data structure supports insertions and deletions of key/value pairs on M and searches for the value $M(k)$ associated with a given key k.

The insert and delete operations are implemented by a single generic operation referred to as an *upsert*. The sequential specification of upsert is as follows. The operation takes a key-value pair (k, v) and updates M to $M[k \rightarrowtail v]$, associating k with the given value v. To delete a key k from the structure, one upserts the pair $(k, \square)$ where $\square$ is a dedicated *tombstone* value used to indicate that k has been deleted. The sequential specification of a search for a key k is then as expected: it returns $M(k)$ if M is defined for k and $\square$ otherwise.

Multicopy structures are commonly used in scenarios where the nodes representing the data structure's logical contents M are spread over multiple media such as memory, solid-state drives, and hard disk drives. Each node therefore contains its own data structure that is designed for the particular characteristics of the underlying medium, typically an unsorted array at the root to allow upserts to perform fast appends and a classical single-copy search structure (e.g., a hash structure or arrays with bloom filters) for non-root nodes. The non-root nodes are typically read-only, so concurrency at the node level is not an issue. In this dissertation, we consider the multicopy data structure as a graph of nodes. We study template algorithms on that graph.

7.2.1 A Library Analogy to Multicopy Search Structures

To train your intuition about multicopy structures, consider a library of books in which new editions of the same book arrive over time. Thus the first edition of book k can enter and later the second edition, then the third and so on. A patron of this library who enters the library at time t and is looking for book k should find an edition that is either current at time t or one that arrives in the library after t. We call this normative property *search recency*.

Now suppose the library is organized as a sequence of rooms. All new books are put in the first room (near the entrance). When a new edition v of a book arrives in the first room, any previous editions of that book in that room are thrown out. When the first room becomes full, the books in that room are moved to the second room. If a previous edition of some book is already in the second room, that previous edition is thrown out. When the second room becomes full, its books are moved to the third room using the same throwing out rule, and so on. This procedure maintains the time-ordering invariant that the editions of the same book are ordered from most recent (at or nearer to the first room) to least recent (farther away from the first room) in the sequence of rooms.

A patron's search for k starting at time t begins in the first room. If the search finds any edition of k in that room, the patron takes a photocopy of that edition. If not, the search

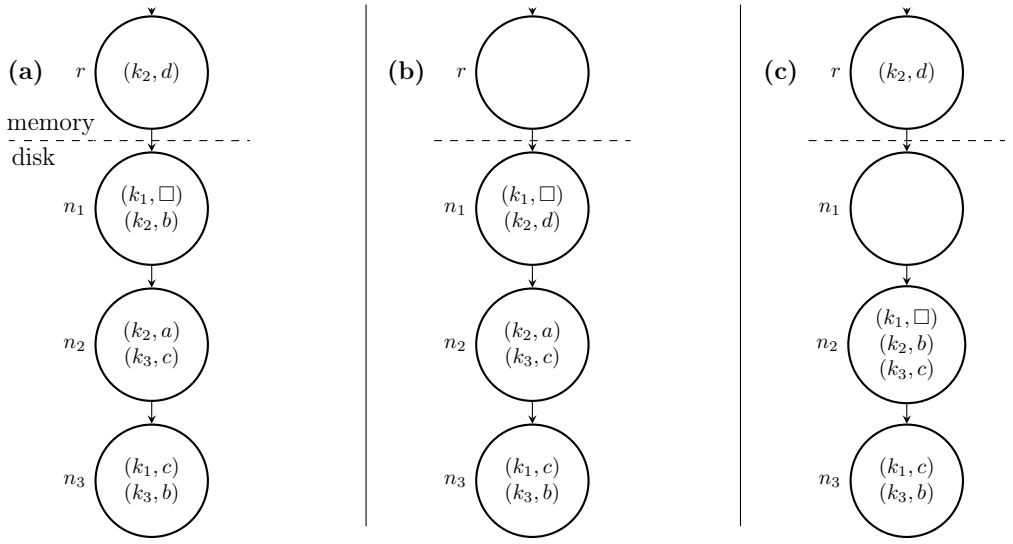


Figure 7.1: LSM tree representations. **(a)** High-level structure of an LSM tree. **(b)** LSM tree obtained from (a) after flushing node r to disk. **(c)** LSM tree obtained (a) after compacting nodes n_1 and n_2.

proceeds to the second room and so on.

Now suppose that the latest edition at time t is edition v and there is a previous edition v'. Because of the time-ordering invariant and the fact that the search begins at the first room, the search will encounter v before it encounters v'. The search may also encounter an even newer edition of k, but will never encounter an older one before returning. That establishes the search recency property.

Any concurrent execution of inserts and searches is equivalent to a serial execution in which (i) each insert is placed in its relative order of entering the root node with respect to other inserts and (iia) a search s is placed after the insert whose edition s copies if that insert occurred after s began or (iib) a search s is placed at the point when s began, if the edition that s copies was inserted before s began (or if s returns no edition at all).

Because the searches satisfy the search recency property, the concurrent execution is *linearizable* [70], which is our ultimate correctness goal.

Note that the analogy as written has treated only inserts and searches. However, updates and deletions can be implemented as inserts: an update to book k can be implemented as the insertion of a new edition; a delete of book k can be implemented as the insertion of an edition whose value is a "tombstone" which is an indication that book k has been deleted.

7.2.2 Log-Structured Merge Trees

A prominent example of a multicopy structure is the LSM tree, which closely corresponds to the library analogy described above. The data structure consists of a root node r stored in memory (the first room in the library), and a linked list of nodes $n_1, n_2, \ldots, n_l$ stored on disk (the remaining rooms). Figure 7.1 (a) shows an example.

The LSM tree operations essentially behave as outlined in the library analogy. The upsert

operation takes place at the root node r. A search for a key k traverses the list starting from the root node and retrieves the value associated with the first copy of k that is encountered. If the retrieved value is $\square$ or if no entry for k has been found after traversing the entire list, then the search determines that k is not present in the data structure. Otherwise, it returns the retrieved value. For instance, a search for key k_1 on the LSM tree depicted in Figure 7.1 (a) would determine that this key is not present since the retrieved value is $\square$ from node n_1. Similarly, k_4 is not present since there is no entry for this key. On the other hand, a search for k_2 would return d and a search for k_3 would return c.

To prevent the root node from growing too large, the LSM tree performs *flushing*. As the name suggests, the flushing operation flushes the data from the root node to the disk by moving its contents to the first disk node. Figure 7.1 (b) shows the LSM tree obtained from Figure 7.1 (a) after flushing the contents of r to the disk node n_1.

Similar to flushing, a *compaction* operation moves data from a full node on disk to its successor. In case there is no successor, then a new node is created at the end of the structure. During the merge, if a key is present in both nodes, then the most recent (closer-to-the-root) copy is kept, while older copies are discarded. Figure 7.1 (c) shows the LSM tree obtained from Figure 7.1 (a) after compacting nodes n_1 and n_2. Here, the copy of k_2 in n_2 has been discarded. In practice, the length of the data structure is bounded by letting the size of newly created nodes grow exponentially.

The net effect of all these operations is that the data structure satisfies the time-ordering invariant and searches achieve search recency.

The LSM tree can be tuned by implementing workload- and hardware-specific data structures at the node level. In addition, research has been directed towards optimizing the layout of nodes and developing different strategies for the maintenance operations used to reorganize these data structures. This has resulted in a variety of implementations today (e.g. [35, 112, 143, 160, 169]). Despite the differences between these implementations, they generally follow the same high-level algorithms for the core search structure operations.

We construct template algorithms for concurrent multicopy structures from the high-level descriptions of their operations and then prove the correctness of these operations. Notably our LSM DAG template generalizes the LSM tree so that the outer data structure can be a DAG rather than just a list. A number of existing LSM structures are based on trees (e.g. [148, 169]). Practical implementations of tree-based concurrent search structures often have additional pointer structures layered on top of the tree that make them DAGs. For instance, many implementations use the *link technique* to increase performance. Here, when a maintenance operation relocates a key k from one node to another, it adds a pointer linking the two nodes, which ensures that k remains reachable via the old search path. A concurrent thread searching for k that arrives at the old node can then follow the link, avoiding a restart of the search from the root. Our verified templates can be instantiated to lock-based implementations of this technique.

7.3 Multicopy Search Structures

We abstract away from the data organization within the nodes, and treat the data structure as consisting of nodes in a mathematical directed acyclic graph.

Since copies of a single key k can be present in different nodes simultaneously, we need a mechanism to differentiate among these copies. To that end, we augment each entry (k, v) stored in a node with the unique timestamp t identifying the point in time when (k, v) was upserted: $(k, (v, t))$. The timestamp plays the role of the book edition in the library analogy from the last section. For example in Figure 7.2, (k_3, c) was upserted after (k_2, a), which was upserted after (k_3, b). These timestamps are derived from the timestamps used to store computation history in the hindsight framework from Section 6.4.2. Note that the timestamp associated with an upserted value is auxiliary, or *ghost*, data that we use in our proofs to track the temporal ordering of the copies present in the structure at any point. Implementations do not need to explicitly store this timestamp information.

Formally, let $\mathbb{K}$ be the set of all keys and V a set of values with a dedicated tombstone value $\square \in \mathsf{V}$. A multicopy (search) structure is a directed acyclic graph $G = (N, E)$ with nodes N and edges $E \subseteq N \times N$. We assume that there is a dedicated *root node* $r \in N$ which uniquely identifies the structure.

Each node n of the graph is labeled by its contents $C_n \colon \mathbb{K} \rightharpoonup \mathsf{V} \times \mathbb{N}$, which is a partial map from keys to pairs of values and timestamps. For a node n and its contents C_n, we say $(k, (v, t))$ is in the contents of n if $C_n(k) = (v, t)$. We denote the absence of an entry for a key k in n by $C_n(k) = \bot$ and let $\mathrm{dom}(C_n) \coloneqq \{k \mid C_n(k) \neq \bot\}$. We further write $\mathsf{val}(()C_n) \colon \mathbb{K} \rightharpoonup \mathsf{V}$ for the partial function that strips off the timestamp information from the contents of a node, $\mathsf{val}(C_n) \coloneqq \lambda k.\ (\exists v.\, C_n(k) = (v, \_)\ ?\ v : \bot)$.

For each edge $(n, n') \in E$ in the graph, the *edgeset* $\mathsf{es}(n, n')$ is the set of keys k for which an operation arriving at a node n would traverse (n, n') if $k \notin \mathrm{dom}(C_n)$. We require that the edgesets of all outgoing edges of a node n are pairwise disjoint. Figure 7.2 shows a potential abstract multicopy structure graph consistent with the LSM tree depicted in Figure 7.1 (a). Here, all edges have edgeset $\mathbb{K}$.

A multicopy structure abstractly represents a map ADT, i.e., the *logical contents* of the data structure is a mathematical map from keys to values, $M \colon \mathbb{K} \to \mathsf{V}$. The map M associates every key k with the most recently upserted value v for k, respectively, $\square$ if k has not yet been upserted:

$$M(k) \coloneqq \begin{cases} v & \text{if } \exists n\, t.\ C_n(k) = (v, t) \wedge t = \max\{t' \mid \exists n'\, v'.\ C_{n'}(k) = (v', t')\} \\ \square & \text{otherwise} \end{cases}$$

We call $M(k)$ the *logical value* of key k.

7.3.1 The LSM DAG Template

This section presents a general template for multicopy structures that generalizes the LSM (log-structured merge) tree discussed in §7.2.2. We prove linearizability of the template by

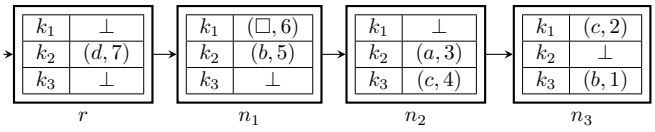


Figure 7.2: Abstract multicopy data structure graph for the LSM tree in Figure 7.1 (a).

```
let rec traverse r n k =
  lockNode n;
  match inContents r n k with
  | Some v -> unlockNode n; v
  | None ->
    match findNext r n k with
    | Some n' ->
      unlockNode n;
      traverse r n' k
    | None -> unlockNode n; □

let search r k = traverse r r k
```

```
let rec upsert r k v =
  lockNode r;
  let res = addContents r k v in
  if res then
    unlockNode r
  else begin
    unlockNode r;
    upsert r k v
  end
```

Figure 7.3: The LSM-DAG template for multicopy operations search and upsert. The template can be instantiated by providing implementations of helper functions inContents, findNext, and addContents. inContents $r\,n\,k$ returns Some v if $(v,t') = C_n(k)$ for some t', and None otherwise. findNext $r\,n\,k$ returns Some n' if n' is the unique node such that $k \in \mathsf{es}(n,n')$, and None otherwise. addContents $r\,k\,v$ updates the contents of r by setting the value associated with key k to v. The return value of addContents is a Boolean which indicates whether the insertion was successful (e.g., if r is full, insertion may fail leaving r's contents unchanged).

verifying that all operations satisfy the client-level atomic triples (ClientSpec). The template and the proof abstract away the implementation of the single-copy data structures used at the node-level. Instantiating the template for a specific implementation involves only sequential reasoning about the implementation-specific node-level operations.

We split the template into two parts. The first part is a template for search and upsert that works on general multicopy structures, i.e., arbitrary DAGs with locally disjoint edge-sets. The second part (discussed in §7.7) is a template for a maintenance operation that generalizes the compaction mechanism found in existing list-based LSM tree implementations to tree-like multicopy structures.

Figure 7.3 shows the code of the template for the core multicopy operations. The operations search and upsert closely follow the high-level description of these operations on the LSM tree (§7.2.2). The operations are defined in terms of implementation-specific helper functions findNext, addContents, and inContents.

The search operation calls the recursive function traverse on the root node. Function traverse $r\,n\,k$ first locks the node n and uses the helper function inContents $r\,n\,k$ to check if a copy of key k is contained in n. If a copy of k is found, then its associated value v is returned after unlocking n. Otherwise, traverse uses the helper function findNext to determine the unique successor n' of the given node n and query key k (i.e., the node n'

satisfying $k \in \mathsf{es}(n, n')$). If such a successor n' exists, `traverse` recurses on n'. Otherwise, `traverse` concludes that there is no copy of k in the data structure and returns $\square$. Note that this algorithm uses fine-grained concurrency, as the thread executing the `search` holds at most one lock at any point (and no locks at the points when `traverse` is called recursively).

The `upsert` r k v operation locks the root node and adds a new copy of the key k with value v to the contents of the root node using `addContents`. `addContents` r k v adds the pair (k, v) to the root node when it succeeds. `upsert` terminates by unlocking the root node. The `addContents` function may however fail if the root node is full. In this case `upsert` calls itself recursively[1].

7.4 Intuitive Proof Argument

We next discuss the correctness proof of the template operations. We will focus on the high-level proof ideas and key invariants and defer the detailed proof outline and encoding of the invariants in Iris to the later sections.

Our goal is to prove the linearizability of concurrent multicopy structure templates with respect to their desired sequential client-level specification. As discussed earlier, the sequential specification is that of a map ADT as shown in Figure 7.4.

$$\Psi_{\mathsf{op}}(M, M', res) \coloneqq \begin{cases} M' = M \land res = M(k) & \mathsf{op} = \mathtt{search}(k) \\ M' = M[k \rightarrowtail v] & \mathsf{op} = \mathtt{upsert}(k,\ v) \end{cases}$$

Figure 7.4: Sequential specification of a multicopy search structure as a Map ADT. k refers to the operation key, v to the upsert value and M and M' to the abstract state before and after operation op, respectively, and res is the return value of op.

Recall that the client-level atomic specification (ClientSpec) that corresponds to linearizability is as follows:

$$\boxed{\mathsf{Inv}(r)} \twoheadrightarrow \langle\, M.\ \mathsf{MCS}(r, M) \,\rangle\ \mathsf{op}\ r\ \langle\, res.\ \exists\ M'.\ \mathsf{MCS}(r, M') * \Psi_{\mathsf{op}}(M, M', res) \,\rangle. \tag{7.1}$$

Here, MCS is a representation predicate denoting the abstract state M for the multicopy structure rooted at r. In order to prove the atomic triple, we must determine the atomic step for each operation at which the operation take effect. In other words, the operation's linearization point. As we have seen with the distributed counter data structure in Chapter 6 and the Skiplist template in Chapter 8, it is straightforward to determine the linearization point for the operation which changes the abstract state. In case of the multicopy structure, it is the `upsert` operation that changes the abstract state. Its linearization point is when the

[1]For simplicity of presentation, we assume that a separate maintenance thread flushes the root if it is full to ensure that upserts eventually make progress.

Figure 7.5: Problem execution for the LSM-DAG template. Sub-figures show possible states of `search(`k`)` in the presence of interference from concurrent `upsert(`k`, `v_1`)` and `upsert(`k`, `v_2`)`.

new copy of the key is successfully added to the root node. However, the `search` operation exhibits future-dependent linearization points. Let us explain this in further detail.

Consider a multicopy structure depicted in Figure 7.5 (S1) as its initial state. For simplicity, assume that the structure handles operations on a single key k. The structure contains only two nodes: the root node r and a disk node n. Let there be thread T_s executing `search(`k`)` concurrently with threads T_1 and T_2 executing `upsert(`k`, `v_1`)` and `upsert(`k`, `v_2`)` respectively. There is also a maintenance thread running in the background which flushes data from node r to n. Consider the following interleaving steps of concurrent execution of T_s, T_1 and T_2:

(S1) T_s finds no copy of k in r and arrives at n, but has not locked n yet.

(S2) T_1 upserts copy $(k, (v_1, 2))$ at r.

(S3) The maintenance thread flushes copy $(k, (v_1, 2))$ from r to n, throwing away the copy $(k, (v_0, 1))$.

(S4) T_1 upserts copy $(k, (v_2, 3))$ at r.

(S5) T_s finally locks n, finds copy $(k, (v_1, 2))$ and returns it.

Figure 7.5 depicts the state of the multicopy at the end of each step. In the above concurrent execution, the linearization point for T_s lies after step (S2), right after copy $(k, (v_1, 2))$ is added and which T_s finds eventually. However, if T_s is able to obtain the lock on n right after step (S1), then it returns the copy $(k, (v_0, 1))$. In this case, the linearization point of T_s would be at the beginning of its execution. Thus, the linearization point of T_s is dependent on the concurrent operations, and it can be determined only by judging the copy of k returned by T_s.

The intuitive proof argument for linearizability of `search` relies on the observation of search recency: each concurrent invocation `search(`k`)` either returns the logical value associated with k at the point when the search started, or any other copy of k that was upserted between the search's start time and the search's end time. Using the above concurrent execution as an example, the logical value of k when T_s started in (S1) is $(k, (v_0, 1))$. Search recency says that T_s either returns $(k, (v_0, 1))$, or a newer copy (such as $(k, (v_1, 2))$).

Search recency provides a strategy for determining the linearization point for `search`. If an invocation of `search(`k`)` returns the logical value associated with k when the search started, then it can be linearized at the beginning of the invocation. Otherwise, search finds a newer copy of k upserted by a concurrent upsert. Here, its linearization point is the point when the newer copy of k was upserted. Note that the linearization point can only be determined in hindsight, once search returns.

We next expand on the intuitive argument above by making concrete why multicopy structures satisfy search recency. In the next section, we will show that if searches satisfy search recency and upserts take effect in a single atomic step that changes the logical contents according to the sequential specification in Figure 7.4, then the multicopy structure satisfies the hindsight specification described in Section 6.3.1. This establishes the linearizability of the concurrent multicopy structures.

7.5 Search Recency

Keeping Track of Upsert History. The logical contents of a multicopy structure only stores knowledge about the latest copy of each key. However, in order to express search recency as a property we must additionally view a multicopy structure in terms of its *upsert history* $U \subseteq \mathbb{K} \times (\mathsf{V} \times \mathbb{N})$ as the set of all copies $(k, (v, t))$ that have been upserted thus far. In particular, we require that any multicopy structure will maintain the following predicates concerning U and the global clock t:

$$\mathsf{Init}(U) \coloneqq \forall k.\ (k, (\square, 0)) \in U$$

$$\mathsf{HUnique}(U) \coloneqq \forall k\, t'\, v_1\, v_2.\ (k, (v_1, t')) \in U \wedge (k, (v_2, t')) \in U \Rightarrow v_1 = v_2$$

$$\mathsf{MaxTS}(t, U) \coloneqq \forall (k, (\_, t')) \in U.\ t' < t$$

The predicate $\mathsf{HUnique}(U)$ ensures that we can lift the total order $t_1 \leqslant t_2$ on timestamps to a total order $(v_1, t_1) \leqslant (v_2, t_2)$ on the pairs of values and timestamps occurring in U. The lifted order simply ignores the value component. Together with $\mathsf{Init}(U)$, this ensures that the following function is well-defined:

$$\bar{U}(k) \coloneqq \max \{(v, t) \mid (k, (v, t)) \in U\} = (M(k), \_).$$

The latest copy of a key will always be contained in some node n of the data structure. If the data structure implementation maintains the additional invariant, $U \supseteq \bigcup_{n \in N} C_n$, then this guarantees that $\bar{U}$ is consistent with the logical contents M, i.e., for all keys k, $\bar{U}(k) = (M(k), \_)$. Finally, the predicate $\mathsf{MaxTS}(t, U)$ guarantees that $\mathsf{HUnique}(U)$ is preserved when a new entry $(k, (v, t))$ is added to U for the current value of the global clock t.

Proof of Search Recency. Using upsert history U, we can state search recency as the following property: if t_0 is the logical timestamp of k at the point when `search` r k is invoked,

then the operation returns v such that $(k, (v, t')) \in U$ and $t' \geqslant t_0$. Since the value v of k retrieved by `search` comes from some node in the structure, we must examine the relationship between the upsert history U of the data structure and the physical contents C_n of the nodes n visited as the search progresses. We do this by identifying the main invariants needed for proving search recency for arbitrary multicopy structures.

We refer to the *spatial ordering* of the copies $(k, (v, t))$ stored in a multicopy structure as the ordering in which those copies are reached when traversing the data structure graph starting from the root node. Our first observation is that the spatial ordering is consistent with the temporal ordering in which the copies have been upserted. We referred to this property as the time-ordering invariant in our library analogy in §7.2.1: the farther from the root a search is, the older the copies it finds are. Therefore, if a `search` r k traverses the data structure without interference from other threads and returns the first copy of k that it finds, then it is guaranteed to return the logical value of k at the start of the search.

We formalize this observation in terms of the *contents-in-reach* of a node. The contents-in-reach of a node n is the partial function $C_{ir}(n) \colon \mathbb{K} \rightharpoonup \mathsf{V} \times \mathbb{N}$ defined recursively over the graph of the multicopy structure as follows:

$$C_{ir}(n)(k) \coloneqq \begin{cases} C_n(k) & \text{if } k \in \mathrm{dom}(C_n) \\ C_{ir}(n')(k) & \text{else if } \exists n'.\, k \in \mathsf{es}(n, n') \\ \bot & \text{otherwise} \end{cases} \tag{7.2}$$

Note that $C_{ir}(n)$ is well-defined because the graph is acyclic and the edgesets labeling the outgoing edges of every node n are disjoint. We further define $\mathsf{ts}(C_{ir}(n)(k)) = t$ if $C_{ir}(n)(k) = (\_, t)$ and $\mathsf{ts}(C_{ir}(n)(k)) = 0$ if $k \notin \mathrm{dom}(C_{ir}(n))$.

For example, in the multicopy structure depicted in Figure 7.2, we have $C_{ir}(r) = \{k_1 \rightarrowtail (\square, 6), k_2 \rightarrowtail (d, 7), k_3 \rightarrowtail (c, 4)\}$ and $C_{ir}(n_3) = C_{n_3}$.

The observation that interference-free searches will find the current logical timestamp of their query key is then captured by the following invariant:

Invariant 1 The logical contents of the multicopy structure is the contents-in-reach of its root node: $\bar{U} = C_{ir}(r)$.

In order to account for concurrent threads interfering with the search, we prove the condition $t_0 \leq t'$ for the timestamp t' associated with the value returned by the search. Intuitively, this is true because the contents-in-reach of a node n can be affected only by upserts or maintenance operations, both of which only increase the timestamps associated with every key of any given node: upserts insert new copies into the root node and maintenance operations move recent copies down in the structure, possibly replacing older copies. This observation is formally captured by the following invariant:

Invariant 2 The contents-in-reach of every node only increases. That is, for every node n and key k, if $\mathsf{ts}(C_{ir}(n)(k)) = t$ at some point in time and $\mathsf{ts}(C_{ir}(n)(k)) = t'$ at any later point in time, then $t \leq t'$.

Finally, in order to prove the condition $(k, (v, t')) \in U$ of search recency, we need one additional property:

Invariant 3 All copies present in the multicopy structure have been upserted at some point in the past. That is, for all nodes n, $C_n \subseteq U$.

Now let us consider an execution of `search` on a operation key k. In addition to the above three general invariants, we need an inductive invariant for the traversal performed by the search: we require as a precondition for $\texttt{traverse}\, r\, n\, k$ that $\mathsf{ts}(C_{ir}(n)(k)) \geqslant t_0$ where t_0 is the timestamp of the logical value v_0 of k at the point when `search` was invoked. To see that this property holds initially for the call to $\texttt{traverse}\, r\, r\, k$ in `search`, let $\bar{U}_0$ be the logical contents at the time point when `search` was invoked. The precondition $\mathsf{SR}(k, v_0, t_0)$ implies $\mathsf{ts}(\bar{U}_0(k)) \geqslant t_0$, which, combined with Invariant 1 implies that we must have had $\mathsf{ts}(C_{ir}(r)(k)) \geqslant t_0$ at this point. Since $\mathsf{ts}(C_{ir}(r)(k))$ only increases over time because of Invariant 2, we can conclude that $\mathsf{ts}(C_{ir}(r)(k)) \geqslant t_0$ when `traverse` is called. We next show that the traversal invariant is maintained by `traverse` and is sufficient to prove search recency.

Consider a call to $\texttt{traverse}\, r\, n\, k$ such that $\mathsf{ts}(C_{ir}(n)(k)) \geqslant t_0$ holds initially. We must show that the call returns v such that $(k, (v, t')) \in U$ and $t' \geqslant t_0$ for some t'. We know that the call to `inContents` on line 3 returns either $\texttt{Some}\, v$ such that $(v, t') = C_n(k)$ or `None` if $C_n(k) = \bot$. Let us first consider the case where `inContents` returns $\texttt{Some}\, v$. In this case, `traverse` returns v on line 4. By definition of $C_{ir}(n)$ we have $C_{ir}(n)(k) = C_n(k)$. Hence, we have $\mathsf{ts}(C_{ir}(n)(k)) = t'$ and the precondition $\mathsf{ts}(C_{ir}(n)(k)) \geqslant t_0$, together with Invariant 2, implies $t' \geqslant t_0$. Moreover, Invariant 3 guarantees $(k, (v, t')) \in U$.

Now consider the case where `inContents` returns `None`. Here, $k \notin \mathrm{dom} C_n(k)$, indicating that no copy has been found for k in n. In this case, `traverse` calls `findNext` to obtain the successor node of n and k. In the case where the successor n' exists (line 7), we know that $k \in \mathsf{es}(n, n')$ must hold. Hence, by definition of contents-in-reach we must have $C_{ir}(n)(k) = C_{ir}(n')(k)$. From $\mathsf{ts} C_{ir}(n)(k) \geqslant t_0$ and Invariant 2, we can then conclude $\mathsf{ts}(C_{ir}(n')(k)) \geqslant t_0$, i.e. that the precondition for the recursive call to `traverse` on line 9 is satisfied and search recency follows by induction.

On the other hand, if n does not have any next node, then `traverse` returns $\square$ (line 10), indicating that k has not yet been upserted at all so far (i.e., has never appeared in the structure). In this case, by definition of contents-in-reach we must have $C_{ir}(n)(k) = \bot$. Invariant 2 then guarantees $\mathsf{ts}(C_{ir}(n)k) = 0 = t_0$. The invariant $\mathsf{Init}(U)$ on the upsert history then gives us $(k, (\square, 0)) \in U$. Hence, search recency holds in this case for $t' = 0$.

Proof of `upsert`. In order to prove the logically atomic specification (ClientSpec) of `upsert`, we must identify an atomic step where the astract state is modified. Intuitively, this atomic step is when the `addContents` succeeds at the root node (line 17 in Figure 7.3). Note that in this case `addContents` changes the contents of the root node from C_r to $C_r' = C_r[k \rightarrowtail (v, t)]$. Hence, in the proof we need to update the ghost state for the upsert history from U to $U' = U \cup \{(k, (v, t))\}$, reflecting that a new copy of k has been upserted. It then remains to show that the three key high-level invariants of multicopy structures identified above are preserved by these updates.

First, observe that Invariant 3, which states $\forall n.\ C_n \subseteq U$, is trivially maintained: only C_r is affected by the upsert and the new copy $(k, (v, t))$ is included in U'. Similarly, we can easily show that Invariant 2 is maintained: $C_{ir}(n)$ remains the same for all nodes $n \neq r$ and for the root node it increases, provided Invariant 1 is also maintained.

Thus, the interesting case is Invariant 1. Proving that this invariant is maintained amounts to showing that $\bar{U}'(k) = (v, t)$. This step critically relies on the following additional observation:

Invariant 4 All timestamps in U are smaller than the current time of the global clock t.

This invariant implies that $\bar{U}'(k) = \max(\bar{U}(k), (v, t)) = (v, t)$, which proves the desired property. We note that Invariant 4 is maintained because the global clock is incremented when U is updated to U', and, as we describe below, while r is locked.

In the next section, we connect search recency with the hindsight specification from Section 6.3.1. We also discuss the key technical issues that arise when formalizing the above proof in a separation logic like Iris.

7.6 Verifying the Template

We begin by providing a high-level overview on how multicopy structures satsify hindsight specification. The proof crucially uses the fact that searches in multicopy structures satisfy search recency. We then discuss the technical details on formalizing the proof of search recency as well as hindsight specification in Iris.

7.6.1 Proof overview

Recall the hindsight specification (HindSpec) from Section 6.3.1 shown below:

$$\begin{array}{l}\forall\ tid\ t_0\ pvs.\ \boxed{\mathsf{Inv}(r)} \mathrel{-\!\!*} \mathsf{Thread}(tid, t_0) \mathrel{-\!\!*} \\ \left\{\mathsf{Proph}(p, pvs)\ *\ (\mathsf{Upd}(pvs) \mathrel{-\!\!*} \mathsf{AU}_{\mathsf{op}}(\Phi))\right\}\ \mathsf{op}\ r\ k \\ \left\{\begin{array}{r} res.\ \exists pvs'.\ \mathsf{Proph}(p, pvs')\ *\ pvs = (\_\ @\ pvs') \\ *\ (\mathsf{Upd}(pvs) \mathrel{-\!\!*} \Phi(res)) \\ *\ (\neg\mathsf{Upd}(pvs) \mathrel{-\!\!*} \mathsf{PastLin}(\mathsf{op}, res, t_0)) \end{array}\right\}\end{array} \qquad \text{(HindSpec)}$$

In the context of multicopy structures, $\mathsf{Upd}(pvs)$ holds when $\mathtt{op} = \mathtt{upsert}(k)$ as it modifies the abstract state of the structure. To satisfy the (HindSpec) for `upsert`, it must be linearized at the point when the abstract state is modified. This point is when `addContents` succeeds (Line 17).

Contrary to `upsert`, $\mathsf{Upd}(pvs)$ does not hold when $\mathtt{op} = \mathtt{search}$. In this case, (HindSpec) requires establishing the predicate $\mathsf{PastLin}(\mathtt{search}(k), res, t_0)$. That is, for a thread executing $\mathtt{search}(k)$ that begins at time t_0 and returns res, we must establish a time point after t_0

at which the sequential specification of $\mathsf{search}(k)$ is true. To be precise, the predicate $\mathsf{PastLin}(\mathsf{search}(k), res, t_0)$ translates to the following:

$$\mathsf{PastLin}(\mathsf{search}(k), res, t_0) \coloneqq \exists\, t,\ t_0 \leq t\ \wedge\ M_t(k) = res$$

Here, M_t refers to the logical contents of the multicopy structure at time t. Let us next show how search recency helps us establish the above predicate.

Search recency can be stated in precise terms as follows: let (v_s, t_s) be the logical copy of k when thread T executing $\mathsf{search}(k)$ begins at time t_0. Then, T will find a copy (v, t) of k such that $t_s \leq t$. Additionally, both copies of k, namely (v_s, t_s) and (v, t) are part of the upsert history.

To show $\mathsf{PastLin}(\mathsf{search}(k), res, t_0)$, let us do a case-analysis based on whether $t_s = t$ or not. First, let us say $t_s = t$. By $\mathsf{HUnique}$, we can establish that $v_s = v$ in this case. Note that (v_s, t_s) was the logical copy of k at time t_0. Hence, $M_{t_0}(k) = v_s$. Thus, we can establish the predicate $\mathsf{PastLin}(\mathsf{search}(k), res, t_0)$ with t_0 as the witness.

Now let us consider the case $t_s < t$. Note that (v, t) is part of the upsert history. We claim that we can establish $\mathsf{PastLin}(\mathsf{search}(k), res, t_0)$ with t as the witness. To see this, first consider the following observation regarding the fidelity of the timestamps:

> **Invariant 5** For any $(k, (v, t))$ that is part of the upsert history, the copy (v, t) must be contained in the root node at time t.

As a consequence of Invariant 5 and the fact that $(k, (v, t))$ is part of the upsert history, we can establish that $C_{ir}(r)(k) = v$ at time t as well as $M_t(k) = v$. We need to additionally establish that $t_0 \leq t$. For the sake of contradiction, assume that $t < t_0$. Consider the following sequence of logical steps to derive a contradiction:

(1) At time t, $C_{ir}(r)(k) = (v, t)$ by Invariant 5.

(2) At time t_0, $C_{ir}(r)(k) = (v_s, t_s)$ by definition of search recency.

(3) By Invariant 2 and $t < t_0$, we know $\mathsf{ts}(C_{ir}(r)(k))$ at time t is less than or equal to that at time t_0. Hence, by (1) and (2), we have $t \leq t_s$. A contradiction!

This completes the proof connecting the hindsight specification with search recency. We next discuss how to encode the above argument in Iris.

7.6.2 Iris Invariant

The Iris proof must capture the key invariants identified in the proof outline given above in terms of appropriate ghost state constructions. We start by addressing the key technical issue that arises when formalizing the above proof in a separation logic like Iris: contents-in-reach is a recursive function defined over an arbitrary DAG of unbounded size. This makes it difficult to obtain a simple local proof that involves reasoning only about the bounded number of modified nodes in the graph. The recursive and global nature of contents-in-reach

mean that modifying even a single edge in the graph can potentially change the contents-in-reach of an unbounded number of nodes (for example, deleting an edge (n_1, n_2) can change $C_{ir}(n)$ for all n that can reach n_1). A straightforward attempt to prove that a template algorithm preserves Invariant 2 would thus need to reason about the entire graph after every modification (for example, by performing an explicit induction over the full graph). We solve this challenge using the flow framework [100].

Encoding Contents-in-Reach using Flows. Equation (7.2) defines contents-in-reach in a bottom-up fashion, starting from the leaves of the multicopy structure graph. That is, the computation proceeds *backwards* with respect to the direction of the graph's edges. This makes a direct encoding of contents-in-reach in terms of a flow difficult because the flow equation (FlowEqn) describes computations that proceed in the forward direction.

We side-step this problem by tracking auxiliary ghost information in the data structure invariant for each node n in the form of a function $Q_n \colon \mathbb{K} \rightharpoonup \mathsf{V} \times \mathbb{N}$. If these ghost values satisfy

$$Q_n = \lambda k. \begin{cases} C_{ir}(n')(k) & \text{if } \exists n'.\ k \in \mathsf{es}(n, n') \\ \bot & \text{otherwise} \end{cases} \tag{7.3}$$

and we additionally define

$$B_n \coloneqq \lambda k.\ (k \in \mathrm{dom}(C_n(k))\ ?\ C_n(k) : Q_n(k))$$

then $C_{ir}(n) = B_n$. The idea is that each node stores Q_n so that node-local invariants can use it to talk about $C_{ir}(n)$. We then use a flow to propagate the purported values Q_n forward in the graph to ensure that they indeed satisfy (7.3). Note that while an `upsert` or maintenance operations on n may change B_n, it preserves Q_n. That is, operations do not affect the contents-in-reach of downstream nodes, allowing local reasoning about the modification of the contents of n.

In what follows, let us fix a multicopy structure over nodes N and some valuations of the partial functions Q_n. The flow domain M for our encoding of contents-in-reach consists of multisets of key/value-timestamp pairs $M \coloneqq \mathbb{K} \times (\mathsf{V} \times \mathbb{N}) \to \mathbb{N}$ with multiset union as the monoid operation. The edge function induced by the multicopy structure is defined as follows:

$$e(n, n')(\_) \coloneqq \chi(\{(k, Q_n(k)) \mid k \in \mathsf{es}(n, n') \wedge k \in \mathrm{dom}(Q_n)\}) \tag{7.4}$$

Here, χ takes a set to its corresponding multiset. Additionally, we let the function in map every node to the empty multiset. With the definitions of e and in in place, there exists a unique flow fl that satisfies (FlowEqn). Now, if every node n in the resulting flow graph satisfies the following two predicates

$$\phi_1(n) \coloneqq \forall k.\ Q_n(k) = \bot \vee (\exists n'.\ k \in \mathsf{es}(n, n')) \tag{7.5}$$

$$\phi_2(n) \coloneqq \forall k\, p.\ fl(n)(k, p) > 0 \Rightarrow B_n(k) = p \tag{7.6}$$

then $B_n = C_{ir}(n)$. Note that the predicates ϕ_1 and ϕ_2 depend only on n's own flow and its local ghost state (i.e., Q_n, C_n and the outgoing edgesets $\mathsf{es}(n, \_)$).

$$
\begin{aligned}
\mathsf{Inv}_{tpl}(r, H, T) \coloneqq\ & \mathsf{resources}(r, H(T)) \\
& * (\forall t,\ 0 \leqslant t \leqslant T \Rightarrow \mathsf{per\_snapshot}(H(t))) \\
& * (\forall t,\ 0 \leqslant t < T \Rightarrow \mathsf{transition\_inv}(H(t), H(t+1))) \\
& * (\forall n\ k\ v\ t\ t',\ \mathsf{B}(H(t),\ n)(k) = (v, t') \Rightarrow \mathsf{B}(H(t'), r)(k) = (v, t')) \\
\mathsf{resources}(s) \coloneqq\ & \mathop{\Large *}_{n \in \mathsf{FP}(s)} \exists b,\ \mathsf{L}(b, n, \mathsf{Node}(r, n, \mathsf{es}(s, n), \mathsf{val}(C(s, n)))) \\
\mathsf{per\_snapshot}(s) \coloneqq\ & (\mathsf{B}(s, r) = |s|) \\
& * (\forall n,\ n \in \mathsf{FP}(s) \Rightarrow C(s, n) \subseteq \mathsf{U}(s)) \\
& * \mathsf{Init}(\mathsf{U}(s)) * \mathsf{HUnique}(\mathsf{U}(s)) * \mathsf{MaxTS}(T, \mathsf{U}(s)) \\
& * (\forall n,\ n \in \mathsf{FP}(s) \Rightarrow \phi_1(s, n) \wedge \phi_2(s, n)) \\
\mathsf{transition\_inv}(s, s') \coloneqq\ & (\mathsf{FP}(s) \subseteq \mathsf{FP}(s')) * (\mathsf{U}(s) \subseteq \mathsf{U}(s')) \\
& * (\forall n\ k,\ \mathsf{ts}(\mathsf{B}(s, n)(k)) \leq \mathsf{ts}(\mathsf{B}(s', n)(k)))
\end{aligned}
$$

Figure 7.6: Instantiating Inv_{tpl} with invariants of the LSM-DAG template.

7.6.3 Snapshot and the LSM-DAG Template Invariant

We define the snapshot of the multicopy structure as a tuple containing the following components:

- the set of nodes N comprising the structure (also referred to as the *footprint* below)
- the abstract state of the structure (a map from keys to values : $\mathbb{K} \to \mathsf{V}$)
- the upsert history
- the edgesets (a map from N to $N \to \mathcal{P}(\mathbb{K})$)
- the node contents (a map from N to $\mathbb{K} \to \mathsf{V} \times \mathbb{N}$)
- the (purported) contents-in-reach B_n and Q_n for each node
- the representation of flow values

For the scope of this section, we reparameterize each of the above quantities to take snapshot s as an additional parameter, so as to be able to express the value at s. For instance, $\mathsf{U}(s)$ refers to upsert history at s. We also represent the purported contents-in-reach values as $\mathsf{B}(s, n)$ to mean the contents-in-reach of node n in s.

Figure 7.6 provides the definition of Inv_{tpl} for the multicopy template. It contains four conjuncts, we describe each in detail. The predicate resources guards the access to the

$\langle\, b\, R.\, \mathsf{L}(b, n, R) \,\rangle$ `lockNode` n $\langle\, \mathsf{L}(\mathit{true}, n, R) * R \,\rangle$

$\langle\, R.\, \mathsf{L}(\mathit{true}, n, R) * R \,\rangle$ `unlockNode` n $\langle\, \mathsf{L}(\mathit{false}, n, R) \,\rangle$

$\{\mathsf{Node}(r, n, es, V_n)\}$ `inContents` n k $\{x.\, \mathsf{Node}(r, n, es, V_n) * x = (k \in \mathrm{dom}(V_n)\ ?\ \mathsf{Some}(V_n(k)) : \mathsf{None})\}$

$\{\mathsf{Node}(r, n, es, V_n)\}$ `findNext` n k $\{x.\, \mathsf{Node}(r, n, es, V_n) * x = (\exists n'.\ k \in es(n')\ ?\ \mathsf{Some}(n') : \mathsf{None})\}$

$\{\mathsf{Node}(r, r, es, V_r)\}$ `addContents` r k v $\{b.\, \mathsf{Node}(r, r, es, V_r') * V_r' = (b\ ?\ V_r[k \rightarrowtail v] : V_r)\}$

Figure 7.7: Specifications of helper functions used by `search` and `upsert`.

node-level resources Node via predicate $\mathsf{L}(b, n, R_n)$ introduced in Chapter 5. The predicate $\mathsf{L}(b, n, R_n)$ captures the abstract state of n's lock and is used to specify the protocol providing exclusive access to the resource R_n protected by the lock via the helper functions `lockNode` and `unlockNode`. The Boolean b indicates whether the lock is (un)locked. We discuss the predicate Node later.

Predicate per_snapshot captures Invariant 1 and 3. It contains properties that hold for the upsert history such as HUnique. The conjunct $\mathsf{MaxTS}(T, \mathsf{U}(s))$ implies Invariant 4. Finally, the invariants to express contents-in-reach via flow interfaces are also part contained here.

Predicate transition_inv captures Invariant 2 as well as monotone properties such as the upsert history never decreases. The final conjunct in Inv_{tpl} captures the fidelity of the time stamps, i.e., Invariant 5.

The $\mathsf{Node}(r, n, \mathsf{es}(n, \cdot), \mathsf{val}(C_n))$ predicate encapsulates all resources specific to the implementation of the node-specific data structure abstracted by node n. In particular, this predicate owns the resources associated with the physical representation of the data structure and ties them to the abstract ghost state representing the high-level multicopy structure: the node's physical contents $\mathsf{val}(C_n)$ (i.e., C_n without timestamps) and the edgesets of its outgoing edges $\mathsf{es}(n, \cdot)$. Our template proof is parametric in the definition of Node and depends only on the following two assumptions that each implementation used to instantiate the template must satisfy. First, we require that Node is not duplicable:

$$\mathsf{Node}(r, n, es, V_n) * \mathsf{Node}(r', n, es', V_n') \vdash \mathsf{False}$$

Moreover, Node must guarantee disjoint edgesets:

$$\mathsf{Node}(r, n, es, V_n) \vdash \forall n_1\, n_2.\, n_1 = n_2 \vee es(n_1) \cap es(n_2) = \emptyset$$

The specifications of the helper functions used by `search` and `upsert`, given in terms of the predicates $\mathsf{L}(b, n, R_n)$ and $\mathsf{Node}(r, n, es, V_n)$ are shown in Figure 7.7.

With above definition of Inv_{tpl} and the helper function specifications, formalizing the intuitive proof is straightforward. Hence, we turn to the maintenance operations for the multicopy structures.

7.7 Multicopy Maintenance Operations

We next show that we can extend our multicopy structure template in §7.3.1 with a generic maintenance operation without substantially increasing the proof complexity. The basic idea

of our proofs here is that for every timestamped copy of key k, denoted as the pair $(k, (v, t))$, every maintenance operation either does not change the distance of $(k, (v, t))$ to the root or increases it while preserving an edgeset-guided path to $(k, (v, t))$. Using these two facts, we can prove that all the structure invariants are also preserved.

7.7.1 Maintenance template

For the maintenance template, we consider a generalization of the compaction operation found in LSM tree implementations such as LevelDB [58] and Apache Cassandra [4, 79]. While those implementations work on lists for the high-level multicopy structure, our maintenance template supports arbitrary tree-like multicopy structures. The code is shown in Figure 7.8. The template uses the helper function `atCapacity` $r\, n$ to test whether the size of n (i.e., the number of non-$\perp$ entries in n's contents) exceeds an implementation-specific threshold. If not, then the operation simply terminates. In case n is at capacity, the function `chooseNext` is used to determine the node to which the contents of n can be merged. If the contents of n can be merged to successor m of n, then `chooseNext` returns `Some` m. In case no such successor exists, then it returns `None`. If `chooseNext` returns `Some` m, then the contents of n are merged to m. By merge, we mean that some copies of keys are transferred from n to m, possibly replacing older copies in m. The merge is performed by the helper function `mergeContents`. It must ensure that all keys k merged from C_n to C_m satisfy $k \in \mathsf{es}(n, m)$.

On the other hand, if `chooseNext` returns `None`, then a new node is allocated using the function `allocNode`. The new node is then added to the data structure using the helper function `insertNode`. Here, the new edgeset $\mathsf{es}(n, m)$ must be disjoint from all edgesets for the other successors m' of n. Afterwards, the contents of n are merged to m as before. Note that the maintenance template never removes nodes from the structure. In practice, the depth of the structure is bounded by letting the capacity of nodes grow exponentially with the depth. The right hand side of Figure 7.8 shows the intermediate states of a potential execution of the `compact` operation.

7.7.2 High-level proof of `compact`

The verification framework presented in Chapter 6 can be easily extended to accommodate any maintenance operation as it does not change the data structure's abstract state. Essentially, we need to prove that `compact` satisfies the following atomic triple:

$$\langle\, M.\ \mathsf{MCS}(r, M) \,\rangle\ \texttt{compact}\ r\ \langle\, \mathsf{MCS}(r, M) \,\rangle$$

This specification says that `compact` logically takes effect in a single atomic step, and at this step the abstract state of the data structure does not change. We prove that `compact` satisfies this specification relative to the specifications of the implementation-specific helper functions shown in Figure 7.9. The postcondition of `mergeContents` is given with respect to an (existentially quantified) set of keys K that are merged from V_n to V_m, resulting in new content sets V_n' and V_m'. The new contents are determined by the functions *mergeLeft* and

```
1  let rec compact r n =
2    lockNode n;
3    if atCapacity r n then begin
4      match chooseNext r n with
5      | Some m ->
6        lockNode m;
7        mergeContents r n m;
8        unlockNode n;
9        unlockNode m;
10       compact r m
11     | None ->
12       let m = allocNode () in
13       insertNode r n m;
14       mergeContents r n m;
15       unlockNode n;
16       unlockNode m;
17       compact r m
18   end
19   else
20     unlock n
```

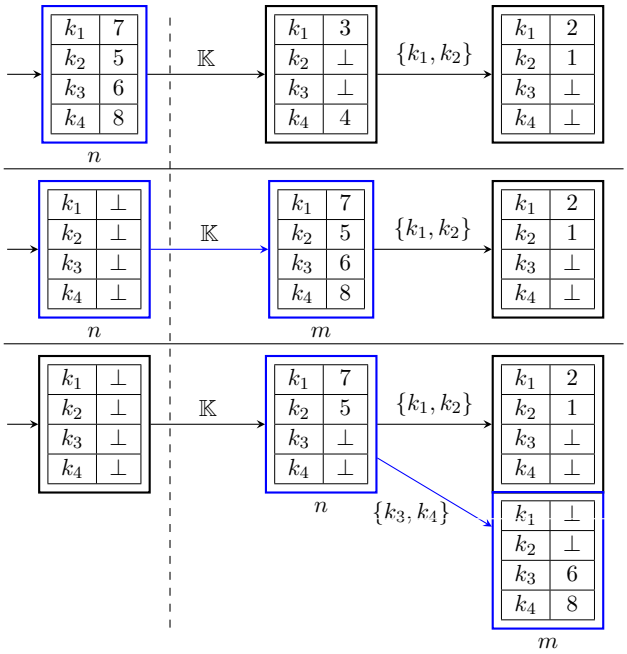


Figure 7.8: Maintenance template for tree-like multicopy structures. The template can be instantiated by providing implementations of helper functions `atCapacity`, `chooseNext`, `mergeContents`, `allocNode`, and `insertNode`. `atCapacity` $r\,n$ returns a Boolean value indicating whether node n has reached its capacity. The helper function `chooseNext` $r\,n$ returns Some m if there exists a successor m of n in the data structure into which n should be compacted, and None in case n cannot be compacted into any of its successors. `mergeContents` $r\,n\,m$ (partially) merges the contents of n into m. Finally, `allocNode` is used to allocate a new node and `insertNode` $r\,n\,m$ inserts node m into the data structure as a successor of n. The right hand side shows a possible execution of `compact`. Edges are labeled with their edgesets. The nodes n and m in each iteration are marked in blue. For simplicity, we here assume that the values are identical to their associated timestamps and only show the timestamps.

mergeRight which are defined as follows:

$$\mathit{mergeLeft}(K, V_n, Es, V_m) \coloneqq \lambda k.\ (k \in K \cap \mathrm{dom}(V_n) \cap Es\ ?\ \bot : V_n(k))$$
$$\mathit{mergeRight}(K, V_n, Es, V_m) \coloneqq \lambda k.\ (k \in K \cap \mathrm{dom}(V_n) \cap Es\ ?\ V_n(k) : V_m(k))$$

Technically, the linearization point of the operation occurs when all locks are released, just before the function terminates. However, the interesting part of the proof is to show that the changes to the physical contents of nodes n and m performed by each call to `mergeContents` at line 7 preserve the abstract state of the structure as well as the invariants. In particular, the changes to C_n and C_m also affect the contents-in-reach of m. We need to argue that this is a local effect that does not propagate further in the data structure, as we did in our proof of `upsert`.

Auxiliary invariants. When proving the correctness of `compact`, we face two technical challenges. The first challenge arises when establishing that `compact` changes the contents of

$\{\mathsf{Node}(r, n, es_n, V_n)\}$ `atCapacity` $r\ n$ $\{b.\, \mathsf{Node}(r, n, es_n, V_n)\}$

$\{\mathsf{Node}(r, n, es_n, V_n)\}$
`chooseNext` $r\ n$
$\{v.\, \mathsf{Node}(r, n, es_n, V_n) * (v = \mathsf{Some}(m) * es_n(m) \neq \emptyset \vee v = \mathsf{None} * \mathsf{needsNewNode}(r, n, es_n, V_n))\}$

$\{\mathsf{True}\}$ `allocNode` r $\{m.\, \mathsf{Node}(r, m, (\lambda n'.\, \emptyset), \emptyset)\}$

$\{\mathsf{Node}(r, n, es_n, V_n) * \mathsf{needsNewNode}(r, n, es_n, V_n) * \mathsf{Node}(r, m, (\lambda n'.\, \emptyset), \emptyset)\}$
`insertNode` $r\ n\ m$
$\{\mathsf{Node}(r, n, es'_n, V_n) * \mathsf{Node}(r, m, (\lambda n'.\, \emptyset), \emptyset) * es'_n = es_n[m \rightarrowtail es'_n(m)] * es'_n(m) \neq \emptyset\}$

$\{\mathsf{Node}(r, n, es_n, V_n) * \mathsf{Node}(r, m, es_m, V_m) * es_n(m) \neq \emptyset\}$
`mergeContents` $r\ n\ m$
$$\left\{\begin{array}{r} \mathsf{Node}(r, n, es_n, V'_n) * \mathsf{Node}(r, m, es_m, V'_m) \\ * V'_n = mergeLeft(K, V_n, Es, V_m) * V'_m = mergeRight(K, V_n, Es, V_m) \end{array}\right\}$$

Figure 7.9: Specifications of helper functions used by `compact`.

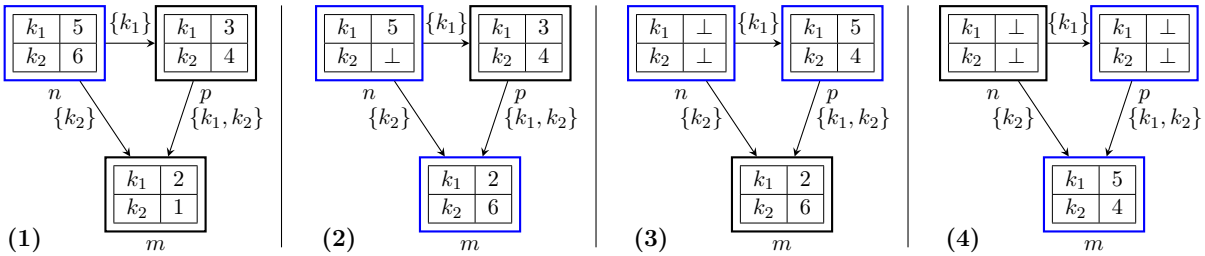


Figure 7.10: Possible execution of the `compact` operation on a DAG. Edges are labeled with their edgesets. The nodes undergoing compaction in each iteration are marked in blue.

the nodes involved in such a way that the high-level invariants are maintained. In particular, we must reestablish Invariant 2, which states that the contents-in-reach of each node can only increase over time. Compaction replaces downstream copies of keys with upstream copies. Thus, in order to maintain Invariant 2, we need the additional auxiliary invariant that the timestamps of keys in the contents of nodes can only decrease as we move away from the root:

Invariant 6 The (timestamp) contents of a node is not smaller than the contents-in-reach of its successor. That is, for all keys k and nodes n and m, if $k \in \mathsf{es}(n, m)$ and $C_n(k) \neq \bot$ then $\mathsf{ts}(C_{ir}(m)(k)) \leqslant \mathsf{ts}(C_n(k))$.

We can capture Invariant 6 in our template invariant $\mathsf{Inv}_{tpl}(r, H, T)$ by adding the following predicate as an additional conjunct to the predicate per_snapshot:

$$\phi_3(n) \coloneqq \forall k.\, \mathsf{ts}(Q_n(k)) \leqslant \mathsf{ts}(B_n(k)) \tag{7.7}$$

The second challenge is that the maintenance template generates only tree-like structures. This implies that at any time there is at most one path from the root to each node in the

structure. We will see that this invariant is critical for maintaining Invariant 6. However, the data structure invariant presented thus far allows for arbitrary DAGs.

To motivate this issue further, consider the multicopy structure in step **(1)** of Figure 7.10. The logical contents of this structure (i.e. the contents-in-reach of n) is $\{k_1 \rightarrowtail (5,5), k_2 \rightarrowtail (6,6)\}$.

The structure in step **(2)** shows the result obtained after executing `compact` $r\, n$ to completion where n has been considered to be at capacity and the successor m has been chosen for the merge, resulting in $(k_2, (6,6))$ being moved from n to m. Note that at this point the logical contents of the data structure is still $\{k_1 \rightarrowtail (5,5), k_2 \rightarrowtail (6,6)\}$ as in the original structure. However, the structure now violates Invariant 6 for nodes p and m since $\mathsf{ts}(B_m(k_2)) > \mathsf{ts}(C_p(k_2))$.

Suppose that now a new compaction starts at n that still considers n at capacity and chooses p for the merge. The merge then moves the copy $(k_1, (5,5))$ from n to p. The graph in step **(3)** depicts the resulting structure. The compaction then continues with p, which is also determined to be at capacity. Node m is chosen for the merge, resulting in $(k_1, (5,5))$ and $(k_2, (4,4))$ being moved from p to m. At this point, the second compaction terminates. The final graph in step **(4)** shows the structure obtained at this point. Observe that the logical contents is now $\{k_1 \rightarrowtail (5,5), k_2 \rightarrowtail (4,4)\}$. Thus, this execution violates the specification of `compact`, which states that the logical contents must be preserved. In fact, a timestamp in the contents-in-reach of n has decreased, which violates Invariant 2.

We observe that although `compact` will create only tree-like structures, we can prove its correctness using a weaker invariant that does not rule out non-tree DAGs, but instead focuses on how `compact` interferes with concurrent `search` operations. This weaker invariant relies on the fact that for every key k in the contents of a node n, there exists a unique search path from the root r to n for k. That is, if we project the graph to only those nodes reachable from the root via edges (n, m) that satisfy $k \in \mathsf{es}(n, m)$, then this projected graph is a list. Using this weaker invariant we can capture implementations based on B-link trees or skip lists which are DAGs but have unique search paths.

To this end, we recall from [153] the notion of the *inset* of a node n, $\mathsf{inset}(n)$, which is the set of keys k such that there exists a (possibly empty) path from the root r to n, and k is in the edgeset of all edges along that path. That is, since a `search` for a key k traverses only those edges (n, m) in the graph that have k in their edgeset, the `search` traverses (and accesses the contents of) only those nodes n such that $k \in \mathsf{inset}(n)$. Now observe that `compact`, in turn, moves new copies of a key k downward in the graph only along edges that have k in their edgeset. The following invariant is a consequence of these observations and the definition of contents-in-reach:

Invariant 7 A key is in the contents-in-reach of a node only if it is also in the node's inset. That is, $\mathrm{dom}(C_{ir}(n)) \subseteq \mathsf{inset}(n)$.

This invariant rules out the problematic structure in step **(1)** of Figure 7.10 because we have $k_2 \in \mathrm{dom}(C_{ir}(p))$ but $k_2 \notin \mathsf{inset}(p) = \{k_1\}$.

Invariant 7 alone is not enough to ensure that Invariant 6 is preserved. For example, consider the structure obtained from **(1)** of Figure 7.10 by changing the edgeset of the edge

(n, p) to $\{k_1, k_2\}$. This modified structure satisfies Invariant 7 but allows the same problematic execution ending in the violation of Invariant 6 that we outlined earlier. However, observe that in the modified structure $k_2 \in \mathsf{es}(n, p) \cap \mathsf{es}(n, m)$, which violates the property that all edgesets leaving a node are disjoint. We have already captured this property in our data structure invariant (as an assumption on the implementation-specific predicate $\mathsf{Node}(r, n, es, C_n)$). However, in our formal proof we need to rule out the possibility that a search for k can reach a node m via two *incoming* edgesets $\mathsf{es}(n, m)$ and $\mathsf{es}(p, m)$. Proving that disjoint *outgoing* edgesets imply unique search paths involves global inductive reasoning about the paths in the multicopy structure. To do this using only local reasoning, we will instead rely on an inductive consequence of locally disjoint outgoing edgesets, which we capture explicitly as an additional auxiliary invariant (and which we will enforce using flows):

Invariant 8 The distinct immediate predecessors of any node n have disjoint insets. More precisely, for all distinct nodes n, p, m, and keys k, if $k \in \mathsf{es}(n, m) \cap \mathsf{es}(p, m)$ then $k \notin \mathsf{inset}(n) \cap \mathsf{inset}(p)$.

Note that changing the edgeset of (n, p) in Figure 7.10 to $\{k_1, k_2\}$ would violate Invariant 8 because the resulting structure would satisfy $k_2 \in \mathsf{es}(n, m) \cap \mathsf{es}(p, m)$ and $k_2 \in \mathsf{inset}(n) \cap \mathsf{inset}(p)$.

In order to capture invariants 7 and 8 in $\mathsf{Inv}_{tpl}(r, T, H)$, we introduce an additional flow that we use to encode the inset of each node. The encoding of insets in terms of a flow follows [97]. That is, the underlying flow domain is multisets of keys $M = \mathbb{K} \to \mathbb{N}$ and the actual calculation of the insets is captured by (FlowEqn) if we define:

$$e(n, n') \coloneqq \lambda m.\, m \cap \mathsf{es}(n, n') \qquad\qquad in(n) \coloneqq \chi\,(n = r\ ?\ \mathbb{K} : \emptyset)$$

If $\mathit{fl}_{\mathsf{inset}}$ is a flow that satisfies (FlowEqn) for these definitions of e and in, then for any node n that is reachable from r, $\mathit{fl}_{\mathsf{inset}}(n)(k) > 0$ iff $k \in \mathsf{inset}(n)$. Invariants 7 and 8 are then captured by the following two predicates, which we add to $\mathsf{N_S}$:

$$\phi_4(n) \coloneqq \forall k.\, k \in \mathrm{dom}(B_n) \Rightarrow \mathit{fl}_{\mathsf{inset}}(n)(k) > 0 \qquad \phi_5(n) \coloneqq \forall k.\, \mathit{fl}_{\mathsf{inset}}(n)(k) \leqslant 1$$

Note that ϕ_5 captures Invariant 8 as a property of each individual node n by taking advantage of the fact that the multiset $\mathit{fl}_{\mathsf{inset}}(n)$ explicitly represents all of the contributions made to the inset of n by n's predecessor nodes.

We briefly explain why we can still prove the correctness of `search` and `upsert` with the updated data structure invariant. First note that `search` does not modify the contents, edgesets, or any other ghost resources of any node. So the additional conjuncts in the invariant are trivially maintained.

Now let us consider the operation `upsert` $r\, k\, v$. Since `upsert` does not change the edgesets of any nodes, the resources and constraints related to the inset flow are trivially maintained, with the exception of $\phi_4(r)$: after the `upsert` we have $k \in \mathrm{dom}(B_r)$ which may not have been true before. However, from $in(r)(k) = 1$, the flow equation, and the fact that the flow domain is positive, it follows that we must have $\mathit{fl}_{\mathsf{inset}}(r)(k) > 0$ (i.e., $k \in \mathsf{inset}(r) = \mathbb{K}$). Hence, $\phi_4(r)$ is preserved as well.

8 Lockfree Templates

This chapter introduces the Skiplist template that covers lock-free linked lists and skiplists. Following the same outline as Chapter 7, we first provide an intuitive proof of correctness of the Skiplist template, followed by details on how to establish the hindsight specification for the Skiplist template.

8.1 Introduction

Earlier works [97, 98, 138] developed a framework to verify a wide range of lock-based implementations of concurrent search structures. Specifically, they proved that these implementations are linearizable [70].

A core ingredient of the framework is the idea of template algorithms [153]. A template algorithm dictates how threads interact but abstracts away from the concrete layout of nodes in memory. Once the template algorithm is verified, its proof can be instantiated on a variety of search structures.

The template algorithms of [97, 98, 138] use locks as a synchronization technique. Locks ensure non-interference on portions of memory to guarantee that certain needed constraints hold in spite of concurrency.

The disadvantage of locks is that if a thread holding a lock on some portion of memory p stops, then no other thread can get a conflicting lock on p. For that reason, some practical implementations such as Java's `ConcurrentSkipListMap` [135] use lock-free algorithms.

This chapter shows how to capture multiple variants of concurrent lock-free skiplists and linked lists in the form of template algorithms. Thus, proving the correctness of such a template algorithm results in a proof that is applicable to many variants at once. Our template algorithms are parametric in the skiplist height and allow variations along the following three dimensions: (i) maintenance style (eager vs lazy) (ii) node implementations and (iii) the order of maintenance operations on the higher levels of the skiplists.

By instantiating our template algorithm with appropriate maintenance operations and node implementations, we obtain verified versions of existing (skip)list algorithms from the literature such as the Herlihy-Shavit skiplist algorithm [68, § 14], the Michael set [122], and the Harris list algorithm [62]. We also obtain new concurrent skiplist algorithms that have not been considered before. These new algorithms are correct by construction thanks to our modular verification framework.

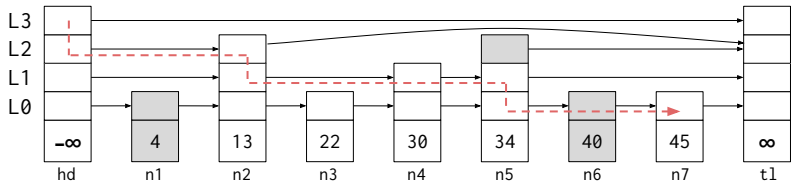


Figure 8.1: Skiplist with four levels. A node that is marked (logically deleted) at a level is shaded gray at that level. The red line indicates the path taken by a traversal searching for key 42.

We mechanize our development in the concurrent separation logic Iris [84, 86]. The mechanization heavily relies on the hindsight reasoning in Iris developed in Chapter 6.

The roadmap for this chapter is as follows: we begin by describing the Skiplist template algorithms, followed by the intuitive proof argument for the linearizability of the Skiplist templates using hindsight reasoning. Finally, we obtain the formal proof of linearizability by connecting the intuitive proof to the developement on hindsight reasoning from Chapter 6.

8.2 The Skiplist Template Algorithm

A *skiplist* is a search structure over a totally ordered set of keys $\mathbb{K}$. We focus our discussion on skiplists that implement mutable sets rather than maps. The extension of the presented algorithms to mutable maps is straightforward. The data structure is composed of sorted lists at multiple levels, with the base list determining the actual contents of the structure, while higher level lists are used to speed up the search. An example is shown in Figure 8.1. A skiplist node contains a key and has a height, determining how many higher level lists this node is a part of. Each node has a next pointer for each of its levels. Two sentinel nodes signify the head (*hd* with key $-\infty$) and the tail (*tl* with key ∞) of the skiplist. Lock-free linked lists often use the technique of logical deletion by *marking* a node before it is physically unlinked from the list. This involves storing a mark bit together with the next pointer, so as to allow reading and updating them together in a single (logically) atomic step. Lock-free skiplist implementations also use this technique. Since a skiplist node can be part of multiple lists, it has one mark bit per level.

The traversal for a key not only goes left to right as usual, but also top to bottom. The red line in Figure 8.1 depicts a traversal searching for key 42. The traversal begins at the highest level of the head node. At each non-base level, the traversal continues till it reaches a node with a key greater than or equal to the search key. Thereafter, the traversal drops down a level, and continues at the lower levels until it terminates on the bottom level at the first node whose key is greater than or equal to the search key.

The traversals in a concurrent skiplist perform *maintenance* in the form of physically unlinking encountered marked nodes. In Figure 8.1, node n_5 has been unlinked at level 2, thus the traversal does not visit it at that level. Operations that mark and change the next pointers at the higher levels do not affect the actual contents of the structure. We therefore consider them to be part of the maintenance.

Many variants of lock-free skiplist algorithms have been proposed in the literature and implemented in practice. These variants differ in (i) their node implementations, (ii) the styles of maintenance operations and/or (iii) the orders in which they perform maintenance operations with regard to other operations.

For example, node implementations in low-level languages often use bit-stealing [68] (or an equivalent of Java's `AtomicMarkableReference`) so that both the next pointer and mark bit can be atomically read or updated. Other implementations use more complex solutions. For instance, the skiplists in [51] use nodes with back links to reduce traversal restarts due to marked nodes. Java's `ConcurrentSkipListMap` [135] implements each node as a list of simpler nodes, one per level. The higher level nodes have both right pointers and down pointers, while the base nodes only have right pointers. Java's implementation also uses *marker nodes* for marking, instead of bit-stealing.

In terms of style of maintenance, the traversal in the Michael Set [122] and Herlihy-Shavit lock-free skiplist [68, § 14] unlinks one marked node at a time. By contrast, the traversal in the Harris List [62] unlinks the entire sequence of marked nodes in one shot with a single CAS operation. The variants also differ in the order of marking of a node at higher levels. In the Herlihy-Shavit skiplist, the marking of a node goes from top level to the bottom level. This differs from skiplists in [135] and [51], whose marking goes from bottom to top.

Despite the differences in the skiplist algorithms described above (and others to be invented in the future), the bulk of their correctness reasoning remains the same. A goal of this dissertation is to show how to exploit that fact.

Template algorithm. Our template algorithm for skiplists abstracts away from node-level implementation details and the way in which traversals perform maintenance. As we shall see, the particular details regarding how the data is stored internal to the node does not affect the correctness of the core operations - `search`, `insert` and `delete`. Nor is the correctness affected by whether the traversal unlinks one marked node at a time or an entire sequence of marked nodes. We also show that the order in which maintenance operations are performed on the higher levels of the list does not matter for correctness. In summary, the template algorithm we present abstracts from: (i) node-level details; (ii) the style of unlinking marked nodes and (iii) the order of maintenance operations on higher levels.

The template algorithm is assumed to be operating on a set of nodes N that contains the two sentinel nodes head hd and tail tl. Let the maximum allowed height of a skiplist node be L (> 1). Each node n is associated with (i) its key $\mathsf{key}(n) \in \mathbb{K} = \mathbb{N} \cup \{-\infty, \infty\}$, (ii) its height $\mathsf{height}(n) \in [1, L)$, (iii) the next pointers $\mathsf{next}(n, i) \in N$ for each i from 0 to $\mathsf{height}(n) - 1$, and (iv) its mark bits per level $\mathsf{mark}(n, i) \in \{true, false\}$ for each i from 0 to $\mathsf{height}(n) - 1$. When discussing $\mathsf{next}(n, i)$ or $\mathsf{mark}(n, i)$, we implicitly assume that i lies between 0 and $\mathsf{height}(n) - 1$. We sometimes say a node n is unmarked to mean that it is unmarked at the base level, i.e., $\mathsf{mark}(n, 0) = false$. The structural invariant maintains the following facts: $\mathsf{key}(hd) = -\infty$, $\mathsf{key}(tl) = \infty$, $\mathsf{height}(hd) = \mathsf{height}(tl) = \mathsf{L}$, $\mathsf{next}(tl, i) = tl$ for all i, $\mathsf{next}(hd, \mathsf{L} - 1) = tl$, $\mathsf{mark}(hd, i) = \mathsf{mark}(tl, i) = false$ for all i.

The core operations of the Skiplist template are expressed using *helper functions* such as `findNext` and `markNode` that abstract from the details of the node implementation. We describe

```
let search k =
  let ps = allocArr L hd in
  let cs = allocArr L tl in
  let _, _, res = traverse ps cs k in
  res

let delete k =
  let ps = allocArr L hd in
  let cs = allocArr L tl in
  let p, c, res = traverse ps cs k in
  if not res then
    false
  else
    maintainanceOp_del c;
    match markNode 0 c with
    | Success -> traverse ps cs k; true
    | Failure -> false
```

```
let insert k =
  let ps = allocArr L hd in
  let cs = allocArr L tl in
  let p, c, res = traverse ps cs k in
  if res then
    false
  else
    let h = randomNum L in
    let e = createNode k h cs in
    match changeNext 0 p c e with
    | Success ->
      maintainanceOp_ins k ps cs e; true
    | Failure -> insert k
```

Figure 8.2: The template algorithm for lock-free skiplists. The template can be instantiated by providing implementations of `traverse` and the helper functions `markNode`, `createNode` and `changeNext`. The `markNode` i c attempts to mark node c at level i atomically, and fails if c has been marked already. `createNode` k h cs creates a new node e of height h containing k, and whose next pointers are set to nodes in array cs. Finally, `changeNext` i p c cn is a CAS operation attempting to change the next pointer of p from c to cn. `changeNext` i p c cn succeeds only if $\mathsf{mark}(p, i) = \mathit{false}$ and $\mathsf{next}(p, i) = c$. Other functions used here include `randomNum` to generate a random number and maintenance operations associated with `insert` and `delete`. `maintainanceOp_del` marks node c at the higher levels, while `maintainanceOp_ins` inserts a new node e at the higher levels.

the behavior of these helper functions as and when we encounter them. The template is instantiated by implementing these functions. The helper functions are assumed to be *logically atomic*, i.e., appear to take effect in a single step during its execution.

Figure 8.2 shows the core operations of the Skiplist template algorithm. (We omit the code for the data structure initialization as it is straightforward.) All three operations begin by allocating two arrays ps and cs via `allocArr`, each of size `L` and values initialized to hd and tl respectively. These arrays are then populated by the `traverse` operation as it computes the predecessor-successor pair for operation key k at each level. Intuitively, these pairs indicate where k would be inserted at each level. The template algorithm here abstracts away from the concrete `traverse` implementation. We later consider two implementations of `traverse` that differ in the way that maintenance is performed, as discussed earlier.

As far as the core operations are concerned, they rely on `traverse` to satisfy the following specification. First, it returns a triple (p, c, res) where p and c are nodes and res a Boolean such that $p = ps[0]$, $c = cs[0]$ and res is true iff k is contained in c. Second, the node c must have been unmarked at some point during the traversal; and third, for each $0 \leqslant i < \mathsf{L}$, the traversal observes that $\mathsf{key}(ps[i]) < k \leqslant \mathsf{key}(cs[i])$.

Let us now describe the core operations, starting with the `search` operation. The `search` operation simply invokes the `traverse` function, whose result establishes whether k was in

the structure. The `delete` operation starts similarly by invoking `traverse` and checking if the key is present in the structure. If it is, then `delete` invokes the maintenance operation `maintainanceOp_del`, which attempts to mark c at the higher levels (i.e. all levels except 0). We provide the implementation of `maintainanceOp_del` in a moment. Once `maintainanceOp_del` terminates, `delete` finally attempts to mark c via `markNode` at the base level. If marking succeeds, it terminates by invoking `traverse` (which performs the task of physically unlinking marked nodes at all levels) and returning *true*. Otherwise, a concurrent thread must have already marked c, in which case `delete` returns *false*.

The `insert` operation also begins with `traverse`. If the traversal returns *true*, then the key must already have been present. Hence, `insert` returns *false* in this case. Otherwise, a new node e is created using `createNode`. The node's height is determined randomly using `randomNum`, which generates a random number h such that $0 < h < L$. After creating a new node, the algorithm attempts to insert it into the list by calling `changeNext` at the base level (line 27). If the attempt succeeds, `insert` proceeds by invoking the maintenance operation `maintainanceOp_ins`, which also inserts the new node into the list at all higher levels. The `insert` then returns with *true*. If the `changeNext` operation fails, then the entire operation is restarted.

We now describe the maintenance operations for `insert` and `delete`, shown in Figure 8.3. The maintenance operations here differ from those in traditional skiplist implementations in regards to the order in which maintenance is performed at higher levels. In traditional implementations, the marking of a node goes from top to bottom, while insertion of a new node goes from bottom to top. The Skiplist template presented here makes sure that the base level gets marked at the end and the insertion first happens at the base level, but it imposes no order on how it proceeds at higher levels. That is, when marking a node, a `delete` thread could for instance first mark odd levels, then even levels and finally the base level 0. The maintenance operations in the Skiplist template captures all such permutations. As our proof shows later, the order of maintenance at higher levels has no bearing on the correctness of the algorithm.

The `maintainanceOp_del` marks node c from levels 1 to `height`(c). It begins by reading the height of c as h, and generating a permutation of $[1 \ldots (h-1)]$ stored in array pm via the `permute` function. The `maintainanceOp_del_rec` then recursively marks c in the order prescribed by pm. Note that the maintenance continues regardless of whether `markNode` succeeds or fails, because c will be marked at the end regardless.

The `maintainanceOp_ins` begins in the same way by reading the height, generating the permutation and invoking `maintainanceOp_ins_rec`. The `maintainanceOp_ins_rec` first collects the predecessor-successor pair at the current level from arrays ps and cs, respectively. Then it tries to insert the new node e using `changeNext` on predecessor node p. If `changeNext` succeeds, then the recursive operation continues. Otherwise, it recomputes the predecessor-successor pairs using `traverse`. After the recomputation, the insertion is retried at the same level.

We can now finally turn to the implementations of `traverse`. We consider two implementations that differ in their treatment of marked nodes. The *eager* traversal attempts

```
let maintainanceOp_del_rec i h pm c =
  if i < h-1 then
    let idx = pm[i] in
    markNode idx c;
    maintainanceOp_del_rec (i+1) h pm c
  else ()

let maintainanceOp_del c =
  let h = getHeight c in
  let pm = permute h in
  maintainanceOp_del 0 h pm c
```

```
let maintainanceOp_ins_rec i h pm ps cs e =
  if i < h-1 then
    let idx = pm[i] in
    let p = ps[idx] in
    let c = cs[idx] in
    match changeNext idx p c e with
    | Success ->
      maintainanceOp_ins_rec (i+1) h pm ps cs e
    | Failure ->
      traverse ps cs k;
      maintainanceOp_ins_rec i h pm ps cs e
  else ()

let maintainanceOp_ins k ps cs e =
  let h = getHeight e in
  let pm = permute h in
  maintainanceOp_ins 0 h pm ps cs e
```

Figure 8.3: The maintenance operations for the Skiplist template. The `getHeight` c helper function returns height(c). The `permute` function generates a permutation of $[1 \ldots (h-1)]$ as an array.

to unlink every marked node it encounters, while the *lazy* traversal simply walks over the marked nodes till it reaches an unmarked node. The traversal then attempts to unlink the entire marked segment at once. We begin with the eager traversal first.

The eager traversal is shown in Figure 8.4. The `traverse` function is implemented using mutually-recursive functions `eager_rec` and `eager_i`[1]. The function `eager_rec` populates the arrays ps and cs with the predecessor-successor pair at level i computed by `eager_i`. The `eager_i` performs a traversal at level i by first reading the mark bit and next pointer of c using `findNext`. If c is found to be marked, then `eager_i` attempts to physically unlink the node using `changeNext`. In the case that `changeNext` fails (because either p is marked or it does not point to c anymore), `eager_i` simply restarts the `traverse` function. In the case of `Success` of `changeNext`, the traversal continues. If c is unmarked, then `traverse_i` proceeds by comparing k to `key`(c). For `key`(c) $< k$, the traversal continues with c and cn. Otherwise, `eager_i` ends at c, returning $(p, c, true)$ if `key`(c) $= k$ and $(p, c, false)$ otherwise. As mentioned before, `eager_i` attempts to unlink immediately whenever a marked node is encountered.

Finally, we describe the lazy traversal. The code for the lazy traversal is shown in Figure 8.5. While the `lazy_rec` implementation is almost identical to `eager_rec`, the two differ in the per-level traversal in `lazy_i` and `eager_i`. The `lazy_i` function here keeps track of three nodes while traversing: node p is the last unmarked node it witnessed, node pn is the node that p was pointing to when p was traversed, and c is the node that is currently being traversed. The function `lazy_i` begins by reading the next pointer and mark bit of

[1]For ease of exposition, the implementation of the eager traversal shown in Figure 8.4 differs slightly from the version we have verified in Iris. The Iris version uses option return types instead of mutually-recursive functions in order to obtain a more modular proof of the eager traversal. We use the mutually recursive implementation here for clarity of exposition.

```
let eager_i i k p c =
  match findNext i c with
  | cn, true ->
    match changeNext i p c cn with
    | Success -> eager_i i k p cn
    | Failure -> traverse ps cs k
  | cn, false ->
    let kc = getKey c in
    if kc < k then
      eager_i i k c cn
    else
      let res = (kc = k ? true : false) in
      (p, c, res)

let eager_rec i ps cs k =
  let p = ps[i+1] in
  let c, _ = findNext i p in
  let p', c', res = eager_i i k p c in
  ps[i] <- p';
  cs[i] <- c';
  if i = 0 then
    (p', c', res)
  else
  eager_rec (i-1) ps cs k

let traverse ps cs k =
  eager_rec (L - 2) ps cs k
```

Figure 8.4: The eager traversal for the Skiplist template. `findNext` $i\,k\,c$ returns a pair $(\mathsf{next}(c,i), \mathsf{mark}(c,i))$. The `getKey` c helper function returns $\mathsf{key}(c)$.

c. If c is found to be marked, then `lazy_i` simply continues with the successor of c. If c is unmarked, then the operation key k is compared with $\mathsf{key}(c)$. In case $\mathsf{key}(c) < k$, then again `lazy_i` continues with c as the last seen unmarked node. Otherwise, $k \leqslant \mathsf{key}(c)$. In this case, it attempts to unlink the marked segment between nodes p and c. The conditional on Line 10 checks if there is a segment to remove. If that is indeed the case, then `lazy_i` calls `changeNext` to unlink the segment between p and c. If `changeNext` succeeds, then there is a further check to make sure c is still unmarked. If so, then `lazy_i` returns with p and c. In all other cases where `changeNext` fails or c is found to be marked, `lazy_i` restarts the `traverse` function.

```
let lazy_i i k p pn c =
  match findNext i c with
  | cn, true -> lazy_i i k p pn cn
  | cn, false ->
    let kc = getKey c in
    if kc < k then
      lazy_i i k c cn cn
    else
      let res = (kc = k ? true : false) in
      if pn = c then
        (p, c, res)
      else
        match changeNext i p pn c with
        | Success ->
          let _, b = findNext i c in
          if b then traverse ps cs k
          else (p, c, res)
        | Failure -> traverse ps cs k

let lazy_rec i ps cs k =
  let p = ps[i+1] in
  let c, _ = findNext i p in
  let p', c', res = lazy_i i k p c c in
  ps[i] <- p';
  cs[i] <- c';
  if i = 0 then
    (p', c', res)
  else
    lazy_rec (i-1) ps cs k

let traverse ps cs k =
  lazy_rec (L - 2) ps cs k
```

Figure 8.5: The lazy traversal for the Skiplist template corresponding to the Harris List style traversal.

8.3 Proof Intuition

Our goal is to show that the Skiplist template is linearizable. That is, we must prove that each of the core operations takes effect in a single atomic step during its execution, the *linearization point*, and satisfies the sequential specification shown in Figure 8.6. For the Skiplist template, we define the abstract state $C(N)$ to be the union of the *logical contents* $C(n)$ of all nodes in N, where $C(n) \coloneqq (\mathsf{mark}(n, 0)\ ?\ \emptyset : \{\mathsf{key}(n)\})$. In other words, the abstract state of the structure is a collection of keys contained in unmarked nodes at the base level. We rely on techniques developed in the earlier chapters to analyse the Skiplist template.

$$\Psi_{\mathtt{op}}(k, C, C', res) \coloneqq \begin{cases} C' = C \wedge (res \iff k \in C) & \mathtt{op} = \mathtt{search} \\ C' = C \cup \{k\} \wedge (res \iff k \notin C) & \mathtt{op} = \mathtt{insert} \\ C' = C \setminus \{k\} \wedge (res \iff k \in C) & \mathtt{op} = \mathtt{delete} \end{cases}$$

Figure 8.6: Sequential specification of a search structure as a Set ADT (repeated). k refers to the operation key, C and C' to the abstract state before and after operation op, respectively, and res is the return value of op.

The two main techniques that we rely on are the *Edgeset Framework* from Chapter 3 and *Hindsight Reasoning* from Chapter 6. We begin by giving a brief overview of the hindsight reasoning in the context of the Skiplist template.

8.3.1 Hindsight Reasoning

To show the applicability of the hindsight framework to the Skiplist template, we elaborate on how the Skiplist template operations exhibit future-dependent linearization points. That is, the linearization point of an operation cannot be determined at any fixed moment, but only at the end of the execution, once any interference of other concurrent operations has been accounted for. To understand the interference issue, consider the `search` operation. Since, `search` returns the result of `traverse`, let us look at the eager traversal implementation. To simplify the explanation further, let us assume that the maximum height allowed for every non-sentinel node is one. Then, we can ignore the `eager_rec` function and focus on `eager_i` called at the base level.

Let there be a thread T executing `search(7)`. Concurrently, there is a thread T_d executing `delete(7)` and a thread T_i executing `insert(7)`. Figure 8.7 shows interesting scenarios that thread T might potentially observe. Box (a) captures the state of the structure at the beginning of the `eager_i` call processing n_2. Let **Scenario 1** be the situation when thread T faces no interference from T_d and T_i. Here, thread T finds the key 7 in n_2 and `eager_i` returns *true*. The point when `eager_i` finds n_2 to be unmarked becomes the linearization point for this scenario.

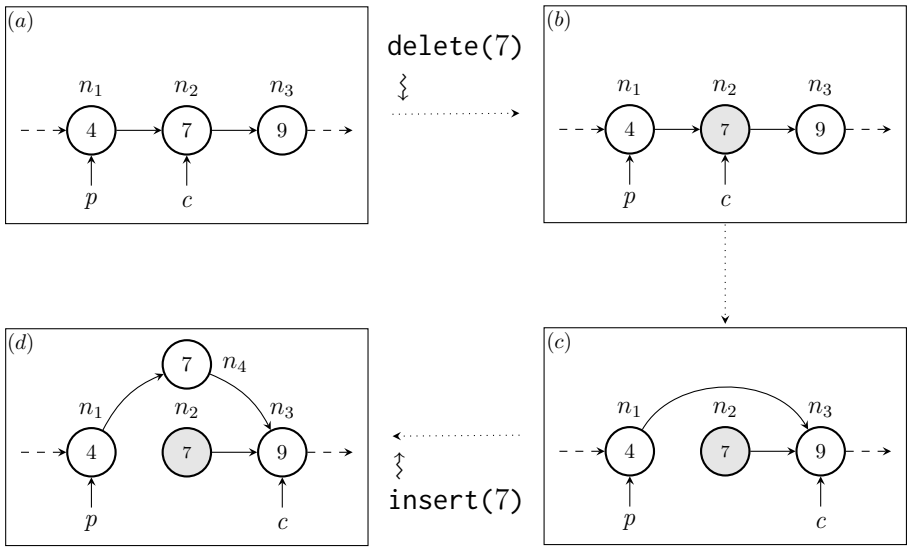


Figure 8.7: Problem execution for the Skiplist template. Boxes represent possible states of `search(7)` on the base level in presence of interference from concurrent `delete(7)` and `insert(7)`.

Now consider **Scenario 2** to be the situation where thread T_d marks n_2 before `eager_i` processes it, as shown in Box (b). Thread T will attempt to unlink n_2, and assuming no further interference, the unlink will result in the structure in Box (c). Thread T will process n_3 next, finding n_3 to be unmarked with key greater than 7, and will terminate with result *false*. So when is the linearization point in this scenario? It cannot be when T finds n_3 unmarked when processing it. Because there could be further interference from thread T_i which inserts key 7 in a new node as shown in Box (d). The new node could be added right before T reads the mark bit of n_3. Thus, when `eager_i` finds n_3 unmarked and returns *false*, key 7 could actually be present in the structure at that point in time.

The linearization point is actually the point in time shown in Box (c), i.e., right after n_2 is unlinked. However, thread T cannot confirm this when n_2 is unlinked because `eager_i` may not terminate at n_3 with *false* as the result. The reason is that by the time T processes n_3, it could get marked in a manner similar to n_2 in Box (b), resulting in the unlinking of n_3 and potentially a restart. That Box (c) is the linearization point is confirmed when T has found n_3 to be unmarked later. The structure maintains the invariant that once a node is marked, it remains marked. Using this invariant, an analysis of thread T's history concludes that n_3 must have been unmarked at the point when n_2 was unlinked. Once `eager_i` terminates at n_3 with *false*, an analysis can *establish in hindsight* that Box (c) indeed was the linearization point.

The discussion above makes clear that the intermediate hindsight specification (HindSpec) developed in Chapter 6 will be a significant step towards proving the client-level specification of the Skiplist template.We are now ready to provide an intuitive proof argument for establishing linearizability of the core operations of the Skiplist template via hindsight.

8.3.2 Proof Outline for Core Operations

Recall that a linearization point is called *modifying* if the operation changes the abstract state of the data structure and *unmodifying* otherwise. In the context of the Skiplist template, succeeding `delete` and `insert` operations exhibit modifying linearization points, while `search` and failing `delete` or `insert` exhibit unmodifying linearization points. The modifying linearization points of the Skiplist template are easier to reason about because they are not future-dependent. For `delete`, the linearization point occurs when `markNode` succeeds, and similarly, for `insert` the linearization point occurs when the call to `changeNext` on line 27 succeeds. The proof strategy for unmodifying linearization points is to combine (KeysetPr) with the $\diamond$ operator from hindsight reasoning. Let us expand on this proof strategy in detail and show why the Skiplist template is linearizable.

We begin by describing the specification for `traverse` that is assumed for analyzing the core operations of the template. Then, we analyze each of the operations in detail. Finally, we show how the eager implementations of `traverse` satisfies the specification that was assumed in the beginning. Along the way, we introduce (as and when necessary) invariants maintained by the Skiplist template that are crucial for proving linearizability.

Specification of `traverse`. The function `traverse` $ps\ cs\ k$ updates arrays ps and cs with predecessor-successor pairs for each level and returns a triple (p, c, res) that satisfies the following past predicate regarding node c: $\diamond(k \in \mathsf{ks}(c) \wedge (res \iff k \in C(c)))$. Recall that our definition of edgesets in Chapter 3 implies the following invariant:

Invariant 1 For all nodes n, if $\mathsf{mark}(n, 0)$ is set to *true* then $\mathsf{ks}(n) = \emptyset$.

Using Invariant 1, we can establish that c is unmarked at the base level at the time point when $k \in \mathsf{ks}(c)$ holds. Note that `traverse` may physically unlink marked nodes. However, this step does not change the abstract state of the structure. Hence, the specification for `traverse` involves no change of the abstract state.

We now consider each of the core operations in detail.

Proof of `search`. Function `search` returns res out of the triple (p, c, res) returned by `traverse`. The specification of `traverse` says $res \iff k \in C(c)$ at some point, say t, during its execution. The specification additionally guarantees $k \in \mathsf{ks}(c)$ at time t. These two facts, combined with the (KeysetPr) at time point t, allow us to immediately infer that res is true iff k was in the structure at that point. Hence, we can establish that $(res \iff k \in C(c))$ was true at some point during the execution of `search`.

Proof of `delete`. We analyze `delete` by case analysis on the value res returned by `traverse`. If res is *false*, then again we can establish that k was not in the structure at some point during `traverse`'s execution by the same reasoning used in the proof of `search`. So let us consider the case that res is *true*. By the specification of `traverse`, we can establish a time point when c was unmarked and contained k. The `delete` operation then calls `maintainanceOp_del` which marks c at all the higher levels. Finally, the `markNode` on Line 15 attempts to mark c at the base level. If `markNode` succeeds, then this step becomes the linearization point of `delete` and k can be considered to be deleted from the structure. But if `markNode` fails, then we gain the knowledge that $\mathsf{mark}(c, 0) = true$. Hindsight reasoning

allows us to infer that c was marked at the base level by a concurrent thread between the end of `traverse` and the invocation of `markNode`. The point right after c was marked by a concurrent thread becomes the linearization point of `delete` in this case, as we can determine that k was not present in the structure at that point.

This hindsight reasoning relies on two facts: first, the key of a node never changes and second, once a mark bit is set to *true* by a successful `markNode` operation (at line 15 in `delete` or line 4 in `maintainanceOp_del`), no other operation will set it back to *false*. In fact, these two facts are invariant for the Skiplist template:

Invariant 2 For all nodes n and level i, once $\mathsf{mark}(n, i)$ is set to *true*, it remains *true*.

Invariant 3 For all nodes n, $\mathsf{key}(n)$ remains constant.

Proof of `insert`. Similar to `delete`, we begin by case analysis on *res* returned by `traverse`. If *res* is true, then we can establish that k was already present in the structure at some point. Otherwise, *res* is *false* and `insert` creates a new node e with key k. Using `changeNext`, an attempt is made to insert node e between nodes p and c. If the attempt succeeds, then k is now part of the structure and this becomes the linearization point. The following `maintainanceOp_ins` operation does not change the abstract state of the structure, and thus, has no effect in terms of linearizability. If the `changeNext` fails, then `insert` simply restarts.

As is evident with the proof outline for the core operations, the specification assumed for `traverse` plays a critical role in case the operation exhibits an unmodifying linearization point. Let us now turn to `traverse` and show how its specification can be proved. We first analyze the eager traversal in detail in the following section. This is followed by the proof argument for the lazy version, which is similar to the one for eager traversal.

8.3.3 Proof Outline for Eager Traversal

As stated earlier, `traverse` returns (p, c, res) such that $\diamond(k \in \mathsf{ks}(c) \wedge (res \iff k \in C(c)))$. Since the returned triple is the result of a call to `eager_i` at the base level, let us begin by analyzing the behavior of this call.

In the sequential setting, the traversal in a search structure maintains the invariant that the search key is always in the inset of the current node. This invariant holds by the design of the Edgeset Framework. Unfortunately, this invariant no longer holds for the Skiplist template in the concurrent setting as evidenced by Box (c) in Figure 8.7. However, we argue first that `eager_i` does maintain the invariant that the search key was in the inset of the current node c between the start of the traversal and the point at which the `eager_i` accesses c. We call this the *inset in hindsight* invariant.

We prove this invariant inductively. We make use of the following locally maintained invariants: (i) At all times, there is one list, denoted the *reachable list*, from the head node that includes all unmarked and some marked nodes. (This list is characterized by the set of nodes with non-empty inset). (ii) The keys in the reachable list are sorted. A consequence

of these two invariants is that if a node n is in the reachable list (whether n is marked or not) and has a key less than k, then k is in the inset of n.

To prove that inset in hindsight is an invariant, we have to show that (a) it is an invariant when `eager_i` takes a step (Line 2) when traversing the base level, and (b) that we can establish inset in hindsight when `eager_rec` initiates `eager_i` (Line 17) at the base level.

To show (a), observe that if a node n becomes unlinked from the reachable list, then it will never again be part of the reachable list. Hence, if n is not in the reachable list when `eager_i` begins executing at the base list, then `eager_i` will never visit n. The contrapositive of this statement allows us to say that if `eager_i` reaches some node c, then it must have been part of the reachable list at some point during the execution of `eager_i`. Additionally, `eager_i` proceeds to the node following c only when $\mathsf{key}(c) < k$. With the help of invariants (i) and (ii) above, we can thus establish that k was in the inset of n at some point.

To show (b), we must do a case analysis on whether node p (Line 16) is marked. If it is unmarked, then it is straightforward to establish that k is in the inset of c currently. However, if p is marked, then we require temporal interpolation based on the following invariant:

Invariant 4 For all nodes n and level i, once $\mathsf{mark}(n, i)$ is set to *true*, $\mathsf{next}(n, i)$ does not change.

This invariant tells us that if p was known to be unmarked in the past, and it is marked currently, then p must have been pointing to c right before it got marked. At that point in time, we can establish that k must have been in the inset of c.

This completes the inductive proof that inset in hindsight is indeed an invariant maintained by the traversal. The inset in hindsight invariant is sufficient to prove the `traverse` specification by the following simple argument. If the traverse encounters k in an unmarked node n, then `traverse` will return *true* as it should. If, by contrast, `traverse` encounters an unmarked node n such that $\mathsf{key}(n) > k$, then by the inset in hindsight invariant, k must have been in the inset of n at some point t in the past and k cannot be in the outset of n (because $\mathsf{key}(n) > k$ and n is unmarked), so therefore k must have been in the keyset of n at time t.

Let us make precise the intuitive proof argument above. We define the traversal invariant $\mathsf{EagerInv}$ for `eager_i` as follows:

$$\mathsf{EagerInv}(k, p, c) \coloneqq \diamond(\mathsf{mark}(p, 0) = \mathit{false} \wedge \mathsf{key}(p) < k) * \diamond(k \in \mathsf{inset}(c))$$

The predicate $\mathsf{EagerInv}$ captures the fact that `eager_i` witnessed node p to be unmarked and its key less than k at some point during its traversal. Additionally, it witnessed that k was in the inset of c at some prior state of the structure. We put the two facts about p and c in separate $\diamond$ predicates because the two facts may hold for different past states.

For the moment, assume that predicate $\mathsf{EagerInv}(k, p, c)$ holds at the beginning of the function call `eager_i` $0\ k\ p\ c$. The proof outline in Figure 8.8 shows that this is sufficient to prove the `traverse` specification. We explain the critical steps of the proof in the following.

The first critical step occurs when c is found to be marked, and the subsequent `changeNext` call on p succeeds (Line 6). Immediately after this step, we can establish that $\mathsf{mark}(p, 0) =$

```
1  {EagerInv(k, p, c)}
2  let eager_i 0 k p c =
3    match findNext 0 c with
4    | cn, true ->
5      match changeNext 0 p c cn with
6      | Success ->
7        {EagerInv(k, p, c) * mark(p, 0) = false * next(p, 0) = cn * key(p) < k}
8        {mark(p, 0) = false * key(p) < k * k ∈ inset(p) * k ∈ outset(p) * k ∈ inset(cn)}
9        {EagerInv(k, p, cn)}
10       eager_i 0 k p cn
11     | Failure -> traverse ps cs k
12   | cn, false ->
13     {EagerInv(k, p, c) * mark(c, 0) = false * next(c, 0) = cn}
       { EagerInv(k, p, c) * mark(c, 0) = false * next(c, 0) = cn
14                  *(key(c) < k  ⟹  k ∈ EagerInv(k, c, cn))
                            *(k ⩽ key(c)  ⟹  ⟐(k ∈ ks(c))) }
15     let kc = getKey c in
16     if kc < k then
17       {EagerInv(k, c, cn)}
18       eager_i 0 k c cn
19     else
20       {⟐(k ∈ ks(c))}
21       let res = (kc = k ? true : false) in
22       (p, c, res)
23       {⟐(k ∈ ks(c) ∧ res  ⟺  k ∈ C(c))}
```

Figure 8.8: Outline for the proof of the `eager_i` for the base level.

false and $\mathsf{next}(p, 0) = cn$ due to the success of `changeNext`. The Invariant 3 allows us to infer $\mathsf{key}(p) < k$ from $\mathsf{EagerInv}(k, p, c)$. In addition, we use the following structural invariant:

Invariant 5 For all nodes n, if $\mathsf{mark}(n, 0)$ is set to *false* then $[\mathsf{key}(n), \infty) \subseteq \mathsf{inset}(n)$. As a consequence of the definition of edgesets and $\mathsf{outset}(n)$ (from Chapter 3), we additionally obtain $\mathsf{outset}(n) = (\mathsf{key}(n), \infty)$.

Invariant 5 helps us establish that $k \in \mathsf{inset}(p)$, $k \in \mathsf{outset}(p)$ and since $\mathsf{next}(p, 0) = cn$, also $k \in \mathsf{inset}(cn)$. Using the rule that p implies $\diamond p$ for an arbitrary proposition p, we can establish $\mathsf{EagerInv}(k, p, cn)$ before the recursive call on Line 10.

The second critical step is when `findNext` determines c to be unmarked (Line 12). The proof goes by case analysis, depending on whether $\mathsf{key}(c) < k$ holds. First consider that this condition is true. Similar to the first critical step, we are able to establish $\mathsf{EagerInv}(k, c, cn)$ here as we have all the relevant facts at this moment. The predicate $\mathsf{EagerInv}(k, c, cn)$ is used for the recursive call on Line 18. Now consider the case that $k \leqslant \mathsf{key}(c)$. Here, we use $\diamond(k \in \mathsf{inset}(c))$ provided by $\mathsf{EagerInv}(k, p, c)$. Let t be the time point when $k \in \mathsf{inset}(c)$ held true. Since c is unmarked currently, c must also have been unmarked at time t due to Invariant 2. And by Invariant 4, we can establish that $k \notin \mathsf{outset}(c)$ at time t. Thus, we conclude that also $k \in \mathsf{ks}(c)$ held at time t. Similarly we conclude that $C(c) = \{\mathsf{key}(c)\}$ as

c is unmarked. Putting this all together, we obtain:

$$k \in C(c) \iff k = \mathsf{key}(c) \iff res.$$

This concludes the proof that predicate EagerInv is indeed an invariant for `eager_i` and is sufficient to prove the `traverse` specification.

In the previous proof, we assumed EagerInv holds when `eager_i` begins executing at the base level. The final piece remaining from the complete proof of the `traverse` specification is to show that `eager_i` collects enough information at higher levels to establish EagerInv before traversing the base level. We explain this below.

It is easy to see that for all higher levels i, `eager_i` can establish $\diamondsuit(\mathsf{mark}(p, i) = \mathit{false} \wedge \mathsf{key}(p) < k)$ when it returns $(p, \_, \_)$. (In fact, EagerInv contains the exact same facts about p and the same argument as in the proof outline for the base level is applicable.) Thus, when a predecessor p is chose from the higher level to initiate `eager_i` at the base level, we know that $\diamondsuit(\mathsf{mark}(p, i) = \mathit{false} \wedge \mathsf{key}(p) < k)$ holds. The current node c is chosen by calling `findNext` 0 p at Line 16 in Figure 8.4. If `findNext` determines p to be unmarked, then we can establish $k \in \mathsf{inset}(c)$ using the fact $\diamondsuit(\mathsf{key}(p) < k)$ and Invariant 4. However, if p is found to be marked, then it is a bit tricky to establish that $\diamondsuit(k \in \mathsf{inset}(c))$. We require the following additional structural invariant.

Invariant 6 For all nodes n, if there exists an i such that $\mathsf{mark}(n, i)$ is set to *false*, then $\mathsf{mark}(n, 0)$ is set to *false*.

Invariant 5 captures the fact that the base level gets marked at the end. This fact is useful for us because we can combine it with $\diamondsuit(\mathsf{mark}(p, i) = \mathit{false})$ to obtain $\diamondsuit(\mathsf{mark}(p, 0) = \mathit{false})$. Since `findNext` has found p to be marked at the base level, we can determine that it got marked at some time point between the point in time when $\diamondsuit(\mathsf{mark}(p, 0) = \mathit{false})$ held and the present. Let t be the time point right before p got marked. Invariant 4 gives us useful information about which node $\mathsf{next}(p, 0)$ can be pointing to at time t. It preceisely says that $\mathsf{next}(p, 0) = c$ at time t. Thus, we have found a moment in time when $\mathsf{mark}(p) = \mathit{false}$, $\mathsf{key}(p) < k$ and $\mathsf{next}(p, 0) = c$. These three facts allow us to immediately conclude that $k \in \mathsf{inset}(c)$ at time t. This finally completes the proof of `traverse`.

8.3.4 Proof Outline for Lazy Traversal

The proof idea for the lazy traversal is similar to that of the eager traversal. We provide a brief overview of the traversal invariant for `lazy_i`, and why it is sufficient to establish the required specification for `traverse`. The following discussion focuses on the `lazy_i` executing on the base level, for sake of simplicity.

The traversal invariant for `lazy_i` is defined as below:

$$\begin{aligned}\mathsf{LazyInv}(k,p,pn,c) &\coloneqq \mathsf{EagerInv}(k,p,pn) * (pn \neq c \Rightarrow \exists\ ls.\ \mathsf{MarkedSeg}(pn,ls,c))\\ \mathsf{MarkedSeg}(pn,ls,c) &\coloneqq (ls = [\,] \Rightarrow \diamond(\mathsf{next}(pn,i) = c))\\ &\quad * (ls = [n_0,\ldots,n_l] \Rightarrow \diamond(\mathsf{next}(pn,0) = n_0)\\ &\qquad * (\forall\ 0 \leqslant j < l.\ \diamond\ \mathsf{next}(n_j,0) = n_{j+1}))\\ &\qquad * \diamond(\mathsf{next}(n_l,0) = c)\end{aligned}$$

The invariant LazyInv relies on EagerInv and a new predicate MarkedSeg that stores the segment of marked nodes. Additionally, the predicate establishes that the nodes pn and c are the start and end points of the marked segment.

To see that LazyInv is indeed an invariant, consider Lines 3 and 7 where `lazy_i` is recursively called. At Line 3, the marked segment can be extended by including the marked node c, while p and pn remain unchanged. On the other hand, at Line 7, the second conjunct of LazyInv becomes trivially true, and $\mathsf{EagerInv}(k, c, cn)$ can be established in the same way as for `eager_i`.

Let us now show why LazyInv is sufficient to prove the `traverse` specification. Consider the points where `lazy_i` terminates (lines 11 and 17). When terminating at Line 11, we know that $pn = c$. Hence, LazyInv provides the predicate $\mathsf{EagerInv}(k, p, c)$. Using the same argument as `eager_i`, we can establish the required postcondition. Now consider Line 17. In this case, the thread has successfully unlinked the marked segment, hence it can establish that $\mathsf{mark}(p, 0) = \mathit{false}$ and $\mathsf{next}(p, 0) = c$ right after the call to `changeNext`. Note that the later check whether c is unmarked must succeed in order for `lazy_i` to terminate. Hence, at the point of the successful `changeNext`, c must also be unmarked, giving us all the facts necessary to establish the required postcondition. This completes the proof of `lazy_i`.

8.4 Verifying the Skiplist Template

We relate the intuitive proof argument from Section 8.3 to the development on hindsight reasoning in Iris in Section 6.3 to obtain a complete proof of the Skiplist template. To achieve this, we must perform three tasks required by the proof method in Section 6.3. The first task is to determine the decisive operations that potentially alter the structure, and resolve the prophecy around those operations. As discussed previously, the decisive operations are `markNode` for `delete` and `changeNext` for `insert`. The `search` operation does not modify the abstract state and hence, it has no decisive operation.

The second task is to define a snapshot in the context of the Skiplist template and instantiate Inv_{tpl} appropriately. This includes the predicate resources that ties the concrete state of the structure to the latest snapshot, as well as invariants that allow temporal interpolation. The third and the final task is to prove the hindsight specification for the core operations.

In this section we focus on the second task of defining the snapshot and providing invariants necessary to formalize the intuitive proof argument. Once, we have set up the right

invariants, the formalized proof follows the intuitive proof very closely. We explain this with `delete` as an example.

8.4.1 Snapshot and the Skiplist Template Invariant

Recall that the notion of keysets are central to the intuitive proof argument for the core operations of the Skiplist template. Hence, a snapshot of the structure must contain information about the keysets. For encoding keysets in Iris, we borrow heavily from [97], especially the representation of keysets via the Flow Framework (from Section 3.6). In addition, we use the keyset RA from Chapter 5.

We define the snapshot of the Skiplist template as a tuple containing the following components:

- the set of nodes N comprising the structure (also referred to as the *footprint* below)
- the abstract state of the structure (a set of keys)
- the mark bits (a map from N to $\mathbb{N} \to \mathsf{Bool}$, i.e., a Boolean per level)
- the next pointers (a map from N to $\mathbb{N} \to N$)
- the keys (a map from N to K)
- the height of nodes (a map from N to $\mathbb{N}$)
- the representation of flow values

We reparameterize the $\mathsf{mark}(n, i)$ function introduced earlier to take the snapshot as an argument. Thus, we use $\mathsf{mark}(s, n, i)$ to mean the mark bit of node n at level i in snapshot s. We redefine $\mathsf{next}(\cdot)$, $\mathsf{key}(\cdot)$, $\mathsf{ks}(\cdot)$ and other such functions similarly by adding the snapshot s as an additional parameter. We also use $\mathsf{FP}(s)$ to represent the footprint of the snapshot s.

We now present the Skiplist template invariant in Figure 8.9. The $\mathsf{resources}$ predicate ties the snapshot to the concrete state through an intermediary node-level predicate $\mathsf{Node}(n, k, h, mk, nx)$. This predicate actually ties the physical representation of a node in the heap to the abstract quantities ($\mathsf{key}(\cdot)$, $\mathsf{height}(\cdot)$, $\mathsf{mark}(\cdot)$ and $\mathsf{next}(\cdot)$, respectively) that the Skiplist template relies on. The Node predicate also owns all the resources needed to execute the helper functions. The Skiplist template proof is parametric in the definition of Node. Thus, we achieve proof reuse across skiplist variants that follow the same high-level skiplist algorithm, but implement the node differently. We provide more details on this matter later. We discuss some concrete node implementations in Chapter 9.

The predicate $\mathsf{resources\_keyset}(s)$ capture the ownership resources required for keyset reasoning. Using the ghost resources in Iris and the keyset camera from [97], it ensures that the keysets and the logical contents of nodes in s satisfy (KeysetPr). The assertion $\boxed{\bullet(\mathbb{K}, C(s))}^{\gamma}$ signifies ownership of the logical contents $C(s)$ of the structure in the state captured by snapshot s. Recall that $\mathbb{K}$ is the set of all keys. Each individual node $n \in$

$$
\begin{aligned}
\mathsf{Inv}_{tpl}(r, H, T) &:= \mathsf{resources}(r, H(T)) \\
&\quad * (\forall t,\ 0 \leqslant t \leqslant T \Rightarrow \mathsf{per\_snapshot}(H(t))) \\
&\quad * (\forall t,\ 0 \leqslant t < T \Rightarrow \mathsf{transition\_inv}(H(t), H(t+1))) \\
\mathsf{resources}(s) &:= \mathop{\Large *}_{n \in \mathsf{FP}(s)} \mathsf{Node}(n, \mathsf{mark}(s,n), \mathsf{next}(s,n), \mathsf{key}(s,n), \mathsf{height}(s,n)) \\
&\quad * \mathsf{resources\_keyset}(s) \\
\mathsf{resources\_keyset}(s) &:= \boxed{\bullet(\mathbb{K}, C(s))}^{\gamma} * \mathop{\Large *}_{n \in \mathsf{FP}(s)} \boxed{\circ(\mathsf{ks}(s,n), C(s,n))}^{\gamma} \\
\mathsf{transition\_inv}(s, s') &:= (\mathsf{FP}(s) \subseteq \mathsf{FP}(s')) \\
&\quad * (\forall n,\ \mathsf{key}(s',n) = \mathsf{key}(s,n) \wedge \mathsf{height}(s',n) = \mathsf{height}(s,n)) \\
&\quad * (\forall n\ i,\ \mathsf{mark}(s,n,i) = \mathit{true} \Rightarrow \mathsf{mark}(s',n,i) = \mathit{true}) \\
&\quad * (\forall n\ i,\ \mathsf{mark}(s,n,i) = \mathit{true} \Rightarrow \mathsf{next}(s',n,i) = \mathsf{next}(s,n,i))
\end{aligned}
$$

Figure 8.9: Instantiating Inv_{tpl} with invariants of the Skiplist template.

$\mathsf{FP}(s)$ owns its keyset $\mathsf{ks}(s,n)$ and node-local logical contents $C(s,n)$. The keyset camera enforces that the $\mathsf{ks}(s,n)$ are pairwise disjoint, $C(s,n) \subseteq \mathsf{ks}(s,n)$ for each n, and that $C(s) = \bigcup_{n \in \mathsf{FP}(s)} C(s,n)$, respectively, $\mathsf{K} = \bigcup_{n \in \mathsf{FP}(s)} \mathsf{ks}(s,n)$.

The predicate $\mathsf{per\_snapshot}$ captures structural invariants that hold for all snapshots recorded in the history. This includes invariants of three kinds: first, invariants to ensure that each component of the snapshot is of the correct type and the maps (from nodes to mark bits, next pointers, etc.) are defined for all nodes in the footprint; second, the node-level invariants relating the node's inset, outset, mark bit, etc (like Invariant 1); and third, invariants about the hd and tl nodes, such as $\mathsf{key}(s, hd) = -\infty$, $\mathsf{height}(tl) = L$, etc.

The predicate $\mathsf{transition\_inv}(s, s')$ captures invariants about how certain quantities evolve over time, such as that mark bits once set to true remain true. The invariants 2, 3, and 6 presented in Section 8.3 are part of this predicate. These invariants form the crux of the hindsight reasoning, as they enable temporal interpolation.

Helper Function Specifications. Before we go into the formal proof argument for `delete`, we must discuss how to reason about the node-level helper functions. Figure 8.10 shows the specification for the helper functions assumed by the Skiplist template. The specifications are logically atomic, i.e., they behave like a single atomic step in the template. The preconditions for all of the functions rely solely on the predicate Node. The functions `getKey`, `getHeight` and `findNext` read various components of the node. Note that `findNext` reads both the mark bit and the next pointer together.

The specification for functions `markNode` and `changeNext` is slightly more complex because they potentially change the structure. Our encoding of the hindsight specification (HindSpec) requires us to resolve a prophecy around each program step that may modify the structure.

$\langle\, k\, h\, mk\, nx.\ \mathsf{Node}(n,k,h,mk,nx)\,\rangle$ `getKey` n $\langle\, k.\ \mathsf{Node}(n,k,h,mk,nx)\,\rangle$

$\langle\, k\, h\, mk\, nx.\ \mathsf{Node}(n,k,h,mk,nx)\,\rangle$ `getHeight` n $\langle\, h.\ \mathsf{Node}(n,k,h,mk,nx)\,\rangle$

$\langle\, k\, h\, mk\, nx.\ \mathsf{Node}(n,k,h,mk,nx) * (i < h)\,\rangle$ `findNext` i n $\langle\, n'.\ \mathsf{Node}(n,k,h,mk,nx) * (nx(i) = n')\,\rangle$

$\langle\, k\, h\, mk\, nx.\ \mathsf{Node}(n,k,h,mk,nx) * (i < h) * \mathsf{Proph}(p, pvs) * (i = 0)\,\rangle$

`markNode` i n p

$$\left\langle \begin{array}{r} x.\ \mathsf{Node}(n,k,h,mk',nx) * \mathsf{Proph}(p,pvs') * (pvs = prf \mathbin{+\!\!+} pvs') \\ * (mk(i) = false \Rightarrow x = \mathsf{Success} * mk' = mk[i \rightarrowtail true] * \mathsf{Upd}(pvs)) \\ * (mk(i) = true \Rightarrow x = \mathsf{Failure} * mk' = mk * \neg \mathsf{Upd}(prf)) \end{array} \right\rangle$$

$\langle\, k\, h\, mk\, nx.\ \mathsf{Node}(n,k,h,mk,nx) * (i < h) * \mathsf{Proph}(p, pvs) * (i = 0)\,\rangle$

`changeNext` i n n' e p

$$\left\langle \begin{array}{r} x.\ \mathsf{Node}(n,k,h,mk,nx') * \mathsf{Proph}(p,pvs') * (pvs = prf \mathbin{+\!\!+} pvs') \\ * ((mk(i) = false \wedge nx(i) = n') \Rightarrow x = \mathsf{Success} * mk' = mk[i \rightarrowtail true] * \mathsf{Upd}(pvs)) \\ * ((mk(i) = true \vee nx(i) \neq n') \Rightarrow x = \mathsf{Failure} * mk' = mk * \neg \mathsf{Upd}(prf)) \end{array} \right\rangle$$

Figure 8.10: Specifications of the helper functions used by the Skiplist template.

In the context of the Skiplist template, the structure is modified using calls to the helper functions `markNode` and `changeNext` on the base-level list. Unfortunately, prophecies in Iris can only be resolved around physical computation steps in the Iris programming language HeapLang and not at a more abstract level around logical atomic triples. Hence, the atomic triples for `markNode` and `changeNext` themselves must capture the relevant aspects of the reasoning related to prophecy resolution.

We focus on the `markNode` specification in particular. The assertions in blue described possible changes to the node-level quantities. For `markNode` on node n at level i, the return value (`Success` or `Failure`) is determined by whether n is already marked at i. If it is, then the function returns `Failure` without modifying the node. If it is unmarked, then `markNode` successfully marks it, and updates the node accordingly. The additional assertions in brown are concerned with the prophecy reasoning and applicable only when `markNode` is called on the base level. The precondition $\mathsf{Proph}(p, pvs)$ gives the right to resolve the prophecy p. The additional postcondition says that `marknode` resolves the prophecy an arbitrary number of times, but if it succeeds, then a `Success` value was seen in the list of prophesied values pvs, yielding $\mathsf{Upd}(pvs)$. Otherwise, it says that no `Success` value was seen so far, yielding $\mathsf{Upd}(prf)$. The template proof for `delete` then uses this information to infer whether $\mathsf{Upd}(pvs)$ holds and prove the appropriate postcondition from the hindsight specification. The specification for `changeNext` can be interpreted similarly. Here, the return value hinges upon the mark bit being false and the next pointer of n pointing to n' at i.

8.4.2 Proof of `delete`

In this section, we discuss the proof of `code` in more detail. The proof outline is shown in Figure 8.11. We want to show that `delete` satisfies (HindSpec). The precondition provides

access to the invariant $\mathsf{Inv}(r)$ and knowledge that the thread ID is *tid* with start time t_0. Additionally, the precondition provides the conjunct $(\mathsf{Upd}(pvs) \mathrel{-\!\!*} \mathsf{AU}(\Phi))$, signifying that the thread can perform its linearization in case of a successful decisive operation. The precondition also provides the right to resolve prophecies. It is easy to see how prophecies are manipulated by the helper functions from the specifications in Figure 8.10, hence we ignore them in the proof outline.

The algorithm of `delete` begins with a call to `traverse`. The specification for `traverse` provides the postcondition

$$\diamondsuit_{s,t_0}(k \in \mathsf{ks}(s,c) \land (res \iff k \in C(s,c))).$$

Note that we use the past operator $\diamondsuit_{s,t_0}$ from Section 6.4.2 in the postcondition of `traverse`. Intuitively, this assertion captures that there is a past state s in the history (after time point t_0) in which k is in the keyset of c and *res* is true iff k is in the logical contents of c. Note that this implies $\mathsf{mark}(s, c, 0) = \mathit{false}$ due to Invariant 1.

The `delete` algorithm proceeds by case analysis on the result of `traverse`. Let us first consider the case that *res* is *false* (Line 8). The `delete` operation terminates with result *false*. The specification (HindSpec) requires establishing the predicate $\mathsf{PastLin}(\mathtt{del}, k, \mathit{false}, t_0)$ in this case. To this end, we must provide a witness past state in which k was not part of the abstract state. We obtain this witness from the postcondition of `traverse` which provides a past state s such that $k \in \mathsf{ks}(s,c)$ and $k \notin C(s,c)$. Applying (KeysetPr) in state s, we can establish the predicate $\mathsf{PastLin}(\mathtt{del}, k, \mathit{false}, t_0)$ (Line 12).

Let us now consider the case when *res* is *true*. The `delete` algorithm here calls the helper function `maintainanceOp_del`, which marks node c at the higher levels. Thus, after `maintainanceOp_del` terminates, we obtain the information that all higher levels of c are marked (Line 16). Next, the helper function `markNode` is called at the base level to logically delete c from the structure. In order to apply the specification of `markNode`, we open the invariant $\mathsf{Inv}(r)$ to access the Node predicate for c (Line 18). Let the current snapshot at this point (obtained by opening the invariant) be s_0. The call to `markNode` call either succeeds or fails depending on whether $\mathsf{mark}(s_0, c, 0) = \mathit{false}$ holds. We consider both of the cases in turn.

First, suppose that `markNode` succeeds (Line 21). This implies that $\mathsf{mark}(s_0, c, 0) = \mathit{false}$ and a new snapshot s_1 must be constructed to record the marking of c. The proof then appends s_1 to the history stored in $\mathsf{Inv}(r)$ in order to reestablish $\mathsf{Inv}(r)$. Additionally, success of `markNode` implies that $\mathsf{Upd}(pvs)$ holds. Hence, the thread must use the resource $\mathsf{AU}(\Phi)$ to linearize itself. The receipt of linearization Φ is exactly what is required in the postcondition of (HindSpec) in this case (Line 25).

Finally, consider the case that `markNode` fails (Line 27). The `delete` algorithm terminates with result *false* in this case. Hence, we must again establish $\mathsf{PastLin}(\mathtt{del}, k, \mathit{false}, t_0)$ to complete the proof. That is, we must identify a witness past state such that k was not part of the abstract state of the structure. To compute this witness, note that $\mathsf{mark}(s, c, 0) = \mathit{false}$ for state s observed by `traverse`, while $\mathsf{mark}(s_0, c, 0) = \mathit{true}$ for the current state s_0. By transition_inv in Figure 8.9 and temporal interpolation, there must exist consecutive states

s_1 and s_2 when c was marked at the base level (Line 28). Here, the predicate $\mathsf{Past}(s, t)$ represents the knowledge that the history recorded snapshot s at time t. The snapshot s_2 is the required witness to establish the postcondition of (HindSpec) (Line 32). This completes the proof of `delete`.

$\left\{\boxed{\mathsf{Inv}(r)} * \mathsf{Thread}(tid, t_0) * (\mathsf{Upd}(pvs) \mathrel{-\!\!*} \mathsf{AU}(\Phi))\right\}$
let delete k =
  let ps = allocArr L hd **in**
  let cs = allocArr L tl **in**
  let p, c, res = traverse ps cs k **in**
  $\left\{\diamond_{s,t_0}(k \in \mathsf{ks}(s,c) \wedge (res \iff k \in C(s,c)))\right\}$
  if not res **then**
    $\left\{\diamond_{s,t_0}(k \in \mathsf{ks}(s,c) \wedge (k \notin C(s,c)))\right\}$
    $\left\{\diamond_{s,t_0}(k \notin |s|)\right\}$
    $\left\{\diamond_{s,t_0}(\Psi_{del}(k, |s|, |s|, false))\right\}$
    $false$
    $\left\{\neg\mathsf{Upd}(pvs) * \mathsf{PastLin}(\mathtt{del}, k, false, t_0)\right\}$
  else
    $\left\{\diamond_{s,t_0}(k \in \mathsf{ks}(s,c) \wedge (k \in C(s,c)))\right\}$
    maintainanceOp_del c;
    $\left\{\diamond_{s,t_0}(\mathsf{mark}(s,c,0) = false) * \diamond_{s',t_0}(\forall i, 0 < i < \mathsf{height}(s',c) \Rightarrow \mathsf{mark}(s',c,i) = true)\right\}$
    $\left\{\diamond_{s,t_0}(\mathsf{mark}(s,c,0) = false) * \mathsf{Hist}(H_0, T_0) * \mathsf{resources}(s_0) * \cdots\right\}$
    $\left\{\diamond_{s,t_0}(\mathsf{mark}(s,c,0) = false) * \mathsf{Hist}(H_0, T_0) * \mathsf{Node}(c, s_0) * \cdots\right\}$
    match markNode 0 c **with**
    | Success ->
      $\left\{(\mathsf{mark}(s_0,c,0) = false) * (\mathsf{mark}(s_1,c,0) = true) * \mathsf{Upd}(pvs) * \cdots\right\}$
      $\left\{\mathsf{Upd}(pvs) * \mathsf{AU}(\Phi) * \Psi_{del}(k, |s_0|, |s_1|, true) * \cdots\right\}$
      $\left\{\mathsf{Upd}(pvs) * \Phi * \mathsf{Hist}(H_0[T_0+1 \rightarrowtail s_1], T_0+1) * \mathsf{resources}(s_1) * \cdots\right\}$
      traverse ps cs k; $true$
      $\left\{\mathsf{Upd}(pvs) * \Phi\right\}$
    | Failure ->
      $\left\{\diamond_{s,t_0}(\mathsf{mark}(s,c,0) = false) * (\mathsf{mark}(s_0,c,0) = true) * \cdots\right\}$
      $\left\{\begin{array}{l}(\exists\ s_1\ s_2\ t,\ (t_0 \le t) * \mathsf{Past}(s_1,t) * \mathsf{Past}(s_2,t+1)) \\ *(\mathsf{mark}(s_1,c,0) = false) * (\mathsf{mark}(s_2,c,0) = true)) * \cdots\end{array}\right\}$
      $\left\{(t_0 \le t) * \mathsf{Past}(s_1,t) * \mathsf{Past}(s_2,t+1) * (\mathsf{key}(s_2,c) \notin |s_2|) * \cdots\right\}$
      $\left\{\diamond_{s_2,t_0}(\Psi_{del}(k, |s_2|, |s_2|, false)) * \cdots\right\}$
      $false$
      $\left\{\neg\mathsf{Upd}(pvs) * \mathsf{PastLin}(\mathtt{del}, k, false, t_0)\right\}$

Figure 8.11: Outline for the proof of the delete.

9 Proof Mechanization and Evaluation

This chapter sheds light on the mechanization of the techniques and proofs in Chapters 5 to 8. The full development of our mechanization effort is available online[1]. These proofs have been mechanically checked using the Coq proof assistant, building on the formalization of Iris [86, 94, 96].

For the Skiplist template, we have derived and verified implementations based on the Herlihy-Shavit skiplist algorithm [68, § 14], the Michael set [122] and the Harris list algorithm [62]. For the LSM-DAG template, we instantiatiate an implementation inspired by Google's LevelDB [58].

The organization of our proofs is shown in Figure 9.1. Going from left to right, the first column relates to the formalization of hindsight reasoning in Iris. The box "Hindsight" captures the assumptions regarding the hindsight specification from Section 6.3. These assumptions not only include the hindsight specification itself but also the relevant definitions of snapshots, histories, etc. The module "Client-level Spec" relates the client-level specification expressed in terms of atomic triples to the hindsight specification used for the template-level proofs. The corresponding proof involves the reasoning about prophecies and the helping protocol, which is done once and for all and applicable to all data structures that fulfill the assumptions made in the "Hindsight" module.

The middle column consists of modules for the verified templates and the associated proofs verifying the template operations against the hindsight specification. We discuss them in detail next.

9.1 Skiplist template case study

All of the proofs we discuss below are mechanized in Iris/Coq. The templates, traversals and the node implementations are written in Iris's default programming language HeapLang. In order to correctly capture the dependence between different layers of the proofs (such as hindsight specification and the templates, the templates and the `traverse`/node implementations), we heavily make use of Coq's module system.

[1] https://github.com/nyu-acsys/template-proofs/tree/lockfree

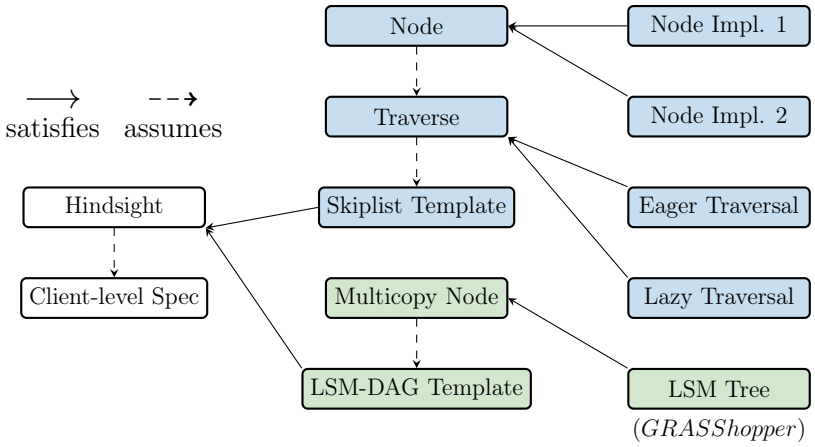


Figure 9.1: Proof organization with additional details. Each box represents a collection of modules relevant to the label. The dashed arrows represent module dependence, i.e., assumption of specifications. The normal arrows represent implementation of the target module (fulfillment of the assumptions).

The Skiplist template, as described in Figure 8.2, abstracts from the concrete implementations of nodes and the `traverse` operation. Hence, we package their specifications into separate modules. To ensure that the specified data structure invariant for the Skiplist template is not vacuous, we also verified an `init` routine that initializes the data structure and establishes the invariant.

The final column shows modules for the two node implementations of the Skiplist template, as well as the eager and lazy traversal discussed in Section 8.2. The helper functions `markNode` and `changeNext` are implemented using an atomic CAS operation in both of the node implementations. The crux of the node implementation for the Skiplist template is to determine a memory representation of the mark bit and the next pointer (at some level) such that both values can be read or written together with one atomic CAS operation. The first node implementation does this by using a sum type. The second node implementation is conceptually similar but uses more low-level data types instead of a sum type.

The traversal and node implementations above correspond to several existing lock-free (skip)list algorithms from the literature. The Herlihy-Shavit skiplist algorithm [68, § 14] is obtained by instantiating our template with the eager traversal, the node implementation 2, and maintenance operations that link higher-level nodes in increasing order of level and unlink nodes in the opposite order. The Michael set [122] is obtained as a degenerate case of the Herlihy-Shavit template instantiation where the skiplist is restricted to $\mathsf{L} = 2$ (For $\mathsf{L} = 2$, Level 1 consists of only a fixed single edge between the sentinel nodes. So, conceptually, Level 1 can be ignored in this case.)

We obtain a novel variant of a skiplist by replacing the eager traversal in the Herlihy-Shavit instantiation with the lazy traversal. The lazy traversal is inspired by the Harris list algorithm [62], which is obtained as a degenerate case of this new lazy skiplist algorithm by restricting it to $\mathsf{L} = 2$.

We present a summary of the proof effort for the Skiplist template in Table 9.1. The

Skiplist Template (Iris/Coq)

Module	Code	Proof	Total	Time(*secs*)
Flow Library	0	5330	5330	33
Hindsight	0	950	950	11
Client-level Spec	9	329	338	18
Skiplist	12	1693	1705	26
Skiplist Init(∗)	6	319	325	15
Skiplist Search(∗)	7	62	69	6
Skiplist Insert(∗)	37	3457	3494	111
Skiplist Delete(∗)	28	2401	2429	72
Node Impl. 1	118	908	1026	35
Node Impl. 2	106	836	942	35
Eager Traversal	38	1165	1203	96
Lazy Traversal	47	2063	2110	145
Total	**408**	**19513**	**19921**	**603**
Herlihy-Shavit	243	11212	11455	390

Table 9.1: Summary of the proof effort for the Skiplist template. For each module, we show the number of lines of program code, lines of proof, total number of lines, and the proof-checking time in seconds. The code for the initialization and the core operations of the skiplist (entries with (∗)) is technically defined in the "Skiplist" module, however here we present them separately for each operation to provide a better picture. The count for Herlihy-Shavit is the summation of rows "Hindsight", "Client-level Spec", all "Skiplist" modules, "Node Impl. 2" and "Eager Traversal".

proof-checking time was measured on the Docker image running on an Apple M1 Pro chip with 16GB RAM. The flow library contains the Iris formalization of the Flow Framework developed in [97, 138]. As a minor contribution, we extend this library with general lemmas for reasoning about graph updates that have an affect on an unbounded number of nodes. These lemmas are useful for the proofs of `insert`, `delete` and lazy `traverse`. The unbounded updates, as well as the maintenance operations, are the reason for the relatively high number of proof lines for the `insert` and `delete` operations.

9.2 LSM-DAG TEMPLATE CASE STUDY

The LSM-DAG was verified twice during the course of this dissertation. It was verified first in [138] which focused on verification of multicopy structures. This approach proposed the search recency (from Section 7.5) as an intermediate template-level specification in a fashion similar to the hindsight specification. The LSM-DAG template was reverified in [139] using the hindsight framework presented in Chapter 6. Table 9.2 presents a comparison of the proof effort using the two approaches for the purpose of evaluating the hindsight framework.

Table 9.2 presents a detailed comparison of the multicopy template proofs from [138] versus the new proof based on the hindsight framework. The original proof consists of a

Multicopy Template (Iris/Coq)		
Module	**Original**	**Hindsight**
Defs	866	(950)
Client-level Spec	434	(338)
LSM	741	540
Search	411	399
Upsert	327	371
Total	**2779**	**1310**

Table 9.2: Comparison of multicopy template proofs. The column "Original" shows the number of lines from the proofs in [138], while "Hindsight" shows them for our new proof effort. Module "Defs" represents definitions required for proving the client-level specification (helping invariant, history predicate, etc). Module "Client-level Spec" contains the proof relating the intermediate specification (Search Recency Specification from [138] and Hindsight Specification in this dissertation) to the client specification. Module "LSM" contains definitions required to instantiate the frameworks for LSM trees. Modules "Search" and "Upsert" refer to the proofs for the search and upsert operations, respectively. Entries in '()' for the 'Hindsight' column are not included in the total due to being part of the hindsight library.

total of 2779 lines. By contrast, the definitions ("Defs") and "Client-level Spec" proofs can be factored out of the total cost of the hindsight-based proof, because it is part of the hindsight library itself. Hence, the new proof based on hindsight reasoning consists of only 1310 lines, which is a reduction of 53%. To summarize, the improvement stems from the fact that the original proof relies on an intermediate specification and a helping protocol that is tailored to multicopy structures, while our new proof uses a helping protocol that is shared among all proofs that build on the new hindsight proof method.

While the majority of the reduction in the proof size stems from the elimination of structure-specific specifications and helping protocol proofs, we also saw a minor reduction in the size of the remainder of the proof. One outlier is the proof of `upsert`. Here, the increase is attributed to the fact that the proof has to construct a fresh snapshot when the operation succeeds. However, this construction is conceptually simple and could be factored out into more abstract lemmas that are provided directly by the hindsight library.

The LSM-DAG template, as shown Figure 9.1, parameterizes over the (multicopy) node implementations. Our implementation uses an unsorted array to store key-timestamp pairs for the (in-memory) root node (with upserts adding to one end of the array), and a read-only sorted array (also known as a sorted string table [58]) for the other (on-disk) nodes. This array models the contents of a file. The implementation uses a library of utility functions and lemmas for arrays that represent partial maps from keys to values.

We verify both the helper functions for the core search structure operations (Figure 7.3) as well as those needed by the maintenance template (Figure 7.8). Each operation demuxes between the code for in-memory and on-disk nodes based on the reference to the operation node. For instance, in the case of `mergeContents` $r\, n\, m$, if $r = n$ then the operation flushes the in-memory node n to the on-disk node m. Otherwise, both n and m must be on-disk nodes, which are then compacted. Alternatively, one could use separate implementations of

each helper function for the two types of nodes. The polymorphism could then be resolved statically by unfolding the recursion in the template algorithms once, letting helper function calls in the unfolded iteration go to the in-memory versions and all remaining ones to the on-disk versions.

Note that the helper functions are invoked by the LSM-DAG template under a lock. We take advantage of this fact in [138] by using SMT-based tool GRASShopper for verifying the node implementation. The verification effort in [138] also verified the maintenance operation `compact`. The maintenance operation satisfes the trivial specification of not changing the abstract state of the structure. Proofs of both, the node implementation as well as `compact`, do not benefit from the hindsight reasoning. Hence, we ignore it for our case-study and instead focus solely on the core operations for comparison.

Part III: Discussion

10 | Related Work

This chapter discusses prior work related to the topics covered in this dissertation. Section 10.1 gives a broad overview of the verification techniques for concurrent programs and situates our techniques in the landscape. Section 10.2 discusses work related to the notion of template algorithms. Section 10.3 discusses work relating to hindsight reasoning. Finally, Section 10.4 and Section 10.5 discuss verification efforts targeting skiplists and multicopy structures.

10.1 Deductive verification of concurrent programs

In this dissertation, we follow a deductive verification approach, which reduces reasoning about program correctness to discharging proof obligations expressed in a formal logic. Here, the program code is annotated with intermediate assertions that enable complex proof obligations to be mechanically decomposed into simpler ones according to program structure. This idea was pioneered by Turing [161] and later cast into formal reasoning systems, starting with the development of (Floyd-)Hoare logic [50, 72] and its mechanization due to Dijkstra [38]. Since then, many formal systems have been proposed that enable compositional reasoning along dimensions of program complexity that are orthogonal to syntactic structure, including concurrency [70, 80, 103, 136], data representation [32, 73], and time [141]. A number of textbooks provide detailed introductions to the relevant topics (see e.g. [113, 140, 168]).

Separation logic [75, 132, 144] is an extension of Hoare logic that was originally conceived to deal with the complexities imposed by mutable state. Separating conjunction and the accompanying frame rule enable "reasoning and specification to be confined to the cells that the program actually accesses" [132]. O'Hearn soon realized that these reasoning principles naturally extend to concurrent programs that manipulate shared resources. This led to the development of concurrent separation logic [20, 129], which has spawned a proliferation of logics that provide a sophisticated arsenal of modular reasoning techniques [15, 33, 39, 40, 48, 60, 71, 86, 127, 142, 164, 171]. For a more comprehensive survey and discussion of the history of this development we refer the reader to [19, 131].

We have formalized the verification of our template algorithms in the concurrent separation logic Iris [82, 83, 86, 95]. Our formalization particularly benefits from Iris's support for logically atomic triples [33, 54, 76, 86] and user-definable resource algebras, which can

capture nontrivial ghost state such as keysets and flow interfaces. However, we expect that our methodology can be replicated in other concurrent separation logics that support these features, such as FCSL [149], which would also be useful if one wanted to extend the template-based approach to verifying non-linearizable data structures [151].

We make use of Iris's support for prophecy variables [85] to reason modularly about the non-local dynamic linearization points of searches in multicopy structures. Our proof discussed in Chapter 6 builds on the prophecy-based Iris proof of the RDCSS data structure from [85] and generalizes it by adapting the helping protocol to consider the sequential specification of the data structure. The idea of using prophecy variables to reason about non-fixed linearization points has also been explored in prior work building on other logics than Iris [163, 173], including situations that involve unbounded helping [111, 162]. A possible alternative approach to using prophecies is to prove that the template-level atomic specification contextually refines the client-level atomic specification of multicopy structures using a relational program logic [9, 54, 111].

Separating conjunction in Iris is affine, which means that it can be weakened by dropping one of its conjuncts. That is, the resources in an assertion can be "thrown away" without invalidating the assertion. In particular, this applies to resources that express ownership of allocated memory. Consequently, Iris cannot be used directly to reason about absence of memory leaks. Iron [13], a recent extension of Iris, allows proving absence of memory leaks in the context of manual memory management. In this dissertation, we assume a garbage collected environment in our proofs. We argue below that this is a reasonable assumption, but relaxing this assumption is certainly attractive.

Another limitation of Iris is that it assumes that memory reads and writes are sequentially consistent. The logic lacks support for reasoning about the weaker consistency notions supported by modern hardware architectures. Several recent projects explore extensions to Iris in order to accommodate weak memory models [34, 61, 87, 158, 166]. The LSM-DAG template presented in this dissertation is lock-based and guarantees data race freedom, provided the locks are implemented correctly. Thus, reasoning about weak consistency can be confined to the verification of the lock implementation. The Skiplist template on the other hand is lock-free, and will require tackling new challenges arising from the underlying weak memory model.

There are numerous other formal proof systems that provide mechanisms for structuring the verification of complex concurrent programs and that do not build on separation logic. A common approach is to transform an abstract mathematical description of an algorithm to an efficient implementation using a sequence of refinement steps, each of which is formally justified by establishing a simulation relation between the model and its refinement [3, 7, 8, 26, 101, 104]. Other approaches use abstractions and reductions to combine primitive atomic operations into composite operations that are logically atomic [45, 53, 65, 92, 93]. These approaches can be combined with techniques for reasoning about shared resources that specify frame conditions in classical logic using explicit ghost variables. Such techniques include, e.g., ownership-based techniques [30, 31, 77, 125, 126], dynamic frames [89], and region logic [10]. Compared to separation logic, the explicit han-

dling of frame reasoning incurs some overhead in terms of the complexity of specifications. Nevertheless, reasoning about these specifications can also be automated more directly using classical first-order theorem provers. Approaches such as implicit dynamic frames [105, 155] and linear maps [102] aim to combine the best of both worlds (concise specifications and ease of automation). The verification methodology presented in this dissertation is not inherently tied to separation logic. A formal comparison between implicit dynamic frames and separation logic is given in [137].

10.2 Template algorithms

Our work builds on the concurrent template algorithms for single-copy search structures of Shasha and Goodman [153], extends it to multicopy and lock-free single-copy structures, and develops a formal verification framework to verify such algorithms using state-of-the-art verification technology.

Several other works present generic proof arguments for verifying concurrent traversals [44, 46, 47, 133]. These focus on lock-free search structures that have dynamic linearization points. However, they do not aim to decouple the reasoning about the thread synchronization mechanism from that of the underlying memory representation and data structure invariant. These works also do not provide foundational correctness guarantees.

Meyer and Wolff [120, 121] propose a technique that decouples the proof of data structure correctness from that of the underlying memory reclamation algorithm, allowing the correctness proof to be carried out under the assumption of garbage collection. The verified data structure implementations can then be composed with standard reclamation algorithms, e.g., based on epochs [52] or hazard pointers [123]. It is a promising direction of future work to integrate these approaches and our technique in order to obtain verified data structures where the user can mix-and-match the synchronization technique, memory layout, and the memory reclamation algorithm.

10.3 Hindsight Reasoning

Our work builds on the idea of template algorithms for lock-based concurrent search structures of [97, 98, 138], which we extend to the setting of lock-free implementations. A common challenge when verifying linearizability of lock-free data structures is the prevalence of future-dependent and external linearization points. Hindsight theory [46, 47, 108, 117, 118, 133] has emerged as a suitable technique to address this challenge in the context of concurrent search structures. To our knowledge, we are the first to formalize hindsight reasoning within a foundational program logic. Tools like Poling [174], plankton [117, 118], and nekton [116] automate hindsight reasoning at the expense of an increased trusted code base. However, these tools currently cannot handle complex data structures with unbounded outdegree like skiplists. Also, they do not aim to characterize the design space of related concurrent data structures like our template algorithms do.

Other techniques for dealing with future-dependent linearization points include arguments based on forward simulation (e.g., by tracking all possible linearizations of ongoing operations [78], tracking a partial order [90], or using commit points [17]) and backward simulation (e.g., using prophecy variables [1, 85, 111]). The idea of tracking auxiliary ghost state about a data structure's history to simplify its linearizability proof has been used in many prior works (e.g. [17, 36, 150]).

Our encoding of hindsight reasoning in Iris combines forward reasoning (by tracking the history of the data structure state) and backward reasoning (by using prophecies). However, the details of this encoding are for the most part hidden from the proof engineer by providing a higher-level reasoning interface based on past predicates and temporal interpolation as proposed in [118]. Our comparison with a prior proof of multicopy structure templates [138] suggests that this abstraction helps to reduce the proof complexity.

A proof that uses only history-based verification and does not rely on atomic triples is likely possible. For instance, one alternative approaches to using atomic triples is to prove that the template-level atomic specification contextually refines the client-level atomic specification of multicopy structures using a relational program logic. A number of prior works have developed such refinement-based approaches [9, 54, 55], including for settings that involve unbounded helping [111, 162]. An alternative approach to using prophecy variables for reasoning about non-fixed linearization points is to explicitly construct a partial order of events as the program executes, effectively representing all the possible linearizations that are consistent with the observations made so far [90].

10.4 Skiplists

Several works propose techniques for automatically verifying concurrent skiplists. Abdulla et al. [2] propose a technique for verifying linearizability of lock-free list-based data structures using forest automata. The evaluation considers bounded skiplists with up to 3 levels. However, the implementation does not scale to larger bounds and the unbounded case is outside the scope of the technique. We note that the height of the skiplist is tied to the expected runtime of the skiplist operations. To guarantee the expected worst-case runtime bounds, the skiplist's height must be of order $O(\log(n))$ where n is the expected maximal number of entries in the list. For this reason, real-world skiplist implementations are also parametric in the height. Heights up to 63 levels are feasible in deployed skiplists [Meta], so the restriction to height of 3 in [2] is unrealistic. By contrast, our proofs cover skiplists of arbitrary height.

Sánchez and Sánchez [147] present an SMT-based approach towards an automated verification of concurrent lock-based skiplists. The approach is based on a decidable theory of unbounded skiplists. However, it does not consider lock-free implementations and focuses on establishing *shape invariants* preserved by the structure instead of proving linearizability.

Unlike these automated tools, our approach does not rely on data-structure specific decidable theories for reasoning about inductive properties of heap graphs. Instead, we build on the Flow Framework [99, 100, 119], which enables local reasoning about such properties over

general graphs in separation logic. As a minor contribution, we extend the mechanization of the Flow Framework from [98] with lemmas to reason about graph updates that affect properties of an unbounded number of nodes.

There are some skiplist algorithms that are not immediately covered by our template algorithm. For example, skiplists based on the algorithm presented in [51] such as Java's `ConcurrentSkipListMap` [135] use *backlinks* to avoid restarts when a traversal fails. However, we believe that our template algorithm can be extended to subsume such algorithms by abstracting from the restart policy, similarly to how the present template abstracts from the maintenance policy.

There are several works formalizing results about the run-time complexity of skiplists. Haslbeck and Eberl [64] have verified the commonly known logarithmic runtime for search under the sequential setting. The result ties the expected path length of a search to the height of the skiplist. This work does not consider any concrete algorithm for a skiplist, but instead defines an abstract search operation whose steps simulate the skiplist search operation. Tassarotti and Harper [159] perform complexity analysis of skiplists using Polaris, an extension of Iris for concurrent randomized programs. The skiplist algorithm under consideration follows the (concurrent) lock-based skiplist algorithm from [67]. However, the skiplists are restricted to height of 2 here and the obtained upper bound for the search operation is linear. The work in [6], conceptually in line with [159], analyzes probablistic programs via a program logic. They re-establish the logarithmic runtime for the search operation for a concrete skiplist algorithm of arbitrary height. However, they only consider sequential skiplists.

10.5 Multicopy Structures

Most closely related to our work is the edgeset framework for verifying single-copy structure templates [97, 153]. The edgeset framework hinges on the notion of the *keyset* of a node, which is the set of keys that are allowed in the node. That is, a node's contents must be a subset of its keyset. Moreover, the keysets of all nodes must be disjoint. The contribution of Krishna et al. [97] is to show how keysets can be related to the search structure graph using flows to enable local reasoning about template algorithms for single-copy structures. Note that this work [97, 153] is limited to single-copy structures since the keyset invariants enforce that every key appears in at most one node. In multicopy structures, the same key may appear in multiple nodes with different associated values.

Relative to [97, 153], the main technical novelties are: (i) we identify a node-local quantity (contents-in-reach) for multicopy structures that plays a similar role to the keyset in the single-copy case. Both the invariants that the contents-in-reach must satisfy as well as how the contents-in-reach is encoded using flows is substantially different from the keyset, (ii) hindsight-based linearizability proof for multicopy structures, and (iii) we develop and verify new template algorithms for multicopy structures.

In data structures based on RCU synchronization such as the Citrus tree [5], the same key may temporarily appear in multiple nodes. However, such structures are not necessarily

multicopy structures. Notably, in a Citrus tree, all copies of a key have the same associated value even in the presence of concurrent updates. Moreover, searches have fixed linearization points. This structure can therefore be handled, in principle, using the single-copy framework of Krishna et al. [97] (by building on the formalization of the RCU semantics developed in [59] and the high-level proof idea for the Citrus tree of Feldman et al. [47]).

Multicopy structures such as the LSM tree are often used in file and database systems to organize data that spans multiple storage media, e.g., RAM and hard disks. Several prior projects have considered the formal verification of file systems. SibyllFS [145] provides formal specifications for POSIX-based file system implementations to enable systematic testing of existing implementations. FSCQ [28], Yggdrasil [16, 154], and DFSCQ [27] provide formally verified file system implementations that also guarantee crash consistency. However, these implementations do not support concurrent execution of file system operations.

In regard to concurrent file system implementations, Perennial [23, 24] is a program logic built as an extension of Iris for reasoning about concurrent and crash-safe storage systems. Perennial has been used to verify a Network File System [25] and a journaling system [24]. These systems were verified to be crash-safe and functionally correct.

Our work provides a framework for reasoning about the in-memory concurrency aspects of multicopy structures. However, we mostly abstract from issues related to the interaction with the different storage media. Notably, in our verified LSM tree implementation, we do not model disk failure and hence do not address crash consistency.

Distributed key/value stores have to contend with copies of keys being present in multiple nodes at a time. Several works verify consistency of operations performed on such data structures [29, 88, 170], including linearizability [167]. Lock-free multicopy structures require the development of new template algorithms, which then need to be shown linearizable with respect to the hindsight specification. However, once this is established, linearizability with respect to the client-level specification is obtained for free. We also believe that the high-level invariants from Chapter 7 are applicable towards proving the template-level specification. For instance, each lock-free node-local list of the Bw-tree behaves like a multicopy structure and satisfies the identified invariants.

JellyFish Skiplist [172] is an intersting multicopy structure that borrows heavily from the design of skiplists. The search structure is essentially an insert-only skiplist, where each node stores (time-stamped) values corresponding to the key of the node. Carrott [21] verified a JellyFish implementation with an underlying lock-based skiplist algorithm using Iris. Carrott [21] does not prove the logically atomic specification for the `Map` ADT (like in Chapter 7), but instead prove a weaker Hoare triple akin to the `Set` ADT specification over (*key*, *value*, *timestamp*) triples. We believe that the techniques presented in this dissertation are sufficient to prove the logically atomic specification for the JellyFish search structure.

11 | Conclusion

This dissertation began with the aim of verifying at scale real-world concurrent data structures that were beyond the state of the art. To achieve this, the dissertation introduced novel techniques that aid proof modularity. In particular, the template algorithms in Chapter 8 and Chapter 7 bring algorithmic modularity to a broad class of lock-free single-copy and lock-based multicopy structures. The hindsight framework from Chapter 6 decomposes the proof argument for the novel template algorithms to increase proof reuse. The hindsight framework is applicable beyond search structures and relates prior verification techniques of hindsight reasoning and prophecy variables. Chapter 5 introduced a novel Iris ghost state that has proven to be indispensable for correctness reasoning of single-copy search structures. All of the proofs and techniques above are mechanized in the foundational program logic Iris (built on Coq). This provides the highest form of assurance towards the soundness of the work presented in this dissertation.

11.1 Future Work

For future work, we consider the following avenues:

- **Increased Proof Automation:** The Iris proofs can be automated to a certain degree by assimilating a tool like Diaframe [124] into the methodology. Our invariants and specification are of similar shape across proofs, resulting repeated patterns in the proof. These patterns can be the prime target for automation.
- **Generalization to a broader class of search structures:** The proof methodology of this dissertation can be extended to lock-free multicopy structures [109], and mixed lock and lock-free [43, 66, 114].
- **Generlized Helping Protocol:** The helping protocol can be generalized to include data structures that exhibit modifying future-dependent linearization points such as RDCSS [63] and Herlihy-Wing Queue [70].
- **Proving Liveness:** So far we have not considered liveness of the template algorithms. This would require strengthening the invariants presented in this dissertation, as well as developing new proof techniques.

Bibliography

[1] Abadi, M. and Lamport, L. (1991). The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284.

[2] Abdulla, P. A., Holík, L., Jonsson, B., Lengál, O., Trinh, C. Q., and Vojnar, T. (2013). Verification of heap manipulating programs with ordered data by extended forest automata. In *ATVA*, volume 8172 of *Lecture Notes in Computer Science*, pages 224–239. Springer.

[3] Abrial, J. and Hallerstede, S. (2007). Refinement, decomposition, and instantiation of discrete models: Application to event-b. *Fundam. Informaticae*, 77(1-2):1–28.

[4] Apache Software Foundation (2021). Apache cassandra. https://cassandra.apache.org/. Last accessed on Aug 29, 2024.

[5] Arbel, M. and Attiya, H. (2014). Concurrent updates with RCU: search tree as an example. In *PODC*, pages 196–205. ACM.

[6] Avanzini, M., Barthe, G., Grégoire, B., Moser, G., and Vanoni, G. (2024). Hopping proofs of expectation-based properties: Applications to skiplists and security proofs. *Proc. ACM Program. Lang.*, 8(OOPSLA1):784–809.

[7] Back, R. (1981). On correct refinement of programs. *J. Comput. Syst. Sci.*, 23(1):49–68.

[8] Back, R. and Sere, K. (1989). Stepwise refinement of parallel algorithms. *Sci. Comput. Program.*, 13(1):133–180.

[9] Banerjee, A., Naumann, D. A., and Nikouei, M. (2016). Relational logic with framing and hypotheses. In *FSTTCS*, volume 65 of *LIPIcs*, pages 11:1–11:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[10] Banerjee, A., Naumann, D. A., and Rosenberg, S. (2013). Local reasoning for global invariants, part I: region logic. *J. ACM*, 60(3):18:1–18:56.

[11] Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

[12] Bhargavan, K., Bond, B., Delignat-Lavaud, A., Fournet, C., Hawblitzel, C., Hritcu, C., Ishtiaq, S., Kohlweiss, M., Leino, K. R. M., Lorch, J. R., Maillard, K., Pan, J., Parno, B., Protzenko, J., Ramananandro, T., Rane, A., Rastogi, A., Swamy, N., Thompson, L., Wang, P., Béguelin, S. Z., and Zinzindohoue, J. K. (2017). Everest: Towards a verified, drop-in replacement of HTTPS. In *SNAPL*, volume 71 of *LIPIcs*, pages 1:1–1:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[13] Bizjak, A., Gratzer, D., Krebbers, R., and Birkedal, L. (2019). Iron: managing obligations in higher-order concurrent separation logic. *Proc. ACM Program. Lang.*, 3(POPL):65:1–65:30.

[14] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. (2003). A static analyzer for large safety-critical software. In *PLDI*, pages 196–207. ACM.

[15] Bornat, R., Calcagno, C., and Yang, H. (2005). Variables as resource in separation logic. In *MFPS*, volume 155 of *Electronic Notes in Theoretical Computer Science*, pages 247–276. Elsevier.

[16] Bornholt, J., Kaufmann, A., Li, J., Krishnamurthy, A., Torlak, E., and Wang, X. (2016). Specifying and checking file system crash-consistency models. In *ASPLOS*, pages 83–98. ACM.

[17] Bouajjani, A., Emmi, M., Enea, C., and Mutluergil, S. O. (2017). Proving linearizability using forward simulations. In *CAV (2)*, volume 10427 of *Lecture Notes in Computer Science*, pages 542–563. Springer.

[18] Brookes, S. (2007). A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270.

[19] Brookes, S. and O'Hearn, P. W. (2016). Concurrent separation logic. *ACM SIGLOG News*, 3(3):47–65.

[20] Brookes, S. D. (2004). A semantics for concurrent separation logic. In *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 16–34. Springer.

[21] Carrott, P. L. R. (2022). Formal specification and verification of the lazy jellyfish skip list. Master's Thesis.

[22] Chajed, T., Jung, R., and Tassarotti, J. (2021a). Iris Tutorial. https://gitlab.mpi-sws.org/iris/tutorial-popl21. Last accessed on Aug 29, 2024.

[23] Chajed, T., Tassarotti, J., Kaashoek, M. F., and Zeldovich, N. (2019). Verifying concurrent, crash-safe systems with perennial. In *SOSP*, pages 243–258. ACM.

[24] Chajed, T., Tassarotti, J., Theng, M., Jung, R., Kaashoek, M. F., and Zeldovich, N. (2021b). Gojournal: a verified, concurrent, crash-safe journaling system. In *OSDI*, pages 423–439. USENIX Association.

[25] Chajed, T., Tassarotti, J., Theng, M., Kaashoek, M. F., and Zeldovich, N. (2022). Verifying the daisynfs concurrent and crash-safe file system with sequential reasoning. In *OSDI*, pages 447–463. USENIX Association.

[26] Chandy, K. M. and Misra, J. (1986). An example of stepwise refinement of distributed programs: Quiescence detection. *ACM Trans. Program. Lang. Syst.*, 8(3):326–343.

[27] Chen, H., Chajed, T., Konradi, A., Wang, S., Ileri, A. M., Chlipala, A., Kaashoek, M. F., and Zeldovich, N. (2017). Verifying a high-performance crash-safe file system using a tree specification. In *SOSP*, pages 270–286. ACM.

[28] Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M. F., and Zeldovich, N. (2015). Using crash hoare logic for certifying the FSCQ file system. In *SOSP*, pages 18–37. ACM.

[29] Chordia, S., Rajamani, S. K., Rajan, K., Ramalingam, G., and Vaswani, K. (2013). Asynchronous resilient linearizability. In *DISC*, volume 8205 of *Lecture Notes in Computer Science*, pages 164–178. Springer.

[30] Clarke, D. G., Potter, J., and Noble, J. (1998). Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64. ACM.

[31] Cohen, E., Moskal, M., Schulte, W., and Tobies, S. (2010). Local verification of global invariants in concurrent programs. In *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 480–494. Springer.

[32] Cousot, P. and Cousot, R. (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM.

[33] da Rocha Pinto, P., Dinsdale-Young, T., and Gardner, P. (2014). Tada: A logic for time and data abstraction. In *ECOOP*, volume 8586 of *Lecture Notes in Computer Science*, pages 207–231. Springer.

[34] Dang, H., Jung, J., Choi, J., Nguyen, D., Mansky, W., Kang, J., and Dreyer, D. (2022). Compass: strong and compositional library specifications in relaxed memory separation logic. In *PLDI*, pages 792–808. ACM.

[35] Dayan, N. and Idreos, S. (2018). Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *SIGMOD Conference*, pages 505–520. ACM.

[36] Delbianco, G. A., Sergey, I., Nanevski, A., and Banerjee, A. (2017). Concurrent data structures linked in time. In *ECOOP*, volume 74 of *LIPIcs*, pages 8:1–8:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[37] Dijkstra, E. W. (1968). A constructive approach to the problem of program correctness. *BIT*, 8:174–186.

[38] Dijkstra, E. W. (1975). Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457.

[39] Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M. J., and Yang, H. (2013). Views: compositional reasoning for concurrent programs. In *POPL*, pages 287–300. ACM.

[40] Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M. J., and Vafeiadis, V. (2010). Concurrent abstract predicates. In *ECOOP*, volume 6183 of *Lecture Notes in Computer Science*, pages 504–528. Springer.

[41] Distefano, D., Fähndrich, M., Logozzo, F., and O'Hearn, P. W. (2019). Scaling static analyses at facebook. *Commun. ACM*, 62(8):62–70.

[42] Dodds, M., Jagannathan, S., Parkinson, M. J., Svendsen, K., and Birkedal, L. (2016). Verifying custom synchronization constructs using higher-order separation logic. *ACM Trans. Program. Lang. Syst.*, 38(2):4:1–4:72.

[43] Drachsler, D., Vechev, M. T., and Yahav, E. (2014). Practical concurrent binary search trees via logical ordering. In *PPoPP*, pages 343–356. ACM.

[44] Drachsler-Cohen, D., Vechev, M. T., and Yahav, E. (2018). Practical concurrent traversals in search trees. In *PPoPP*, pages 207–218. ACM.

[45] Elmas, T., Qadeer, S., Sezgin, A., Subasi, O., and Tasiran, S. (2010). Simplifying linearizability proofs with reduction and abstraction. In *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 296–311. Springer.

[46] Feldman, Y. M. Y., Enea, C., Morrison, A., Rinetzky, N., and Shoham, S. (2018). Order out of chaos: Proving linearizability using local views. In *DISC*, volume 121 of *LIPIcs*, pages 23:1–23:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[47] Feldman, Y. M. Y., Khyzha, A., Enea, C., Morrison, A., Nanevski, A., Rinetzky, N., and Shoham, S. (2020). Proving highly-concurrent traversals correct. *Proc. ACM Program. Lang.*, 4(OOPSLA):128:1–128:29.

[48] Feng, X., Ferreira, R., and Shao, Z. (2007). On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 173–188. Springer.

[49] Filipovic, I., O'Hearn, P. W., Rinetzky, N., and Yang, H. (2009). Abstraction for concurrent objects. In *ESOP*, volume 5502 of *Lecture Notes in Computer Science*, pages 252–266. Springer.

[50] Floyd, R. W. (1967). Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32.

[51] Fomitchev, M. and Ruppert, E. (2004). Lock-free linked lists and skip lists. In *PODC*, pages 50–59. ACM.

[52] Fraser, K. (2004). *Practical lock-freedom.* PhD thesis, University of Cambridge, UK.

[53] Freund, S. N. and Qadeer, S. (2004). Checking concise specifications for multithreaded software. *J. Object Technol.*, 3(6):81–101.

[54] Frumin, D., Krebbers, R., and Birkedal, L. (2018). Reloc: A mechanised relational logic for fine-grained concurrency. In *LICS*, pages 442–451. ACM.

[55] Frumin, D., Krebbers, R., and Birkedal, L. (2020). Reloc reloaded: A mechanized relational logic for fine-grained concurrency and logical atomicity. *CoRR*, abs/2006.13635.

[56] Fu, M., Li, Y., Feng, X., Shao, Z., and Zhang, Y. (2010). Reasoning about optimistic concurrency using a program logic for history. In *CONCUR*, volume 6269 of *Lecture Notes in Computer Science*, pages 388–402. Springer.

[57] Google (2024). Error prone. https://errorprone.info/.

[58] Google (2024). LevelDB. https://github.com/google/leveldb. Last accessed on August 29, 2024.

[59] Gotsman, A., Rinetzky, N., and Yang, H. (2013). Verifying concurrent memory reclamation algorithms with grace. In *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 249–269. Springer.

[60] Gu, R., Shao, Z., Kim, J., Wu, X. N., Koenig, J., Sjöberg, V., Chen, H., Costanzo, D., and Ramananandro, T. (2018). Certified concurrent abstraction layers. In *PLDI*, pages 646–661. ACM.

[61] Hammond, A., Liu, Z., Pérami, T., Sewell, P., Birkedal, L., and Pichon-Pharabod, J. (2024). An axiomatic basis for computer programming on the relaxed arm-a architecture: The axsl logic. *Proc. ACM Program. Lang.*, 8(POPL):604–637.

[62] Harris, T. L. (2001). A pragmatic implementation of non-blocking linked-lists. In *DISC*, volume 2180 of *Lecture Notes in Computer Science*, pages 300–314. Springer.

[63] Harris, T. L., Fraser, K., and Pratt, I. A. (2002). A practical multi-word compare-and-swap operation. In *DISC*, volume 2508 of *Lecture Notes in Computer Science*, pages 265–279. Springer.

[64] Haslbeck, M. W. and Eberl, M. (2020). Skip lists. *Arch. Formal Proofs*, 2020.

[65] Hawblitzel, C., Petrank, E., Qadeer, S., and Tasiran, S. (2015). Automated and modular refinement reasoning for concurrent programs. In *CAV (2)*, volume 9207 of *Lecture Notes in Computer Science*, pages 449–465. Springer.

[66] Heller, S., Herlihy, M., Luchangco, V., Moir, M., III, W. N. S., and Shavit, N. (2005). A lazy concurrent list-based set algorithm. In *OPODIS*, volume 3974 of *Lecture Notes in Computer Science*, pages 3–16. Springer.

[67] Herlihy, M., Lev, Y., Luchangco, V., and Shavit, N. (2006). A provably correct scalable concurrent skip list. In *Conference On Principles of Distributed Systems (OPODIS). Citeseer*, volume 103.

[68] Herlihy, M. and Shavit, N. (2008). *The art of multiprocessor programming*. Morgan Kaufmann.

[69] Herlihy, M. and Tygar, J. D. (1987). How to make replicated data secure. In *CRYPTO*, volume 293 of *Lecture Notes in Computer Science*, pages 379–391. Springer.

[70] Herlihy, M. and Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492.

[71] Heule, S., Leino, K. R. M., Müller, P., and Summers, A. J. (2013). Abstract read permissions: Fractional permissions without the fractions. In *VMCAI*, volume 7737 of *Lecture Notes in Computer Science*, pages 315–334. Springer.

[72] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

[73] Hoare, C. A. R. (1972). Proof of correctness of data representations. *Acta Informatica*, 1:271–281.

[74] Iris Team (2021). The Iris 3.4 documentation. https://plv.mpi-sws.org/iris/appendix-3.4.pdf. Last accessed on Aug 29, 2024.

[75] Ishtiaq, S. S. and O'Hearn, P. W. (2001). BI as an assertion language for mutable data structures. In *POPL*, pages 14–26. ACM.

[76] Jacobs, B. and Piessens, F. (2011). Expressive modular fine-grained concurrency specification. In *POPL*, pages 271–282. ACM.

[77] Jacobs, B., Piessens, F., Leino, K. R. M., and Schulte, W. (2005). Safe concurrency for aggregate objects with invariants. In *SEFM*, pages 137–147. IEEE Computer Society.

[78] Jayanti, P., Jayanti, S., Yavuz, U., and Hernandez, L. (2024). A universal, sound, and complete forward reasoning technique for machine-verified proofs of linearizability. *PACMPL*, 8(POPL).

[79] Jonathan Ellis (2011). Leveled compaction in Apache Cassandra. https://www.datastax.com/blog/2011/10/leveled-compaction-apache-cassandra. Last accessed on Aug 29, 2024.

[80] Jones, C. B. (1983). Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332. North-Holland/IFIP.

[81] Jung, J., Lee, J., Choi, J., Kim, J., Park, S., and Kang, J. (2023). Modular verification of safe memory reclamation in concurrent separation logic. *Proc. ACM Program. Lang.*, 7(OOPSLA2):828–856.

[82] Jung, R., Krebbers, R., Birkedal, L., and Dreyer, D. (2016). Higher-order ghost state. In *ICFP*, pages 256–269. ACM.

[83] Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., and Dreyer, D. (2018a). Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20.

[84] Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., and Dreyer, D. (2018b). Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20.

[85] Jung, R., Lepigre, R., Parthasarathy, G., Rapoport, M., Timany, A., Dreyer, D., and Jacobs, B. (2020). The future is ours: prophecy variables in separation logic. *Proc. ACM Program. Lang.*, 4(POPL):45:1–45:32.

[86] Jung, R., Swasey, D., Sieczkowski, F., Svendsen, K., Turon, A., Birkedal, L., and Dreyer, D. (2015). Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*, pages 637–650. ACM.

[87] Kaiser, J., Dang, H., Dreyer, D., Lahav, O., and Vafeiadis, V. (2017). Strong logic for weak memory: Reasoning about release-acquire consistency in iris (artifact). *Dagstuhl Artifacts Ser.*, 3(2):15:1–15:2.

[88] Kaki, G., Nagar, K., Najafzadeh, M., and Jagannathan, S. (2018). Alone together: compositional reasoning and inference for weak isolation. *Proc. ACM Program. Lang.*, 2(POPL):27:1–27:34.

[89] Kassios, I. T. (2006). Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 268–283. Springer.

[90] Khyzha, A., Dodds, M., Gotsman, A., and Parkinson, M. J. (2017). Proving linearizability using partial orders. In *ESOP*, volume 10201 of *Lecture Notes in Computer Science*, pages 639–667. Springer.

[91] Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D. A., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. (2010). sel4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115.

[92] Kragl, B. and Qadeer, S. (2018). Layered concurrent programs. In *CAV (1)*, volume 10981 of *Lecture Notes in Computer Science*, pages 79–102. Springer.

[93] Kragl, B., Qadeer, S., and Henzinger, T. A. (2020). Refinement for structured concurrent programs. In *CAV (1)*, volume 12224 of *Lecture Notes in Computer Science*, pages 275–298. Springer.

[94] Krebbers, R., Jourdan, J., Jung, R., Tassarotti, J., Kaiser, J., Timany, A., Charguéraud, A., and Dreyer, D. (2018). Mosel: a general, extensible modal framework for interactive proofs in separation logic. *Proc. ACM Program. Lang.*, 2(ICFP):77:1–77:30.

[95] Krebbers, R., Jung, R., Bizjak, A., Jourdan, J., Dreyer, D., and Birkedal, L. (2017a). The essence of higher-order concurrent separation logic. In *ESOP*, volume 10201 of *Lecture Notes in Computer Science*, pages 696–723. Springer.

[96] Krebbers, R., Timany, A., and Birkedal, L. (2017b). Interactive proofs in higher-order concurrent separation logic. In *POPL*, pages 205–217. ACM.

[97] Krishna, S., Patel, N., Shasha, D. E., and Wies, T. (2020a). Verifying concurrent search structure templates. In *PLDI*, pages 181–196. ACM.

[98] Krishna, S., Patel, N., Shasha, D. E., and Wies, T. (2021). *Automated Verification of Concurrent Search Structures*. Synthesis Lectures on Computer Science. Morgan & Claypool Publishers.

[99] Krishna, S., Shasha, D. E., and Wies, T. (2018). Go with the flow: compositional abstractions for concurrent data structures. *Proc. ACM Program. Lang.*, 2(POPL):37:1–37:31.

[100] Krishna, S., Summers, A. J., and Wies, T. (2020b). Local reasoning for global graph properties. In *ESOP*, volume 12075 of *Lecture Notes in Computer Science*, pages 308–335. Springer.

[101] Kuppe, M. A., Lamport, L., and Ricketts, D. (2019). The TLA+ toolbox. In *F-IDE@FM*, volume 310 of *EPTCS*, pages 50–62.

[102] Lahiri, S. K., Qadeer, S., and Walker, D. (2011). Linear maps. In *PLPV*, pages 3–14. ACM.

[103] Lamport, L. (1977). Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143.

[104] Lamport, L. (1994). The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923.

[105] Leino, K. R. M. and Müller, P. (2009). A basis for verifying multi-threaded programs. In *ESOP*, volume 5502 of *Lecture Notes in Computer Science*, pages 378–393. Springer.

[106] Leino, K. R. M., Müller, P., and Smans, J. (2009). Verification of concurrent programs with chalice. In *FOSAD*, volume 5705 of *Lecture Notes in Computer Science*, pages 195–222. Springer.

[107] Leroy, X. (2006). Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*, pages 42–54. ACM.

[108] Lev-Ari, K., Chockler, G. V., and Keidar, I. (2015). A constructive approach for proving data structures' linearizability. In *DISC*, volume 9363 of *Lecture Notes in Computer Science*, pages 356–370. Springer.

[109] Levandoski, J. J., Lomet, D. B., and Sengupta, S. (2013). The bw-tree: A b-tree for new hardware platforms. In *ICDE*, pages 302–313. IEEE Computer Society.

[110] Leveson, N. and Turner, C. (1993). An investigation of the therac-25 accidents. *Computer*, 26(7):18–41.

[111] Liang, H. and Feng, X. (2013). Modular verification of linearizability with non-fixed linearization points. In *PLDI*, pages 459–470. ACM.

[112] Luo, C. and Carey, M. J. (2020). Lsm-based storage techniques: a survey. *VLDB J.*, 29(1):393–418.

[113] Manna, Z. and Pnueli, A. (1995). *Temporal verification of reactive systems - safety*. Springer.

[114] Mao, Y., Kohler, E., and Morris, R. T. (2012). Cache craftiness for fast multicore key-value storage. In *EuroSys*, pages 183–196. ACM.

[Meta] Meta. Facebook Open Source Library: ConcurrentSkipList. https://github.com/facebook/folly/blob/main/folly/ConcurrentSkipList.h. Last accessed on Aug 29, 2024.

[116] Meyer, R., Opaterny, A., Wies, T., and Wolff, S. (2023a). nekton: A linearizability proof checker. In *CAV (1)*, volume 13964 of *Lecture Notes in Computer Science*, pages 170–183. Springer.

[117] Meyer, R., Wies, T., and Wolff, S. (2022). A concurrent program logic with a future and history. *Proc. ACM Program. Lang.*, 6(OOPSLA2):1378–1407.

[118] Meyer, R., Wies, T., and Wolff, S. (2023b). Embedding hindsight reasoning in separation logic. *Proc. ACM Program. Lang.*, 7(PLDI):1848–1871.

[119] Meyer, R., Wies, T., and Wolff, S. (2023c). Make flows small again: Revisiting the flow framework. In *TACAS (1)*, volume 13993 of *Lecture Notes in Computer Science*, pages 628–646. Springer.

[120] Meyer, R. and Wolff, S. (2019). Decoupling lock-free data structures from memory reclamation for static analysis. *Proc. ACM Program. Lang.*, 3(POPL):58:1–58:31.

[121] Meyer, R. and Wolff, S. (2020). Pointer life cycle types for lock-free data structures with memory reclamation. *Proc. ACM Program. Lang.*, 4(POPL):68:1–68:36.

[122] Michael, M. M. (2002). High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, pages 73–82. ACM.

[123] Michael, M. M. (2004). Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distributed Syst.*, 15(6):491–504.

[124] Mulder, I., Krebbers, R., and Geuvers, H. (2022). Diaframe: automated verification of fine-grained concurrent programs in iris. In *PLDI*, pages 809–824. ACM.

[125] Müller, P. (2001). *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen.

[126] Müller, P., Poetzsch-Heffter, A., and Leavens, G. T. (2006). Modular invariants for layered object structures. *Sci. Comput. Program.*, 62(3):253–286.

[127] Nanevski, A., Ley-Wild, R., Sergey, I., and Delbianco, G. A. (2014). Communicating state transition systems for fine-grained concurrent resources. In *ESOP*, volume 8410 of *Lecture Notes in Computer Science*, pages 290–310. Springer.

[128] NASA (December 2007). System Failure Case Studies. https://sma.nasa.gov/docs/default-source/safety-messages/safetymessage-2008-03-01-northeastblackoutof2003.pdf. Last accessed on Aug 29, 2024.

[129] O'Hearn, P. W. (2004). Resources, concurrency and local reasoning. In *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 49–67. Springer.

[130] O'Hearn, P. W. (2007). Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307.

[131] O'Hearn, P. W. (2019). Separation logic. *Commun. ACM*, 62(2):86–95.

[132] O'Hearn, P. W., Reynolds, J. C., and Yang, H. (2001). Local reasoning about programs that alter data structures. In *CSL*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer.

[133] O'Hearn, P. W., Rinetzky, N., Vechev, M. T., Yahav, E., and Yorsh, G. (2010). Verifying linearizability with hindsight. In *PODC*, pages 85–94. ACM.

[134] O'Neil, P. E., Cheng, E., Gawlick, D., and O'Neil, E. J. (1996). The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385.

[135] Oracle (2024). Java concurrent skiplist set. https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/concurrent/ConcurrentSkipListSet.html. Last accessed on Aug 29, 2024.

[136] Owicki, S. S. and Gries, D. (1976). Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285.

[137] Parkinson, M. J. and Summers, A. J. (2012). The relationship between separation logic and implicit dynamic frames. *Log. Methods Comput. Sci.*, 8(3).

[138] Patel, N., Krishna, S., Shasha, D. E., and Wies, T. (2021). Verifying concurrent multicopy search structures. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–32.

[139] Patel, N., Shasha, D. E., and Wies, T. (2024). Verification of lock-free search structure templates. *Proc. ACM Program. Lang.*, (ECOOP). To appear.

[140] Pierce, B. C., de Amorim, A. A., Casinghino, C., Gaboardi, M., Greenberg, M., Hritcu, C., Sjöberg, V., and Yorgey, B. (2020). *Logical Foundations*, volume 1 of *Software Foundations*. Electronic textbook. Version 5.8, http://softwarefoundations.cis.upenn.edu.

[141] Pnueli, A. (1977). The temporal logic of programs. In *FOCS*, pages 46–57. IEEE Computer Society.

[142] Raad, A., Villard, J., and Gardner, P. (2015). Colosl: Concurrent local subjective logic. In *ESOP*, volume 9032 of *Lecture Notes in Computer Science*, pages 710–735. Springer.

[143] Raju, P., Kadekodi, R., Chidambaram, V., and Abraham, I. (2017). Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *SOSP*, pages 497–514. ACM.

[144] Reynolds, J. C. (2002). Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society.

[145] Ridge, T., Sheets, D., Tuerk, T., Giugliano, A., Madhavapeddy, A., and Sewell, P. (2015). Sibylfs: formal specification and oracle-based testing for POSIX and real-world file systems. In *SOSP*, pages 38–53. ACM.

[146] Rungta, N. (2022). A billion smt queries a day. In *CAV 2022*.

[147] Sánchez, A. and Sánchez, C. (2014). Formal verification of skiplists with arbitrary many levels. In *ATVA*, volume 8837 of *Lecture Notes in Computer Science*, pages 314–329. Springer.

[148] Sears, R. and Ramakrishnan, R. (2012). blsm: a general purpose log structured merge tree. In *SIGMOD Conference*, pages 217–228. ACM.

[149] Sergey, I., Nanevski, A., and Banerjee, A. (2015a). Mechanized verification of fine-grained concurrent programs. In *PLDI*, pages 77–87. ACM.

[150] Sergey, I., Nanevski, A., and Banerjee, A. (2015b). Specifying and verifying concurrent algorithms with histories and subjectivity. In *ESOP*, volume 9032 of *Lecture Notes in Computer Science*, pages 333–358. Springer.

[151] Sergey, I., Nanevski, A., Banerjee, A., and Delbianco, G. A. (2016). Hoare-style specifications as correctness conditions for non-linearizable concurrent objects. In *OOPSLA*, pages 92–110. ACM.

[152] Severance, D. G. and Lohman, G. M. (1976). Differential files: Their application to the maintenance of large databases. *ACM Trans. Database Syst.*, 1(3):256–267.

[153] Shasha, D. E. and Goodman, N. (1988). Concurrent search structure algorithms. *ACM Trans. Database Syst.*, 13(1):53–90.

[154] Sigurbjarnarson, H., Bornholt, J., Torlak, E., and Wang, X. (2016). Push-button verification of file systems via crash refinement. In *OSDI*, pages 1–16. USENIX Association.

[155] Smans, J., Jacobs, B., and Piessens, F. (2009). Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP*, volume 5653 of *Lecture Notes in Computer Science*, pages 148–172. Springer.

[156] Source, O. (2024). Clang static analyzer. https://clang-analyzer.llvm.org/.

[157] Svendsen, K. and Birkedal, L. (2014). Impredicative concurrent abstract predicates. In *ESOP*, volume 8410 of *Lecture Notes in Computer Science*, pages 149–168. Springer.

[158] Svendsen, K., Pichon-Pharabod, J., Doko, M., Lahav, O., and Vafeiadis, V. (2018). A separation logic for a promising semantics. In *ESOP*, volume 10801 of *Lecture Notes in Computer Science*, pages 357–384. Springer.

[159] Tassarotti, J. and Harper, R. (2019). A separation logic for concurrent randomized programs. *Proc. ACM Program. Lang.*, 3(POPL):64:1–64:30.

[160] Thonangi, R. and Yang, J. (2017). On log-structured merge for solid-state drives. In *ICDE*, pages 683–694. IEEE Computer Society.

[161] Turing, A. M. (1949). Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69. Univ. Math. Lab., Cambridge.

[162] Turon, A. J., Thamsborg, J., Ahmed, A., Birkedal, L., and Dreyer, D. (2013). Logical relations for fine-grained concurrency. In *POPL*, pages 343–356. ACM.

[163] Vafeiadis, V. (2008). *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, UK.

[164] Vafeiadis, V. and Parkinson, M. J. (2007). A marriage of rely/guarantee and separation logic. In *CONCUR*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer.

[165] VanHattum, A., Schwartz-Narbonne, D., Chong, N., and Sampson, A. (2023). Verifying dynamic trait objects in rust. In *ICSE 2023*.

[166] Vindum, S. F. and Birkedal, L. (2023). Spirea: A mechanized concurrent separation logic for weak persistent memory. *Proc. ACM Program. Lang.*, 7(OOPSLA2):632–657.

[167] Wang, C., Enea, C., Mutluergil, S. O., and Petri, G. (2019). Replication-aware linearizability. In *PLDI*, pages 980–993. ACM.

[168] Winskel, G. (1993). *The formal semantics of programming languages - an introduction*. Foundation of computing series. MIT Press.

[169] Wu, X., Xu, Y., Shao, Z., and Jiang, S. (2015). Lsm-trie: An lsm-tree-based ultra-large key-value store for small data items. In *USENIX ATC*, pages 71–82. USENIX Association.

[170] Xiong, S., Cerone, A., Raad, A., and Gardner, P. (2020). Data consistency in transactional storage systems: A centralised semantics. In *ECOOP*, volume 166 of *LIPIcs*, pages 21:1–21:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[171] Xiong, S., da Rocha Pinto, P., Ntzik, G., and Gardner, P. (2017). Abstract specifications for concurrent maps. In *ESOP*, volume 10201 of *Lecture Notes in Computer Science*, pages 964–990. Springer.

[172] Yeon, J., Kim, L., Han, Y., Lee, H. G., Lee, E., and Kim, B. S. (2020). Jellyfish: A fast skip list with MVCC. In *Middleware*, pages 134–148. ACM.

[173] Zhang, Z., Feng, X., Fu, M., Shao, Z., and Li, Y. (2012). A structural approach to prophecy variables. In *TAMC*, volume 7287 of *Lecture Notes in Computer Science*, pages 61–71. Springer.

[174] Zhu, H., Petri, G., and Jagannathan, S. (2015). Poling: SMT aided linearizability proofs. In *CAV (2)*, volume 9207 of *Lecture Notes in Computer Science*, pages 3–19. Springer.