# 석사학위논문 Master's Thesis

# 동시성 분리 논리를 사용한 Chase-Lev 덱의 엄밀한 검증

Formal Verification of Chase-Lev Deque in Concurrent Separation Logic

2023

최 재 민 (崔 在 珉 Choi, Jaemin)

한국과학기술원

Korea Advanced Institute of Science and Technology

# 석사학위논문

동시성 분리 논리를 사용한 Chase-Lev 덱의 엄밀한 검증

2023

최재민

한 국 과 학 기 술 원 전산학부 **MCS** 

최재민. 동시성 분리 논리를 사용한 Chase-Lev 덱의 엄밀한 검증. 전산학부. 2023년. 43+iv 쪽. 지도교수: 강지훈. (영문 논문)

Jaemin Choi. Formal Verification of Chase-Lev Deque in Concurrent Separation Logic. School of Computing . 2023. 43+iv pages. Advisor: Jeehoon Kang. (Text in English)

#### 초 록

Chase-Lev 덱은 멀티프로세서 스케줄링에서 효율적으로 부하를 분산시키는 데 사용되는 동시성 자료 구조이다. 이는 다음과 같은 작업 훔치기 기법을 지원한다. 각 스레드는 작업 저장소로서 Chase-Lev 덱을 하나씩 소유하고, 할 작업이 없어진 스레드는 다른 스레드의 작업 저장소로부터 작업을 훔쳐 대신 실행한다. 하지만 멀티프로세서 환경을 비롯한 모든 소프트웨어에서 버그의 위험이 내재되어 있기 때문에, 프로그램이나 자료 구조가 올바르게 동작함을 엄밀하게 증명하는 것이 중요하다. 현재까지 알려진 바로는 Chase-Lev 덱의 엄밀한 검증 연구 중 (1) 증명에서 믿고 넘어가야 하는 요소가 가능한 한 작고, (2) 현실적이면서 제약이 없는 구현을 사용하며, (3) 강한 명세를 증명한 사례는 없었다.

본 논문에서는 이러한 한계를 해결하기 위해 동시성 분리 논리를 사용하여 Chase-Lev 덱의 엄밀한 검증을 제시한다. 이 검증은 Coq 증명 보조 도구를 사용하여 작성되었으며, 검증된 구현은 현실적이면서 작업의 수에 제약이 없다. 또한 동시성 자료 구조의 강한 명세로 흔히 인정되는 linearizability를 명세로 사용한다. 따라서 본 연구의 검증은 위의 세 조건을 모두 충족한다. 추가로, 검증 작업을 확장하여 메모리 재활용 기법을 사용하는 구현을 검증하고, 느슨한 메모리 모델에서 Chase-Lev 덱을 검증하기 위한 토대를 마련하였다.

핵 심 낱 말 동시성, 엄밀한 검증, 프로그램 논리, 자료 구조, 분리 논리, Chase-Lev 덱

#### Abstract

Chase-Lev deque is a concurrent data structure designed for efficient load balancing in multiprocessor scheduling. It employs a work-stealing strategy, where each thread possesses its own work-stealing deque to store tasks, and idle threads steal tasks from other threads. However, given the inherent risk of bugs in software, particularly in a multiprocessor environment, it is crucial to formally establish the correctness of programs and data structures. To our knowledge, no formal verification work for the Chase-Lev deque has met three key criteria: (1) utilizing a minimal trusted computing base, (2) using a realistic and unrestricted implementation, and (3) proving a strong specification.

In this thesis, we address this gap by presenting the formal verification of the Chase-Lev deque using a concurrent separation logic. Our work is mechanized in the Coq proof assistant, and our verified implementation is both realistic and unbounded in terms of the number of tasks it can handle. Also, we adopt linearizability as the specification, as it is widely recognized as a strong specification for concurrent data structures. Consequently, our work satisfies all three aforementioned criteria for formal verification. Additionally, we extend our verification to support safe memory reclamation, and provide a basis for verifying the Chase-Lev deque in the relaxed memory model.

**Keywords** Concurrency, formal verification, program logic, data structure, separation logic, Chase-Lev deque

## Contents

Content	S	1			
List of 7	Tables	iii			
List of I	Figures	iv			
Chapter	1. Introduction	1			
1.1	Background: Chase-Lev Deque for Efficient Work-stealing	1			
1.2	Problem: Lack of Formal Verification for Chase-Lev Deque	1			
	1.2.1 Necessity of Formal Verification	1			
	1.2.2 Prior Works on Chase-Lev Deque Verification	2			
	1.2.3 Verification Challenges	3			
1.3	Our Solution: Foundational Verification of Chase-Lev Deque	4			
Chapter	2. Background: Chase-Lev Deque	5			
2.1	The APIs of Work-stealing	5			
2.2	Structure of Chase-Lev Deque	5			
2.3					
2.4	Intuition	8			
Chapter	3. Background: Iris Separation Logic	9			
3.1	Separation Logic	9			
3.2	Specification and Invariant	10			
3.3	Persistent Propositions	10			
3.4	Ghost States	11			
3.5	Linearizability and Logical Atomicity	12			
Chapter	4. Verification of Chase-Lev Deque without Resizing	14			
4.1	Specification with Private Postconditions	14			
4.2	Resource Definitions	15			
4.3	Verification of Each Function	17			
	4.3.1 New	17			
	4.3.2 Push	17			
	4.3.3 Pop	19			
	4.3.4 Steal	20			
4.4	Construction of the Ghost Deque State	22			

Chapter	5.	Full Verification of Chase-Lev Deque	24
5.1	Speci	fication of Resizing	24
5.2	New	Deque State and Invariant	24
5.3	Chan	ges in the Verification	26
	5.3.1	Push	26
	5.3.2	Steal	26
5.4	Const	truction of the Full Deque State	27
Chapter	6.	Verification in Extended Settings	30
6.1	Verifi	cation with Safe Memory Reclamation (SMR)	30
	6.1.1	Background: Memory Reclamation	30
	6.1.2	Specification and Verification in SMR	30
6.2	Found	dation for Verification in Relaxed Memory Model	32
	6.2.1	Background: Relaxed Memory Model and iRC11	32
	6.2.2	Implementation and Safety Verification in iRC11	34
Chapter	7.	Conclusion	36
7.1	Sumn	nary	36
7.2	Relat	ed Work	36
7.3	Futur	re Work	37
Acknowl	edgme	nts in Korean	43

## List of Tables

1.1 Comparison of Chase-Lev deque verification.		3
---	--	---

# List of Figures

1.1	A demonstration of work-stealing for 5 threads	2
2.1	A Chase-Lev deque with the content [10, 11, 3]	6
2.2	The initialization and field access functions	6
2.3	The push function	7
2.4	The pop and steal functions	7
3.1	Proof rules for points-to	9
3.2	Proof rules for ghost variables	11
3.3	Proof rules for monotonic natural numbers	12
4.1	Specification of Chase-Lev deque	15
4.2	Proof rules of deque state	17
4.3	Specification of allocation in HeapLang	17
4.4	Proof rules of ghost map.	22
4.5	The definition of ghost deque state	23
5.1	Additional proof rules of deque state with eras	25
5.2	The definition of full deque state	28
6.1	Changes in the implementation to support hazard pointers	31
6.2	Selected proof rules of hazard pointers	31

## Chapter 1. Introduction

## 1.1 Background: Chase-Lev Deque for Efficient Work-stealing

High throughput is a significant obstacle when it comes to managing multiple processors concurrently. To optimize this objective, efficient load balancing is required. The aim is to allocate tasks evenly, ensuring that certain threads are not overwhelmed while others remain idle. However, due to various constraints such as dependencies and unknown execution times, it is challenging to distribute tasks effectively from the get-go.

Work-stealing [10] tackles this problem by dynamically changing the work distribution. In this approach, when a thread becomes idle, it actively searches for tasks in the task pool of other threads and "steals" a task to execute, ensuring that idle threads remain productive. Each thread maintains its own task storage called a work-stealing deque, containing the tasks assigned to it. When a thread has a work to perform, it removes a task from its deque. Also, when a thread becomes idle but its deque is empty, it steals a task from another thread's deque. This is illustrated in Figure 1.1, where thread 5 is trying to remove task 3 from its own deque, while thread 3 and 4 are trying to steal task 1 from thread 5's deque.

Numerous work-stealing deques have been developed for efficient work-stealing [8, 13, 33, 37, 42, 6, 43, 11, 45]. Among these, the work presented by Chase and Lev [13], known as the *Chase-Lev deque*, stands out as a popular high-performance and realistic design. It achieves fine-grained concurrency by avoiding the use of locks and allows for an unlimited number of elements in a single deque. The Chase-Lev deque is widely used in various real-world concurrency frameworks [2, 5], further demonstrating its practicality and effectiveness.

### 1.2 Problem: Lack of Formal Verification for Chase-Lev Deque

#### 1.2.1 Necessity of Formal Verification

However, the multiprocessor environment presents additional complexities and challenges compared to single-threaded settings, making it more prone to errors. In the context of work-stealing, multiple threads may concurrently attempt to steal the same task from the same deque, leading to collisions. At the same time, the owner thread may also try to insert or remove its tasks, potentially colliding with the stealers. A proper synchronization is required to handle this situation: an incorrect implementation can result in various bugs, such as tasks being executed multiple times, removed in an undesirable order, or not inserted into the deque.

The complexity of the situation further intensifies in modern architectures and compilers. Most of this thesis assumes the sequentially consistent (SC) memory model, where instructions within each thread are executed in order. However, modern architectures follow the relaxed memory model, which allows for out-of-order execution of instructions for optimization purposes. While programmers can impose certain ordering constraints to ensure proper synchronization, it must be done with caution. Overly

<sup>&</sup>lt;sup>1</sup>In this paper, we may refer to work-stealing deques as just deques, although they don't provide all the APIs of a regular deque.

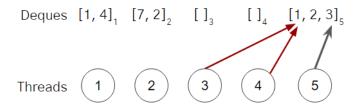


Figure 1.1: A demonstration of work-stealing for 5 threads.

strict ordering can degrade the performance, while overly weak ordering can introduce bugs. It is worth noting that even the implementation of Chase-Lev deque in relaxed-memory, written and peer-reviewed by experts [33], was found to contain a bug [40].

Given the complexity of real-world architecture and programs, how can we ensure the absence of bugs? While program testing can help identify unexpected bugs, it does not guarantee the lack of them. As such, it is not uncommon to see bug reports in heavily tested commercial programs. This gives rise to the significance of *formal verification*, the process of specifying and proving the correctness of programs using formal methods. By providing a rigorous and systematic approach to verifying program correctness, formal verification helps mitigate the risk of bugs and ensures a higher level of reliability in software.

#### 1.2.2 Prior Works on Chase-Lev Deque Verification

To the best of our knowledge, there has not been a foundational, realistic, and strong verification of Chase-Lev deque. Specifically, the following are the properties of interest for formal verification:

- Foundational verification: the verification process should be mechanized in proof assistants like Coq [1], allowing for a minimal trusted computing base. While model checkers offer the advantage of automation, their credibility relies on trusting the theory and implementation of the whole model checking program. On the other hand, proof assistants only require trust in the proof checker and not the underlying logic or framework.
- Realistic implementation: the implementation under verification should not be simplified to the extent of being restrictive or unrealistic. It is crucial to capture the essential complexities and behaviors of the actual Chase-Lev deque to ensure that the verification results hold in practical scenarios.
- Strong specification: the verified specification should be strong enough to allow clients to utilize the data structure in various ways. In the SC memory model, *linearizability* [20] is a de facto standard for the correctness of concurrent data structures. For the relaxed memory model, the Compass framework [16] has recently been developed to provide a strong specification.

Lê et al. [33] implemented Chase-Lev deque in the relaxed memory model and provided a proof of its correctness. However, the proof was conducted using pen-and-paper, which is not sufficient for a foundational verification. Pen-and-paper proofs are prone to human errors and require thorough review compared to machine-checked proofs. Additionally, although the verified specification is strong with regard to the SC memory model, it does not fully capture the relaxed memory behavior such as the synchronization between clients of the deque. Moreover, the lack of modularity in the specification poses

	Machine-	Foundational	Realistic	Strong	Relaxed	Real-world
	checked		impl.	spec	memory	language
Lê et al. [33]	X	×	✓	<b>A</b>	✓	✓
Mutluergil and Tasiran [39]	✓	×	X	✓	Х	Х
Kokologiannakis et al. [30]	✓	×	X	Х	✓	✓
Our work	✓	✓	✓	1	<b>A</b>	X

Table 1.1: Comparison of Chase-Lev deque verification.

challenges when verifying programs that utilize the Chase-Lev deque, as it is not straightforward to incorporate and reason about its specification in a modular manner.

Mutluergil and Tasiran [39] verified the linearizability of the Chase-Lev deque in the SC memory model using the Civl verifier [31]. As such, it is the first machine-checked proof of linearizability for the Chase-Lev deque. However, it is not foundational: Civl is a complex proof system, and its correctness should be trusted for the basis of the verification. Furthermore, the implementation in the verification assumes an infinitely large array, which is unrealistic in practice. It is crucial to note that this is not a minor simplifying assumption, as the synchronization and reasoning required in the full implementation are significantly more complex than with this simplification.

Kokologiannakis et al. [30] developed a model checker for C programs in configurable memory models, and used it to verify the Chase-Lev deque in a relaxed memory setting as a benchmark. While it has an advantage of verifying a C implementation in the relaxed memory model, it does not meet any of the three criteria of interest. First, it relies on the model checker as a trusted computing base. Second, although it uses a finite array, the capacity of the deque is limited [3], introducing the same simplification issue as assuming an infinite array. Finally, linearizability or other strong specifications were not verified, but only some weaker guarantees such as safety.

The comparison of Chase-Lev deque verification approaches is summarized in Table 1.1. In addition to the the criteria of our interest plus machine-checked verification (as a subcriterion of foundational verification), we also compare them with regard to verification in the relaxed memory model, and the use of real-world languages like C.

#### 1.2.3 Verification Challenges

Recent advancements in formal verification have enabled the foundational verification of strong specifications for various concurrent data structures, including those utilized in industrial projects [44, 32], as well as those in relaxed memory [26, 16]. However, the verification of the Chase-Lev deque poses unique challenges. Its complexity lies in the intricacies of synchronization, to the extent that it is nontrivial to comprehend its correctness even intuitively.

The Chase-Lev deque uses a dynamic circular array which automatically resizes on overflow. The contents of the deque are represented by a circular slice of this array. The deque also maintains two integer indices, namely *top* and *bottom*. These indices denote the starting and ending positions of the slice, respectively. The owner of the deque may insert or remove elements (tasks) from the bottom end of the deque, and the stealers may remove elements from the top end.

An obvious source of conflict arises when multiple threads attempt to steal the same element from the top end of the deque. In addition, if only one element remains, the owner thread may also attempt to pop that element, potentially joining the conflict. This situation is handled by a CAS operation on the top index, and the verification can proceed by just doing a case analysis on whether the CAS succeeded.

However, the synchronization involved in the Chase-Lev deque extends beyond simple CAS operations. Verifying its correctness requires intricate reasoning about the array modifications. One source of complication is that the value to steal is determined before CAS-ing. A steal operation involves CAS-ing the top index, but before that, the stealer must read the array to remember the value it intends to steal (the reason will be elaborated in chapter 2). After a successful CAS, the value remembered will be returned. Then why does it work correctly, despite the owner being able to pop everything and push a new element in the meantime, or wrap around and overwrite the slot by pushing? To answer this question, we should be able to establish that certain values are preserved during a specific period of execution.

Another challenge arises from the fact that the owner can replace the array while a stealer is in the process of stealing. Consider a scenario where a stealer reads the address of the array but then gets stalled, and the owner replaces the array while pushing new elements. It is even possible to replace the array multiple times by pushing a significant number of elements. Eventually, the stealer resumes execution but reads a value from the old, replaced array. Surprisingly, despite the array being replaced, the stealer can still successfully complete the operation and affect the deque's state. This is in contrast to other data structures like Harris' linked list [18], where an operation fails and restarts if it detects a detached node. To account for this situation in the Chase-Lev deque, the verification process must reason not only about the current array but also about all past arrays, and establish some form of linkage between them.

### 1.3 Our Solution: Foundational Verification of Chase-Lev Deque

This thesis presents the first full foundational verification of Chase-Lev deque using the Iris separation logic [4, 23]. The verification is mechanized in the Coq proof assistant [1], using the mechanization for Iris. Our verified implementation of the deque is both realistic and unbounded, as it utilizes a finite circular array that dynamically resizes upon overflow. Moreover, we establish the linearizability of the Chase-Lev deque, which provides a strong specification in the SC memory model. Thus, our verification satisfies all the criteria of foundational verification, realistic implementation, and strong specification. Specifically, we make the following contributions:

- In chapter 4, we present the verification of the Chase-Lev deque without considering array resizing.
- In chapter 5, we extend the verification to encompass the full implementation of the Chase-Lev deque, which includes array resizing.
- In chapter 6, we explore the extension of our verification approach to different settings. Specifically, we verify Chase-Lev deque under safe memory reclamation, and provide a basis for verification in the relaxed memory model.

The other chapters are organized as follows. In chapter 2, we describe the implementation of the Chase-Lev deque, and explain the intuition behind its correctness. In chapter 3, we give a brief introduction to the Iris separation logic, focusing on the features relevant to our work. In chapter 7, we summarize our results, and present related and future works. All our results are mechanized in Coq, and the mechanization for chapter 4 and chapter 5 are available in the following link: https://github.com/kaist-cp/chase-lev-verification. The mechanization for chapter 6 will be published in the future along with the corresponding papers.

## Chapter 2. Background: Chase-Lev Deque

### 2.1 The APIs of Work-stealing

Recall from Figure 1.1: in a scheduling scheme that supports work-stealing, each thread maintains its own work-stealing deque containing its assigned tasks. The owner of a deque can push a newly assigned task to its deque, or pop a task from its deque and start executing it. Other threads, which we will call *stealers*, can steal a task from the same deque so that they can execute it instead. This helps balancing out the workload, and preventing some threads to become idle while other threads are overloaded with tasks. Technically, the owner can also steal a task from its own deque, although this breaks the owner's LIFO behavior.

A work-stealing deque provides the following deque-like interface:

- **Push** inserts a task at the bottom end of the deque. This method can only be called by the owner of the deque.
- Pop tries to remove a task from the bottom end of the deque. This method can only be called by the owner of the deque. The attempt may fail if the deque is empty or it clashes with other threads' steals.
- Steal tries to remove a task from the top end of the deque. This method can be called by any thread. The attempt may fail if the deque is empty or it clashes with other threads' steals or the owner's pop.

## 2.2 Structure of Chase-Lev Deque

Now we go over the details of Chase-Lev deque. The structure of the deque is illustrated in Figure 2.1. It is implemented as a circular array, along with top and bottom indices. Being a circular array, the indexing is done modulo the array's size. If the top index, bottom index, and array are t, b, and arr, respectively, then the contents of the deque is represented by the half-open circular slice [t, b), i.e.  $[arr[t \mod N], arr[(t+1) \mod N], \cdots, arr[(b-1) \mod N]]$  where N = |arr|. Pushing a value to the deque amounts to writing the value at the bottom index of the array and incrementing the bottom. Similarly, a successful pop decrements the bottom index and returns the value at the new bottom index (except in a corner case we will soon discuss); a successful steal does the same but uses the top index.

Additionally, the owner resizes the array when it tries to push but the array is full. To do this, the owner allocates a larger array, copies the values from top to bottom in a way such that modulo indexing gives the same value, and substitutes the array with the new one.

Since all array accesses are done modulo its size, we will denote arr[i] as a shorthand for  $arr[i \mod |arr|]$ .

## 2.3 Implementation

The implementation of Chase-Lev deque in HeapLang, a language provided by Iris, is presented in this section.

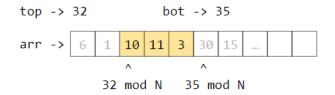


Figure 2.1: A Chase-Lev deque with the content [10, 11, 3].

```
Definition new_wsdeque : val := λ: "sz",
let: "array" := AllocN "sz" #0 in
  (ref ("array", "sz"), ref #1, ref #1). (* array+size, top, bot *)

Definition arr : val := λ: "deque", Fst (Fst "deque").
Definition top : val := λ: "deque", Snd (Fst "deque").
Definition bot : val := λ: "deque", Snd "deque".
Definition access : val := λ: "arr" "i" "n", "arr" + ("i" 'rem' "n").
```

Figure 2.2: The initialization and field access functions.

The initialization and field access functions are given in Figure 2.2. A deque is represented as a tuple of circular array, top index, and bottom index. The top and bottom indices start from 1 instead of 0 just for a technical reason. Also, the size is stored along with the array itself so that threads can index into it modulo its size. arr, top, and bot are used as shorthands for field accesses. access is used to access a slot of the array: it takes a circular array arr, index i, and the size of the array n, and returns arr[i].

Now we describe the implementation of the three main APIs. The push function is given in Figure 2.3. push uses the function grow to replace the deque's array. It takes a circular array circle, top t, and bottom b, and allocates and returns a new circular array circle' with double the size of the original one, such that they have the same values from t to b modulo their sizes:  $\forall i, t \leq i < b \implies circle[i] = circle'[i]$ . We skip the implementation detail here; refer to the link provided in chapter 1.

To push a value v, we start by reading bottom (b), top (t), and array (arr) along with its size (sz). If the array is full, i.e.  $sz \leq b - t + 1$ , the array is replaced by the **grow** function which returns a new circular array. Then, we read the array (arr') and its size (sz') again since they might have been changed, write the value v at arr'[b], and increment the bottom.

Next, the pop and steal function is given in Figure 2.4. We describe steal function first, because we have to understand it first to discuss pop. If  $b \leq t$ , there is nothing to steal. Otherwise, we read arr[t] and then attempt to steal by CAS-ing the top to t+1. Upon a successful CAS, the value we read earlier is returned. It is important to read arr[t] before CAS-ing: after CAS succeeds, the owner may wrap around and push a new element at arr[t+sz]. Then it is too late to read arr[t] because it was overwritten. On the other hand, it is correct to read it before CAS-ing, because arr[t] is guaranteed to stay the same until the successful CAS. We will discuss this point in more detail in the next section.

The last function to discuss is pop, which may look peculiar. The explanation is as follows:

• We start by immediately decrementing the bottom, as if the "bottom element" is already popped, even though the deque may be empty. If the deque *is* empty, we increment it back.

It is actually mandatory to decrement the bottom before reading the top. To see why, suppose the

```
Definition grow : val := λ: "circle" "t" "b",
  let: "sz" := Snd "circle" in
  let: "nsz" := #2 * "sz" in
  let: "narr" := AllocN "nsz" #0 in
  (* copy from Fst "circle" to "narr" for
    the indices in [t, b), omitted here *)
  ("narr", "nsz").
```

```
Definition push : val := λ: "deque" "v",
  let: "b" := !(bot "deque") in
  let: "t" := !(top "deque") in
  let: "circle" := !(arr "deque") in
  let: "sz" := Snd "circle" in
  (if: "t" + "sz" ≤"b" + #1
    then arr "deque" <- grow "circle" "t" "b"
    else #()
  ) ;;
  let: "circle'" := !(arr "deque") in
  let: "sz'" := Snd "circle'" in
  (access (Fst "circle'") "b" "sz'") <- "v" ;;
  bot "deque" <- "b" + #1.</pre>
```

Figure 2.3: The push function.

```
Definition pop :val := \lambda:"deque",
 let: "b" := !(bot "deque") - #1 in
 let: "circle" := !(arr "deque") in
 let: "sz" := Snd "circle" in
 bot "deque" <- "b" ;;
 let: "t" := !(top "deque") in
 if: "b" < "t" then
   (* empty pop *)
   bot "deque" <- "t" ;;NONE</pre>
 else let: "v" :=
   !(access (Fst "circle") "b" "sz") in
 if: "t" < "b" then
   (* normal case *)
   SOME "v"
 else let: "ok" :=
   CAS (top "deque") "t" ("t" + #1) in
 bot "deque" <- "t" + #1;;
 if: "ok" then SOME "v" (* popped *)
 else NONE. (* stolen *)
```

```
Definition steal :val := λ:"deque",
  let: "t" := !(top "deque") in
  let: "b" := !(bot "deque") in
  let: "circle" := !(arr "deque") in
  let: "sz" := Snd "circle" in
  if: "b" ≤ "t" then
    (* no chance *)
    NONE
  else let: "v" :=
   !(access (Fst "circle") "t" "sz") in
  if: CAS (top "deque") "t" ("t" + #1)
  then SOME "v" (* success *)
  else NONE. (* fail *)
```

Figure 2.4: The pop and steal functions.

owner does not decrement the bottom (b) and read the top (t), then learn b > t so we enter the "normal case" branch. But right after reading the top, stealers come in and steal every element. Now the owner resumes and pops the bottom element, not realizing that it was already stolen, resulting in an incorrect behavior since then multiple threads would execute the same task at the same time. Any attempt to read the bottom and top again in the normal case would suffer from the same problem of stealing after reading. We instead decrement the bottom prematurely, protecting the bottom element from the stealers.

Of course, this does not prevent the stealers from stealing the bottom element before even decrementing the bottom. Fortunately, this case is safe because then t would be large enough that we enter the "empty pop" branch.

- If there are more than one elements in the deque, we enter the "normal case" branch where the bottom element is simply returned. As we saw earlier, this element is safe from concurrent steals.
- If there is only one element, it is incorrect to just return the bottom element. Instead, we CAS the top just like stealing. The reason is because there is a potential conflict with concurrent stealers. Specifically, suppose a stealer reads the top and bottom before the owner decrements the bottom. Then the stealers enter the normal case where the CAS on top is attempted. At the same time, the owner starts to pop and notices that there is only one element. At this point, if the stealer succeeds the CAS, the owner should not be able to pop the only element since it is being stolen. This conflict is resolved by joining the stealers and CAS-ing the top. Despite popping from the top, this does not break the owner's LIFO behavior since there is no difference from the client's viewpoint.

#### 2.4 Intuition

Now we discuss a few observations that will be used in verifying Chase-Lev deque. First, only the owner can modify the bottom and the array. This is because **steal** does not do so. As a result, the owner can completely keep track of these two fields. This property is especially crucial in **grow** since the contents are copied over multiple steps, and in **pop** since it involves a complex reasoning on the bottom index.

Next, the top can only increase. This is because the only way to change the top is by reading its value t and CAS-ing it to t + 1. As a corollary, only one of the CAS attempts from t to t + 1 can ever succeed.

Finally, if the top t and bottom b satisfies t < b, the inequality stays true and arr[t] is preserved until t increases. This is because the only way to remove the element at the top (if it exists), either from pop or steal, is by CAS-ing the top. This property holds even if arr has already been replaced, because the values in the old array never get overwritten. This is why it is safe for steal to read arr[t] before CAS-ing the top even if the owner replaces the array in the meantime.

## Chapter 3. Background: Iris Separation Logic

Using the intuitive properties discussed in chapter 2, we can informally explain why Chase-Lev deque "works correctly", e.g. no elements are removed twice, the stealers remove the elements in FIFO order, and so on. However, we seek foundational verification; we need a way to formally express the specification and the reasoning.

In this chapter, we introduce Iris, a framework for concurrent separation logic [4, 23]. We do not explain Iris in full detail here; we only focus on the features that are relevant to the thesis and omit or simplify some subtle details for presentation. For a comprehensive introduction, refer to the documents in the Iris webpage [9, 24, 22].

Throughout the thesis, we may skip some parts of the notation if the context is obvious.

#### 3.1 Separation Logic

Concurrent separation logic [41] is a logic for concurrent programs. It is built around *resources* that can be manipulated, composed, and split. Resources enable modular, thread-local reasoning: instead of reasoning on thread interleavings, we can reason within a thread with the resources it owns.

One of the most common resources is a *points-to* assertion  $\ell \mapsto v$ , meaning that the heap has a location  $\ell$  pointing to the value v. A thread owning  $\ell \mapsto v$  is allowed to modify the value that  $\ell$  points to.

For resources P and Q, P\*Q is a resource called the *separating conjunction*, which asserts that the heap can be split into two fragments, one satisfying P and the other satisfying Q. Since the two parts must be disjoint, owning  $\ell_1 \mapsto v_1 * \ell_2 \mapsto v_2$  implies  $\ell_1 \neq \ell_2$ , systematically preventing multiple threads from writing to the same location in an unwanted way. Being able to separate the parts of the heap enables local reasoning when desired, hiding away the parts that are not necessary for the proof target.

Next, P \* Q is a resource called the *separating implication* or magic wand. It is a resource that, when combined with a heap satisfying P, asserts Q. In other words, we can combine P and P \* Q to obtain Q, consuming the two in the process.

To share a location between threads, we use fractional points-to: for a fraction q where  $0 < q \le 1$ , the resource  $\ell \mapsto^q v$  represents a fractional ownership of the location  $\ell$ . A thread with any fractional ownership can read a value from it, but only a thread with full (q = 1) ownership can write a value to it. This reflects that there should be either one writer and no readers, or multiple readers and no writer, to avoid data race. Note that  $\ell \mapsto^1 v$  is the same as  $\ell \mapsto v$ . Fractional points-to can be split and distributed

Points-To-Agree Points-To-Fractional 
$$\ell \mapsto^{q_1} v_1 * \ell \mapsto^{q_2} v_2 - v_1 = v_2 \qquad \qquad \ell \mapsto^{q_1+q_2} v * * \ell \mapsto^{q_1} v * \ell \mapsto^{q_2} v (0 < q_1, 0 < q_2)$$
 Load Store 
$$\{\ell \mapsto^q v\} ! \ell \{u.u = v * \ell \mapsto^q v\} \qquad \qquad \{\ell \mapsto v\} \ell \leftarrow w \{\ell \mapsto w\}$$

Figure 3.1: Proof rules for points-to.

to threads, or vice-versa, and two points-to from the same location has the same value. These rules are presented formally in Figure 3.1.

#### 3.2 Specification and Invariant

The specification of a program is represented with *Hoare triples* of the form  $\{P\}$  e  $\{v.Q\}$ . This means that given a resource P, the computation of e does not get stuck, and upon completion, it transforms P to a resource Q and returns a value v. These P and Q are called the *precondition* and the *postcondition*, respectively. We may skip v if e returns nothing.

For example, the bottom two rules of Figure 3.1 shows two specifications for reading and writing to a location. Load states that we can use any fractional points-to to read from it, and Store states that we can use a full points-to to write to it.

The power of Hoare triples in separation logic is that we can use the following frame rule to apply the specification for any heap containing P:

$$\frac{\{P\} e \{Q\}}{\{P * R\} e \{Q * R\}}$$

Therefore, we can apply Load or Store in the presence of other resources and they will not be affected by it.

For a data structure, a Hoare triple usually doesn't hold by itself, because the correctness relies on the internal property being held by the data structure. An *invariant* denoted  $\overline{I}$  is used to express this property and to prove the specification: it says that the property I holds at every program step.

Invariants are used with the following proof rule:

$$\frac{\left\{I*P\right\}e\left\{v.I*Q\right\}\quad\mathsf{atomic}(e)}{\boxed{I}\vdash\left\{P\right\}e\left\{v.Q\right\}}$$

This states that we can use the content I of the invariant during an atomic instruction, and must give it back to the invariant after the instruction. e being atomic ensures that the invariant is indeed satisfied at every step.

Using a Hoare triple and an invariant, the specification of push would look like the following:

$$\boxed{\mathsf{DequeInv}(p)} \vdash \{\mathsf{Deque}(p,\ell)\} \ push(p,v) \ \{\mathsf{Deque}(p,\ell+[v])\}$$

Unfortunately, this specification is not actually strong enough to be usable. We defer the discussion on a stronger specification to section 3.5.

### 3.3 Persistent Propositions

Some resources are *persistent*: once they hold, they stay true forever. Examples of persistent propositions are:

- Invariants |I|.
- Pure propositions such as equality and inequality.

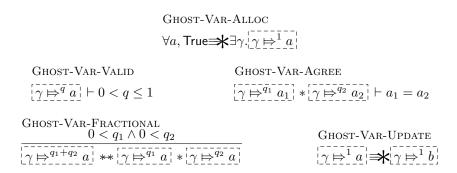


Figure 3.2: Proof rules for ghost variables.

• Persistent points-to  $\ell \mapsto^{\square} v$ . This means  $\ell$  is always a pointer pointing to the value v. Any fractional points-to can be made persistent, and once made so, it cannot be turned back to non-persistent and the corresponding value cannot be changed anymore.

#### 3.4 Ghost States

Iris supports *ghost states*, purely logical resources that are not directly affected by program executions. Unlike physical resources like points-to, users verifying a program have freedom over the choice of ghost states and can manipulate them as they wish. By tying them with physical states via an invariant, they can ensure some properties that physical states themselves cannot express.

A ghost state is an element defined in a set of values called a resource algebra (RA), equipped with suitable operators satisfying certain conditions like commutativity and associativity (exact conditions are omitted here). Users may use multiple independent ghost states using *ghost names*: the ghost states under different ghost names do not interfere with each other and may even be defined under different RAs. Given an element a of an RA, and a ghost name  $\gamma$ , the proposition  $\begin{bmatrix} \bar{a} \end{bmatrix}^{\gamma}$  asserts ownership of the ghost state a under the ghost name  $\gamma$ .

Iris comes with many pre-defined ghost states. One of them is a ghost variable, which is a resource similar to a fractional points-to but for a ghost (imaginary) location identified by  $\gamma$ . For ghost variables only, we will denote them as  $[\gamma \Longrightarrow^q a]$  instead of putting  $\gamma$  outside of the dotted box. Ghost variables provide the rules in Figure 3.2. Most rules are analogous to the fractional points-to rules, but section Ghost-Var-Update uses a view shift  $\Longrightarrow$ . Here,  $P \Longrightarrow^* Q$  means we can transform P to Q by updating ghost states. Note that this is different from  $\Longrightarrow$ : unlike a magic wand, a view shift may change the ghost states we own.

A more interesting example is a *monotonic natural number*, the ghost state of natural numbers which can only increase, with proof rules given in Figure 3.3. This is one of the *authoritative ghost states* which consist of an authoritative form  $\bullet$  and a snapshot form  $\circ$ . The authoritative form represents the exclusive ownership of the full information about the ghost state. This exclusiveness is expressed

```
\begin{array}{lll} \operatorname{Mono-Nat-Alloc} & \operatorname{Mono-Nat-Lb-Persistent} \\ \forall n, \operatorname{True} \Longrightarrow \exists \gamma. \boxed{\bullet \operatorname{MN} n}^{\gamma} & \operatorname{persistent} (\boxed{\circ \operatorname{MN} n}^{\gamma}) \\ & \operatorname{Mono-Nat-Auth-Exclusive} & \underline{n \leq n'} \\ \hline (\bullet \operatorname{MN} n)^{\gamma} * \boxed{\bullet \operatorname{MN} n'}^{\gamma} \vdash \operatorname{False} & \operatorname{Mono-Nat-Lb-Valid} \\ & \operatorname{Mono-Nat-Lb-Valid} & \operatorname{Mono-Nat-Lb-Get} \\ \hline (\bullet \operatorname{MN} n)^{\gamma} * \boxed{\circ \operatorname{MN} n}^{\gamma} \vdash m \leq n & \boxed{\bullet \operatorname{MN} n}^{\gamma} \xrightarrow{-*} \boxed{\circ \operatorname{MN} n}^{\gamma} \end{array}
```

Figure 3.3: Proof rules for monotonic natural numbers.

in Mono-Nat-Auth-Exclusive. Since it asserts the full ownership, we can update it via Mono-Nat-Auth-Update. However, being monotonic, the contained number can only increase. On the other hand, the snapshot form represents the persistent knowledge of the ghost state at some past point of execution. As such, using Mono-Nat-Lb-Valid, we can infer that the number held by the authoritative form is at least the number taken by the past snapshot. Finally, we can take a new snapshot of the authoritative form using Mono-Nat-Lb-Get. Note that this rule is not a view shift: it rather says that the ownership of  $\left[ \bullet \overline{\mathsf{MN}} \, n \right]^{\gamma}$  automatically derives the ownership of  $\left[ \bullet \overline{\mathsf{MN}} \, n \right]^{\gamma}$ . Since  $\left[ \bullet \overline{\mathsf{MN}} \, n \right]^{\gamma}$  is persistent, we retain  $\left[ \bullet \overline{\mathsf{MN}} \, n \right]^{\gamma}$  after applying this rule.

## 3.5 Linearizability and Logical Atomicity

While Hoare triples are intuitive, they are limited to sequential programs: since push is not a physically atomic instruction, it is not possible to open an invariant around it. Consequently, a thread trying to use this specification must exclusively own the resource  $WSDeque(p,\ell)$  instead of sharing it via an invariant. This defeats the purpose of supporting concurrent accesses to the data structure. Furthermore, the specification given in section 3.2 does not make sense with concurrency: the contents of the deque before and after pushing may be completely different because other threads can change it in the meantime.

Instead, the specification of concurrent data structures is commonly given as *linearizability* [20]. A concurrent data structure is linearizable if its concurrent invocations can be reordered to sequential invocations with the same effect on the data structure. For example, suppose the owner pushes a value to an empty deque, but right before returning from the operation, a stealer steals the same value. Although the deque was accessed concurrently, this has the same effect as pushing and then stealing sequentially.

A common way to prove the linearizability is to identify the *commit points* of each operation: the point in which the operation "appears" to take place atomically from a client's viewpoint. Functions with such commit points are called *logically atomic*. For push, reading the bottom index does not affect the data structure; neither does writing a value to the array since it cannot be accessed by clients yet. Only after incrementing the bottom index is the whole effect of the operation visible to other clients. Therefore, incrementing the bottom is the commit point of push. Once we identify the commit points of all other operations, we can order any set of concurrent invocations by the order in which their commit points are reached, which gives a linearization order of these invocations.

The idea of commit points is encoded in *logically atomic triples* (LATs) [25]. They are denoted as

 $\langle P \rangle e \langle v.Q \rangle$ , which means that there is a commit point in e which transforms the resource P to Q, and e returns v in the end. To distinguish with Hoare triples, P and Q are called the *atomic precondition* and the *atomic postcondition*, respectively.

Just like physically atomic instructions, we can open an invariant around a logically atomic instruction:

$$\frac{\langle I*P\rangle\,e\,\langle v.I*Q\rangle}{\boxed{I}\,\vdash \langle P\rangle\,e\,\langle v.Q\rangle}$$

Of course, using this rule requires proving the LAT  $\langle I*P\rangle e\ \langle v.I*Q\rangle$ , i.e. that e really is logically atomic. As we can't just open invariants forever, how can a data structure prove that its methods are logically atomic? That is done by the following rule:

$$\frac{\forall \Phi. \{\mathsf{AU}_{P,Q}(\Phi)\} \ e \ \{\Phi\}}{\langle P \rangle \ e \ \langle v.Q \rangle}$$

Here, we receive an atomic update  $\mathsf{AU}_{P,Q}(\Phi)$  as a precondition. It is a resource representing the right and obligation to commit. At the commit point of e, we open  $\mathsf{AU}$  just like an invariant and obtain P. Then in a single physical step, we must transform P into Q. Upon transformation,  $\mathsf{AU}$  is consumed and we get the assertion  $\Phi$ . Since  $\Phi$  is universally quantified and contained in the postcondition, the only way to prove this Hoare triple is by completing the obligation of  $\mathsf{AU}$ . This can be roughly formalized as  $\mathsf{AU} \Longrightarrow (P * (Q \Longrightarrow \Phi))$ , except for the single physical step requirement.

## Chapter 4. Verification of Chase-Lev Deque without Resizing

Now we are almost ready to give a specification to Chase-Lev deque and verify it. Before that, we need to extend the concept of logically atomic triples in order to give a proper specification. We start this chapter with the extension of LATs to support *private postconditions*. Then we present the formal verification of Chase-Lev deque but without array resizing; it will be added back in the next chapter.

### 4.1 Specification with Private Postconditions

Using a logically atomic triple we saw in chapter 3, the modified specification of push would look like the following:

$$\boxed{\mathsf{DequeInv}(p)} \vdash \langle \mathsf{Deque}(p,l) \rangle \ push(p,v) \ \langle \mathsf{Deque}(p,l+[v]) \rangle$$

However, this specification is still incorrect: this allows concurrent pushes to the same deque p, which lead to data race on the bottom index or the array. Since Iris is a sound logic, such a specification cannot be proven either. The problem is that **push** is not supposed to be called by multiple threads concurrently, but only by the sole owner. At the same time, we do have to use LATs because we want to prove linearizability.

What is missing is an exclusive resource for the owner of the deque p, which we will denote as  $\mathsf{OwnDeque}(p)$ . Threads accessing a shared work-stealing deque have uneven authority over it. Specifically, only the owner of the deque may push or pop an element from it, and has a total control over some information that only  $\mathsf{push}$  and  $\mathsf{pop}$  may alter.  $\mathsf{OwnDeque}(p)$  represents this owner-exclusive permission to  $\mathsf{push}$  or  $\mathsf{pop}$  from the deque p.

Then is the following the correct specification for push?

$$\boxed{\mathsf{DequeInv}(p)} \vdash \langle \mathsf{Deque}(p,l) * \mathsf{OwnDeque}(p) \rangle \ push(p,v) \ \langle \mathsf{Deque}(p,l+[v]) * \mathsf{OwnDeque}(p) \rangle$$

The answer is no: this still allows concurrent pushes to p. Clients using the deque p would share the permission to use p by storing  $\exists l.\mathsf{Deque}(p,l)$  in the invariant. But with the specification like this, they can just store  $\mathsf{OwnDeque}(p)$  in the invariant as well and call  $\mathsf{push}$  concurrently. This is again an unsafe specification which cannot be proven either.

The root of the problem is that OwnDeque is supposed to be stay local to the owner thread, not shared with an invariant. To represent this restriction, we move OwnDeque outside of the LAT:

$$|\mathsf{DequeInv}(p)| \vdash \mathsf{OwnDeque}(p) \twoheadrightarrow \langle \mathsf{Deque}(p,l) \rangle push(p,v) \langle \mathsf{Deque}(p,l+[v]) \ast \mathsf{OwnDeque}(p) \rangle$$

This resolves the problem for the atomic precondition, but the atomic postcondition still has an issue: OwnDeque is consumed at the commit point so it cannot be used later. This may not be a problem for push since the commit point is at the end of the operation anyway, but that is not the case for pop.

We cannot move OwnDeque out of the atomic postcondition, as that just separates OwnDeque from the specification. We have to require this OwnDeque as the postcondition, but at the end of the program, not at the commit point. This is called the *private postcondition*. A LAT with private postcondition R is denoted as  $\langle P \rangle e \langle v.Q; R \rangle$ . When proving this LAT, we are given  $\mathsf{AU}_{P,Q}(R \twoheadrightarrow \Phi)$ . That is, upon

$$\begin{cases} \text{NEW-SPEC} \\ \{0 < n\} \ new\_deque(n) \ \Big\{ p. \exists \gamma. \boxed{\mathsf{DequeInv}^{\gamma}(p)} \ * \ \mathsf{Deque}^{\gamma}([]) \ * \ \mathsf{OwnDeque}^{\gamma}(p) \Big\} \end{cases}$$
 
$$\begin{aligned} & \mathsf{PUSH-SPEC} \\ & \boxed{\mathsf{DequeInv}(p)} \vdash \mathsf{OwnDeque}(p) \twoheadrightarrow \langle \mathsf{Deque}(l) \rangle \ push(p,v) \ \langle \mathsf{Deque}(l+[v]); \mathsf{OwnDeque}(p) \rangle } \end{aligned}$$
 
$$\begin{aligned} & \mathsf{POP-SPEC} \\ & \boxed{\mathsf{DequeInv}(p)} \vdash \mathsf{OwnDeque}(p) \twoheadrightarrow \\ & \langle \mathsf{Deque}(l) \rangle \ pop(p) \ \bigg\langle w. \exists l'. \mathsf{Deque}(l') \ * \ \bigvee \left\{ \begin{array}{l} w = None \land l = l' \\ \exists v. w = Some(v) \land l = l' + [v] \end{array} \right. ; \mathsf{OwnDeque}(p) \right\} \end{aligned}$$
 
$$\begin{aligned} & \mathsf{STEAL-SPEC} \\ & \boxed{\mathsf{DequeInv}(p)} \vdash \langle \mathsf{Deque}(l) \rangle \ steal(p) \ \bigg\langle w. \exists l'. \mathsf{Deque}(l') \ * \ \bigvee \left\{ \begin{array}{l} w = None \land l = l' \\ \exists v. w = Some(v) \land l = [v] + l' \end{array} \right. \end{aligned}$$

Figure 4.1: Specification of Chase-Lev deque.

committing, we receive  $R ext{-}* \Phi$ ; we can then normally progress through the proof, but at the end of the program we have to prove and consume R in order to obtain  $\Phi$  and finish the proof. Currently Iris' Coq formalization of LAT does not support private postconditions, so we slightly extended the framework for them.

Now we can finally give the specification of Chase-Lev deque as Figure 4.1. New-Spec is just a regular Hoare triple because there is no concurrency involved here. Push-Spec is straightforward. Pop-Spec and Steal-Spec have case analysis on whether the attempt succeeded or not. If successful, the top or bottom element is removed from the abstract state and the removed element is returned. Otherwise, the abstract state does not change and nothing is returned.

#### 4.2 Resource Definitions

So far, we haven't defined the resources Deque, Dequelnv, and OwnDeque. The clients of the deque can just use our specification without delving into the details of each resource, but since we have to prove the specification, we should define each resource using the pre-defined propositions.

The definition of our deque resources use several ghost states, so we need multiple ghost names. Fortunately, we can group multiple ghost names into a single one<sup>1</sup>. In our case, we define  $\gamma$  as a tuple of three other ghost names ( $\gamma_q, \gamma_{sw}, \gamma_{state}$ ).

We use one type of ghost state for each ghost name, for a total of three. For  $\gamma_q$ , we use a ghost variable representing the abstract state of the deque. This is shared between Deque and Dequelnv. For  $\gamma_{sw}$  (shorthand for "single-writer"), we use a ghost variable for the values that only the owner of the deque can alter. This is shared between Dequelnv and OwnDeque. Finally, for  $\gamma_{state}$ , we use a custom ghost state  $^{\bullet}$ , encoding some properties about the synchronization guarantee in Chase-Lev deque.

Deque is simply defined as a ghost variable for  $\gamma_a$ :

$$\mathsf{Deque}^{\gamma}(l) := \exists \gamma_q. \gamma = (\gamma_q, \neg, \neg) * \boxed{\gamma_q \mapsto^{1/2} \overline{l}}$$

<sup>&</sup>lt;sup>1</sup>Formally, a ghost name is represented by a natural number, and multiple ghost names can be encoded into a single ghost name with a suitable bijection between  $\mathbb{N}^k$  and  $\mathbb{N}$ .

Dequelnv consists of the points-to for each field of the deque, and the three ghost states:

Finally, OwnDeque consists of the remaining points-to, and the ghost state for  $\gamma_{sw}$ :

In chapter 2, we listed several properties about Chase-Lev deque: (1) only the owner can modify the bottom and the array; (2) the top can only increase; (3) once t < b holds, it stays true and the top element is preserved until t increases. Among those, (1) is encoded in our invariant as fractional points-to. Since steal does not have access to OwnDeque, they only get a half points-to for arr and bot, so they cannot write to them. On the other hand, the owner of the deque can combine OwnDeque with Dequelny to get a full points-to for them and use it to write a new value.

- (2) and (3) are encoded as our custom ghost state we will call the deque state. It is an authoritative ghost state, and there are two types of ghost states:
  - $(L,t,b)^{\gamma}$ , the authoritative form, represents the ownership of the current state of the deque. The state consists of the whole array L, top index t, and bottom index b.
  - $\stackrel{\frown}{\stackrel{\frown}{\boxtimes}} (L,t,b)\stackrel{\frown}{\stackrel{\frown}{\sqcup}}$ , the snapshot form, represents the persistent knowledge of a past state of the deque.

The proof rules for deque state are listed in Figure 4.2. Dqst-Auth-Alloc, Dqst-Frag-Persistent, and Dqst-Frag-Get are similar to the proof rules for monotonic natural number as seen in chapter 3. Dqst-Write-Array, Dqst-Push, Dqst-Pop, and Dqst-CAS-Top are the update rules that change the internal states.<sup>2</sup>

Dqst-Frag-Valid is the key rule that contains our reasoning of top element preservation. Here, Preservation<sub>1,2</sub> is a persistent proposition and the shorthand for  $(t_1 = t_2 \land t_1 < b_1) \implies t_2 < b_2 \land L_1[t_1] =$  $L_2[t_2]$ . Put together, this rule means: given a snapshot of a past state 1, and the ownership of the current state 2, we learn  $t_1 \leq t_2$ ; and if t has not increased and t < b initially held, then it still holds in the current state and the top element also stays the same.

<sup>&</sup>lt;sup>2</sup>It turns out Dqst-Write-Array and Dqst-Pop don't have to be a view shift; a wand is sufficient. We used a view shift here for consistency, and either of them works in verifying the deque operations.

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma . \boxed{ 2 (L,1,1,1) }^{\gamma} \qquad \text{persistent}$$

$$|L| \neq 0 \Longrightarrow \exists \gamma .$$

Figure 4.2: Proof rules of deque state.

Alloc 
$$\{\mathsf{True}\}\ AllocN(n,v)\ \{l.l\mapsto [v,v,\cdots,v]\wedge |l|=n\}$$
 
$$\mathsf{Ref}$$
 
$$\{\mathsf{True}\}\ ref(v)\ \{l.l\mapsto v\}$$

Figure 4.3: Specification of allocation in HeapLang.

#### 4.3 Verification of Each Function

#### 4.3.1 New

To verify new\_deque (Figure 2.2), we have to allocate all physical and ghost resources required by the specification. HeapLang has physical allocation rules given in Figure 4.3. Following the code, we allocate the array using Alloc, and allocate the pair (array, sz), top index, and bottom index using Ref. Then we make the pair points-to persistent, and we have  $C \mapsto^{\square} (arr, n) * arr \mapsto L * top \mapsto 1 * bot \mapsto 1$ , where  $L = [v, v, \dots, v]$  with length n.

Next, we allocate the ghost variables  $[\gamma_q \mapsto^1]$  and  $[\gamma_{sw} \mapsto^1 (L, 1, \mathsf{False})]$  via Ghost-Var-Alloc, and  $[(L, t, b)]^{\gamma_{state}}$  via Dqst-Auth-Alloc. Finally, we split the points-to and ghost variables, group the resources as Dequelnv(p) \* Deque(l) \* OwnDeque(p), and turn Dequelnv(p) as the invariant Dequelnv(p), which proves the postcondition.

#### 4.3.2 Push

As we are skipping the resizing part for now, the implementation of push should be changed as well. We use the same code as Figure 2.3, except:

• If  $t+sz \le b+1$ , we call diverge which just runs an infinite loop. Since a Hoare triple only concerns with the condition at termination (remind that LATs are also proved using a Hoare triple with the atomic update AU), we do not need to consider the branch ending up with an infinite loop. Specifically, diverge has the specification {True} diverge() {False}, and using this specification we obtain False as a resource, which can prove anything.

• We skip "circle'" := !(arr "deque") and "sz'" := Snd "circle", since there is no resizing.

All other occurrences of "circle'" and "sz'" are replaced with "circle" and "sz" respectively.

Now we verify the variant of push without resizing. In the beginning, we have  $\mathsf{OwnDeque}(p)$ :  $C \mapsto^{\square} (arr, |L|) * [\gamma_{\underline{sw}} \mapsto^{1/2} (\underline{L}, \underline{b}, \mathsf{False})] * arr \mapsto^{1/2} L * bot \mapsto^{1/2} b$ . To read from the array or bottom, we just use the resources in  $\mathsf{OwnDeque}$  since we can use any fraction to read. To read from the top, we open the invariant to get  $top \mapsto t$ , read the value t using it, and close the invariant.

To write to the array, we can't use  $arr \mapsto^{1/2} L$ ; we should open the invariant to get the other half. Note that Dequelnv has an existential for L, and opening the invariant does not guarantee by itself that L from Dequelnv equals L from OwnDeque. That is where fractional resources come into play: since we have  $arr \mapsto^{1/2} L'$  and  $arr \mapsto^{1/2} L$ , we can use Points-To-Agree to prove L' = L.

Now we can combine them into  $arr \mapsto L$ , use Store to write a new value, which turns it into  $arr \mapsto L_2$  for some new list  $L_2$ , and split them back into  $arr \mapsto^{1/2} L_2 * arr \mapsto^{1/2} L_2$ . However, we cannot close the invariant yet. We would like to pick  $L_2$  as its existential for L, but this does not work because we don't have  $[\gamma_{sw} \mapsto^{1/2} (L_2, b', pop)]$ . Instead we have  $[\gamma_{sw} \mapsto^{1/2} (L_2, b', pop)]$ , so we should update this.

To do so, we apply the same procedure as above but for ghost variables: since we have the other half  $\gamma_{\underline{sw}} \Rightarrow^{1/2} (L, b, \mathsf{False})$  from OwnDeque, we can use Ghost-Var-Agree to prove  $(L, b', pop) = (L, b, \mathsf{False})$ . Then we combine the two identical ghost variables into  $\gamma_{\underline{sw}} \Rightarrow^1 (L, b, \mathsf{False})$  by Ghost-Var-Fractional, update it to  $\gamma_{\underline{sw}} \Rightarrow^1 (L_2, b, \mathsf{False})$  by Ghost-Var-Fractional.

We also have to update  $[\underbrace{\gamma_q \mapsto^{1/2} L[t..b]}]$  to  $[\underbrace{L_2,t,b}]$ , which can be done by Dqst-Write-Array. We don't need to update  $[\underbrace{\gamma_q \mapsto^{1/2} L[t..b]}]$ : we can instead prove  $L_2[t..b] = L[t..b]$  because we actually wrote right next to the boundary of [t..b], so there is nothing to update. Finally, we can prove  $|L_2| = |L| = n$ . Now that we changed all occurrences of L to  $L_2$ , we can close the invariant by choosing  $L_2$  as its existential for L.

After writing to the array, we have to increment the bottom. This is the commit point of push. We open not only the invariant, but also the atomic update AU to get the atomic precondition  $\mathsf{Deque}(l)$ . Now we have to turn it into the atomic postcondition  $\mathsf{Deque}(l+[v])$ , which can be done by combining the two ghost variables for  $\gamma_q$  and updating them, just like what we did for writing to the array. After committing AU and closing the invariant, we are left with  $\mathsf{OwnDeque} \twoheadrightarrow \Phi$  and  $\mathsf{OwnDeque}$ . We combine them to get  $\Phi$ , finishing the proof.

To describe the proof in more detail, let's progress through the program one by one:

- Start with DequeInv \* OwnDeque.
- "b" := !(bot "deque"): read b from  $bot \mapsto^{1/2} b$ .
- "t" := !(top "deque"): we do not have a points-to for top, but it is hidden in Dequelnv. We proceed as follows:
  - (1) Open Dequelnv. We obtain all resources in Dequelnv for some instantiation of its existentials, but this time we are only interested in  $top \mapsto t$ .
  - (2) Read t from  $top \mapsto t$ .
  - (3) Close Dequelnv with the same instantiation of its existentials by returning all resources for it including  $top \mapsto t$ .
- "circle" := !(arr "deque"): read C from  $C \mapsto^{\square} (arr, |L|)$ .
- "sz" := Snd "circle": sz = |L|.

- Assume t + sz > b + 1, so there is a slot in the array to write v; otherwise we end up with an infinite loop so we are done.
- (access (Fst "circle") "b" "sz") <- "v": to write to the array, we need a full points-to. We proceed as follows:
  - (1) Open Dequelnv. Let  $\gamma'_q, \dots, pop'$  be the instantiation for its existentials.
  - (2) From  $\gamma = (\gamma_q, \gamma_{sw}, \gamma_{state})$  and  $\gamma = (\gamma'_q, \gamma'_{sw}, \gamma'_{state})$ , prove  $\gamma_q = \gamma'_q, \gamma_{sw} = \gamma'_{sw}$ , and  $\gamma_{state} = (\gamma_q, \gamma_{sw}, \gamma_{state})$  $\gamma'_{state}$ . Similarly, prove C = C', top = top', and bot = bot', and then prove arr = arr' and |L| = n'.
  - (3) Use Ghost-Var-Agree on  $\gamma_{sw}$  to also prove (L', b', pop') = (L, b, pop).
  - (4) Combine  $arr \mapsto^{1/2} L * arr \mapsto^{1/2} L$  into  $arr \mapsto L$ , apply Store to change it to  $arr \mapsto L[b \leftarrow v]$ , and split it back.
  - (5) Similarly, use Ghost-Var-Fractional and Ghost-Var-Update to update the ghost variable for  $\gamma_{sw}$ to  $[\gamma_{sw} \mapsto^{1/2} (L[b \leftarrow v], b, pop)]$ .

    (6) Update  $[ \stackrel{\bullet}{=} (L, t, b)]$  to  $[ \stackrel{\bullet}{=} (L[b \leftarrow v], t, b)]$  by Dqst-Write-Array.

  - (7) Close Dequelnv with  $L[b \leftarrow v]$  as the choice of L.
- bot "deque" <- "b" + 1: this is the commit point.
  - (1) Open Dequelnv and prove  $(L', b', pop) = (L[b \leftarrow v], b, False)$  by Ghost-Var-Agree.
  - (2) Combine and update the resources to  $bot \mapsto b+1$  by Store,  $\gamma_{sw} \mapsto^1 (L[b \leftarrow v], b+1, \mathsf{False})$  by Ghost-Var-Update, and  $\lfloor \underbrace{b \leftarrow v}, t, b+1 \rfloor$  by Dqst-Push, then split them back as required.
  - (3) Open the atomic update AU which contains Deque and prove  $l = L[b \leftarrow v][t..b)$  by Ghost-Var-Agree. This also implies  $l + [v] = L[b \leftarrow v][t..b + 1)$ .
  - (4) Combine and update the resources to  $\gamma_q \Rightarrow l + [v]$  by Ghost-Var-Update, and split it back.
  - (5) Now we have the atomic postcondition Deque(l+[v]), so commit AU and obtain  $\Phi$ .
  - (6) Close the invariant with b+1 as the choice of b.
- In the end, we have  $\mathsf{OwnDeque}(p) * (\mathsf{OwnDeque} * \Phi)$ . Combine them into  $\Phi$  and finish the proof.

#### Pop 4.3.3

Verification of pop roughly follows a similar procedure to verifying push, except for some key differences. First, instead of changing L, we change pop. This value is initially False. When we decrement the bottom in the beginning, we set it to True and update the ghost variables accordingly. When we increment it back, or end up in the normal case, we set it back to False and update the ghost variables again.

Another difference is that the commit point depends on the number of elements in the deque when reading the top. If the deque is empty, reading the top is the commit point since that's when we observe the empty deque. The atomic precondition and postcondition have the same resources in this case, so we can just open AU and commit immediately. If the deque has more than one element, reading the top is still the commit point. If the deque has exactly one element, CAS-ing the top is the commit point, regardless of whether the CAS succeeds or not.

Note that this is not the only possible way to determine the commit points. For example, we believe that the following approach also works: if the deque has more than one element, decrementing the bottom is the commit point. In fact, this may intuitively make more sense to some because that's when the bottom element is removed. This approach, however, has a big downside: we have to do a case

analysis on which value of the top index will be read in the next step. This reasoning is possible and actually necessary in some data structures [25], but it's complicated and beyond the scope of this thesis.

Here is the proof outline in more detail:

- Start with DequeInv \* OwnDeque.
- "b" := !(bot "deque") #1: read b from  $bot \mapsto^{1/2} b$ .
- "circle" := !(arr "deque"): read (arr, |L|) from  $C \mapsto^{\square} (arr, |L|)$ .
- "sz" := Snd "circle": sz = |L|.
- bot "deque" <- "b": open Dequelnv, update  $bot \mapsto b$  to  $bot \mapsto b-1$ , update  $\gamma_{sw} \mapsto^1 (L, b, \mathsf{False})$  to  $\gamma_{sw} \mapsto^1 (L, b, \mathsf{True})$ , and close Dequelnv with True as the choice of pop.
- "t" := !(top "deque"):
  - (1) Open Dequelnv and read t from  $top \mapsto t$ .
  - (2) If t < b (normal case), open AU as well, update the ghost states to  $\lceil \gamma_g \Rightarrow^1 L[t..b-1) \rceil * \lceil \gamma_{sw} \Rightarrow^1 (L,b-1,\mathsf{False}) \rceil * \lceil (L,b-1) \rceil$  by Dqst-Pop, and commit.
  - (3) If b < t (empty case), open AU and immediately commit.
  - (4) Otherwise, do nothing. In all cases, we close Dequelnv in the end.
- If b < t (empty case), bot "deque" < "t": open Dequelnv, update  $bot \mapsto b-1$  to  $bot \mapsto b$ , update  $|\gamma_{sw} \mapsto^1 (L, b, \text{True})|$  back to  $|\gamma_{sw} \mapsto^1 (L, b, \text{False})|$ , and close Dequelnv with False as the choice of pop. We already committed, so we have OwnDeque  $*\Phi$  and we are done.
- "v" := !(access (Fst "circle") "b" "sz"): read v from  $arr \mapsto^{1/2} L$ .
- If t < b (normal case), we already committed so we are done.
- CAS (top "deque") "t" ("t" + #1): open Dequelnv and attempt CAS with  $top \mapsto t'$ .
- If t=t', we succeed the CAS and get  $top \mapsto t+1$ . Open AU, update the ghost states to  $\gamma_q \mapsto^1 L[t+1..b] * [(L,t+1,b)],$  commit, and close Dequelov with t+1 as the choice of t.
- If  $t \neq t'$ , we fail the CAS. Open AU, immediately commit, and close Dequelnv.
- bot "deque" <- "t" + #1: open Dequelnv, update  $bot \mapsto b-1$  to  $bot \mapsto b$ , update  $\gamma_{\underline{sw}} \mapsto^{\underline{1}} (\underline{L}, \underline{b}, \mathsf{True})$  to  $\gamma_{\underline{sw}} \mapsto^{\underline{1}} (\underline{L}, \underline{b}, \mathsf{False})$ , and close Dequelnv with False as the choice of pop.
- In the end, we have  $\Phi * \mathsf{OwnDeque}$  so we are done.

#### 4.3.4 Steal

When verifying steal, we don't have OwnDeque, so we cannot keep track of the array and bottom. Let  $t_i$ ,  $b_i$ , and  $L_i$  be the top, bottom, and array, respectively, when Dequelnv is opened for the i-th time. We open the invariant four times: (1) reading the top, (2) reading the bottom, (3) reading the value at the top, and (4) CAS-ing the top.

The commit points are similar to pop. If the deque is "empty" (no element, or one element and the owner is trying to pop), reading the bottom is the commit point. Otherwise, CAS-ing the top is the commit point.

The main challenge in verifying steal is to prove that the value at the top is preserved at the successful CAS. Specifically, we have to prove  $L_3[t_3] = L_4[t_4]$ . The reason is that when opening AU to get Deque(l) and committing, we should prove  $\exists l'.l = [L_3[t_3]] + l'$  and update  $[\gamma_q \mapsto^{\bar{1}} l]$  to  $[\gamma_q \mapsto^{\bar{1}} l']$ . We have  $l = L_4[t_4..b_4)$  from  $[\gamma_g \mapsto^{\bar{1}/2} l]$  and  $[\gamma_g \mapsto^{\bar{1}/2} L_4[t_4..b_4)]$ , so the first element of l is  $L_4[t_4]$ . Therefore our goal reduces to  $L_3[t_3] = L_4[t_4]$ .

The intuitive reason why this holds is because if CAS succeeded, the top has never increased between reading the top and CAS-ing it, i.e.  $t_1 = t_2 = t_3 = t_4$ . Since we did not take the "no chance" branch, we have  $t_1 < b_2$ , so  $t_2 < b_2$  as well. We saw in chapter 2 that once t < b holds, L[t] is preserved until t increases; therefore,  $L_3[t_3] = L_4[t_4]$ .

This reasoning can be expressed in Iris using our deque state  $\ ^{\ }$ . Each time we open the invariant, we take the persistent snapshot  $\ ^{\ }$  by Dqst-Frag-Get. From the second opening onwards, we use  $\ ^{\ }$  from the current opening,  $\ ^{\ }$  from the previous opening, and Dqst-Frag-Valid to prove that the top has not decreased. This gives  $t_1 \le t_2 \le t_3 \le t_4$ . If CAS succeeded, we know  $t_1 = t_4$ , so all of the  $t_i$  must equal. To prove  $L_3[t_3] = L_4[t_4]$ , during the third opening, we use  $\ ^{\ }$ , the second  $\ ^{\ }$ , and Dqst-Frag-Valid to get Preservation<sub>2,3</sub>. Then during the fourth opening, we learn  $t_2 = t_3$  and thus  $t_3 < t_3$ . Finally, we apply Dqst-Frag-Valid again to get Preservation<sub>3,4</sub> and thus  $L_3[t_3] = L_4[t_4]$ .

Here is the proof outline in more detail:

- We start with just Dequelnv.
- "t" := !(top "deque"): open Dequelnv, read  $t_1$  with  $top \mapsto t_1$ , take a snapshot  $\stackrel{\mbox{\tiny $C$}}{\hookrightarrow} (L_1, t_1, b_1)$  by Dqst-Frag-Get, duplicate  $C \mapsto^{\square} (arr, n)$  and keep it locally, and close Dequelnv.
- "b" := !(bot "deque"): open Dequelnv, read  $b_2$  with  $bot \mapsto b_2$ , use  $\begin{cases} \begin{cases} \begin{case$
- "circle" := !(arr "deque"): read (arr, n) from  $C \mapsto^{\square} (arr, n)$ .
- "sz" := Snd "circle: sz = n.
- If  $b_2 \leq t_1$  (no chance), we already committed and we are done.
- "v" := !(access (Fst "circle") "t" "sz"): open Dequelnv, read  $v = L_3[t_3]$  with  $arr \mapsto^{1/2} L_3$ , use  $\ ^{\ }$  to prove  $t_2 \leq t_3 \wedge \mathsf{Preservation}_{2,3}$  by Dqst-Frag-Valid, take a snapshot  $\ ^{\ }$   $(L_3,t_3,b_3)$ , and close Dequelnv.
- CAS (top "deque") "t" ("t" + #1): open Dequelnv, use  $\ ^{\bullet}$  to prove  $t_3 \leq t_4$  by Dqst-Frag-Valid, and attempt CAS with  $top \mapsto t_4$ .
- If  $t_1 = t_4$ :
  - (1) We succeed the CAS and we get  $top \mapsto t_1 + 1$ . Since  $t_1 \le t_2 \le t_3 \le t_4 = t_1$ , we get  $t_1 = t_2 = t_3 = t_4$  and in particular  $t_2 = t_1 < t_2$ .
  - (2) Use Preservation<sub>2,3</sub> and  $t_2 = t_3 \wedge t_2 < b_2$  to prove  $t_3 < b_3$ .
  - (3) Use  $\stackrel{\triangle}{=}$  and  $\stackrel{\triangle}{=}$   $(L_3, t_3, b_3)$  to prove  $t_4 < b_4$  and  $L_3[t_3] = L_4[t_4] = v$  by Dqst-Frag-Valid.
  - (4) Open AU, update the ghost states to  $[\gamma_q \Rightarrow L_4[t_4+1..b_4]] * L_4, t_4+1, b_4$  by Dqst-CAS-Top, commit, and close Dequelnv.

Figure 4.4: Proof rules of ghost map.

- If  $t_1 \neq t_4$ , we fail the CAS. Open AU, immediately commit, and close Dequelnv.
- In the end in both cases, we have  $\Phi$  which finishes the proof.

#### 4.4 Construction of the Ghost Deque State

So far, we gave the specification of Chase-Lev deque, defined the resources necessary for the specification, and used them to verify each operation. But there is one last problem remaining: we have to prove that the ghost deque state and actually make sense. The proof rules for Iris' built-in ghost states like ghost variables and monotonic natural numbers have already been proven sound, but not our custom ghost state.

Fortunately, we can simply define our ghost states using other pre-defined ghost states, just like the resources in the specification like Deque. To define the ghost deque state, we need one more type of built-in ghost states called a *ghost map*. This ghost state maintains a finite partial map from a domain set to a value set.  $\lceil \bullet \text{Map } M \rceil^{\gamma}$  represents the authoritative ownership of the whole map M. For each key k with its associated value v,  $\lceil \bullet \text{Map } k \mapsto \square v \rceil^{\gamma}$  represents the persistent knowledge of this key-value pair. We can insert a new key-value pair  $k \mapsto v$  into M, provided that k is not mapped in M already. The proof rules of ghost map are listed in Figure 4.4.

We are finally ready to define the deque state. Remind that deque state should encode two properties: (1) the top only increases, and (2) the inequality between the top and bottom, and the top element are preserved along with the top.

For (1), we simply use a monotonic natural number. But we can use the same ghost state to also

represent one half of (2). Specifically, let 
$$tbs(t,b) := \begin{cases} 2t+1 & \text{if } t < b \\ 2t & \text{otherwise} \end{cases}$$
. Then we can prove:

$$tbs(t_1, b_1) \le tbs(t_2, b_2) \iff t_1 \le t_2 \land (t_1 = t_2 \land t_1 < b_1 \implies t_2 < b_2)$$

Therefore, the ownership of  $[\bullet MN \ tbs(t,b)]^{\gamma}$  ensures that (1) t only increases, and (2) once t < b holds, it keeps true until t increases.

For the other half of (2), we use a ghost map  $\left[\underbrace{\bullet \mathsf{Map}\,M}\right]^{\gamma}$ , assigning to each top index its preserved top element. Each mapping  $\left[\underbrace{\circ \mathsf{Map}\,i \mapsto^{\square}v}\right]^{\gamma}$  represents that v is the preserved top element when the top index was i. If t = b, there is a mapping for each i from 1 to t - 1. If t < b, there is a mapping up to t.

<sup>&</sup>lt;sup>3</sup>There is also a ghost state for non-persistent key-value ownership so that we can change its associated value, but we don't use it here.

$$\exists \gamma_{tb}, \gamma_{elt}, M.$$

$$\star \begin{cases} \gamma = (\gamma_{tb}, \gamma_{elt}) \\ \underbrace{\bullet \mathsf{MN} \ tbs(t, b)}^{\gamma_{elt}} \\ \underbrace{\bullet \mathsf{Map} \ M}^{\gamma_{elt}} * (t = b \lor \underbrace{\circ \mathsf{Map} \ t \mapsto \Box \ L[t]}^{\gamma_{elt}}) \\ \forall k \in \mathsf{dom}(M).k \leq \begin{cases} t & \text{if } t < b \\ t - 1 & \text{otherwise} \end{cases}$$

$$\exists \gamma_{tb}, \gamma_{elt}.$$

$$\star \begin{cases} \gamma = (\gamma_{tb}, \gamma_{elt}) \\ \underbrace{\circ \mathsf{MN} \ tbs(t, b)}^{\gamma_{tb}} \\ t = b \lor \underbrace{\circ \mathsf{Map} \ t \mapsto \Box \ L[t]}^{\gamma_{elt}} \end{cases}$$

Figure 4.5: The definition of ghost deque state.

To maintain this information, we insert a new mapping  $t \mapsto L[t]$  whenever the top is incremented and it's still smaller than the bottom, or the bottom is incremented when the top and bottom were equal.

The full definition of the ghost deque state is given in Figure 4.5. The proof rules in Figure 4.2 are proven as follows:

- Dqst-Auth-Alloc: allocate  $[\bullet MN \ 2]^{\gamma_{tb}}$  by Mono-Nat-Alloc and  $[\bullet Map \ \emptyset]^{\gamma_{elt}}$  by Ghost-Map-Alloc. The other propositions are held automatically.
- Dqst-Frag-Persistent: follows from Mono-Nat-Lb-Persistent, Ghost-Map-Elem-Persistent, and the persistence of pure propositions.
- $\bullet$  Dqst-Frag-Get: use Mono-Nat-Lb-Get.
- Dqst-Frag-Valid: use Mono-Nat-Lb-Valid to prove  $t_1 \leq t_2 \wedge (t_1 = t_2 \wedge t_1 < b_1 \implies t_2 < b_2)$ . Next, given  $t_1 < b_1$  and  $t_2 < b_2$ , we have  $\lceil \circ \mathsf{Map} \ t_1 \mapsto \sqcap L_1[t_1] \rceil^{\gamma_{elt}} * \lceil \circ \mathsf{Map} \ t_2 \mapsto \sqcap L_2[t_2] \rceil^{\gamma_{elt}}$ , so Ghost-Map-Elem-Agree proves  $L_1[t_1] = L_2[t_2]$ .
- Dqst-Write-Array: If t = b, trivial. Otherwise,  $L[t] = L[b \leftarrow v][t]$  since  $t \not\equiv b \pmod{|L|}$ .
- $\bullet \ \, \text{Dqst-Push: update} \, \big[ \bullet \mathsf{MN} \, tbs(t,b) \big]^{\gamma_{tb}} \, \, \text{to} \, \big[ \bullet \mathsf{MN} \, tbs(t,b+1) \big]^{\gamma_{tb}} \, \, \text{by Mono-Nat-Auth-Update. If} \, t = b, \\ \text{also update} \, \big[ \bullet \mathsf{Map} \, M \big]^{\gamma_{elt}} \, \, \text{to} \, \big[ \bullet \mathsf{Map} \, M \big] t \leftarrow L[t] \big]^{\gamma_{elt}} \, \, \text{by Ghost-Map-Insert and obtain} \, \big[ \circ \mathsf{Map} \, t \mapsto^{\square} L[t] \big]^{\gamma_{elt}}.$
- Dqst-Pop: trivial.
- Dqst-CAS-Top: update  $[\bullet MN \ tbs(t,b)]^{\gamma_{tb}}$  to  $[\bullet MN \ tbs(t+1,b)]^{\gamma_{tb}}$  by Mono-Nat-Auth-Update. If t+1 < b, also update  $[\bullet Map \ M]^{\gamma_{elt}}$  to  $[\bullet Map \ M[t+1 \leftarrow L[t+1]]]^{\gamma_{elt}}$  by Ghost-Map-Insert and obtain  $[\circ Map \ t+1 \mapsto \Box \ L[t+1]]^{\gamma_{elt}}$ .

## Chapter 5. Full Verification of Chase-Lev Deque

In this chapter, we extend the verification in chapter 4 to the full version, where the array resizes on overflow. We give a specification for resizing the array, extend the invariant and other resource definitions to account for array replacement, and list the changes in the verification of each operation.

## 5.1 Specification of Resizing

Recall from chapter 2 that grow (2.3) takes a circular array circle, top t, and bottom b, and returns a new circular array circle' such that circle[t..b] = circle'[t..b]. The implementation is skipped in the paper, but on the high-level it operates as follows: we first read the size n of circle (the size is given in circle along with the array) and allocate a new circular array circle' with size 2n. Then for each index  $i \in [t..b)$ , we read the index i in the array of circle, and copy the value to the index i in the array of circle'. The new circle' is then returned.

The correctness of grow relies on the fact that the array is used read-only. If the contents of the array change while copying a range, the new array may not make much sense. This assumption indeed holds because only the sole owner of the deque calls this function. Therefore, all we need is a fractional points-to for the array. But since the owner keeps track a half points-to locally, we do not need to open any invariant around it. We can thus give the specification of grow as the following regular Hoare triple:

$$t \leq b < t + |L| \implies \left\{ arr \mapsto^{1/2} L \right\} grow((arr, |L|), t, b) \left\{ (arr', |L'|). \begin{array}{l} |L| < |L'| \wedge L[t..b) = L'[t..b) * \\ arr \mapsto^{1/2} L * arr' \mapsto L' \end{array} \right\}$$

The proof is a straightforward induction on b-t, so we omit the details here.

## 5.2 New Deque State and Invariant

Since we are removing the assumption that the array is never replaced, the resource definitions should change as well. Specifically,  $C \mapsto^{\square} \cdots$  no longer works because C can be overwritten by resizing the array. Therefore we change them to  $C \mapsto^{1/2} \cdots$  for both Dequelnv and OwnDeque.

If we attempt the verification with these modified resource definitions, we still encounter a problem at steal because we don't keep track of the old arrays. When reading the array, we open the invariant to get  $C \mapsto^{1/2} (arr, n)$ , read (arr, n) from it, and close the invariant. Now all resources about arr are lost: there is no local resource to synchronize because stealers do not have OwnDeque. When opening the invariant again, we get a different array arr' as its existential. As a result, we get  $arr' \mapsto \cdots$  even though we have to read from arr.

To receive the points-to for arr at that point, the invariant should maintain not only the current array but also all the arrays that have been used so far. Informally, divide the timeline of operations as eras, where a new era starts whenever the array is replaced. Then we extend the ghost deque state to contain the archive of the information about all the past arrays: the array at the moment of archival along with its length, the points-to to that array, and the value of tbs(t,b) at the moment of archival. Each era is represented by a ghost name  $\gamma_{era}$  since we eventually have to tie a ghost state with each era.

DQST-ARCHIVED-FRAG-GET 
$$\stackrel{\circ}{\otimes} (\gamma, arr, L, t, b) \twoheadrightarrow \stackrel{\circ}{\otimes} (\gamma, arr, L, t, b)$$
 DQST-ARCHIVED-FRAG-VALID 
$$\stackrel{\circ}{\otimes} (\gamma, arr, L_1, t_1, b_1) * \stackrel{\circ}{\otimes} (\gamma, arr, L_2, t_2, b_2) \twoheadrightarrow t_1 \leq t_2 \wedge \mathsf{Preservation}_{1,2}$$
 DQST-FRAG-AGREE 
$$\stackrel{\circ}{\otimes} (\gamma, arr_1, L_1, t_1, b_1) * \stackrel{\circ}{\otimes} (\gamma, arr_2, L_2, t_2, b_2) \twoheadrightarrow arr_1 = arr_2 \wedge |L_1| = |L_2|$$
 DQST-ARCHIVED-GET 
$$\gamma_1 \neq \gamma_2 \wedge \stackrel{\circ}{\otimes} (\gamma_1, arr_1, L_1, t_1, b_1) * \stackrel{\bullet}{\otimes} (\gamma_2, arr_2, L_2, t_2, b_2)$$
 
$$\twoheadrightarrow \exists L, t, b. \implies \begin{cases} \stackrel{\circ}{\otimes} (\gamma_1, arr_1, L, t, b) * arr_1 \mapsto L \\ \stackrel{\circ}{\otimes} (\gamma_1, arr_1, L, t, b) * arr_1 \mapsto L \end{cases}$$
 DQST-ARCHIVE 
$$|L| \leq |L'| \wedge L[t..b) = L'[t..b) * arr \mapsto L$$
 
$$\twoheadrightarrow (\gamma, arr, L, t, b) \implies \exists \gamma'. \stackrel{\bullet}{\otimes} (\gamma', arr', L', t, b)$$

Figure 5.1: Additional proof rules of deque state with eras.

The information at the moment of archival poses a problem if we use  $\stackrel{\triangle}{=}$ , because it could mean any point of execution, not just the moment of archival. We cannot use  $\stackrel{\bullet}{=}$  either because the stealer cannot access the  $\stackrel{\bullet}{=}$  at the moment of archival; only the owner can. Instead, we introduce another form of deque state, representing the information precisely at archival, which can be accessed from  $\stackrel{\bullet}{=}$  of a future era.

Now there are three forms of deque state:  $(\gamma_{era}, arr, L, t, b)^{\gamma}$  and  $(\gamma_{era}, arr, L, t, b)^{\gamma}$  representing the full ownership and persistent knowledge of a deque state respectively, and  $(\gamma_{era}, arr, L, t, b)^{\gamma}$  representing the ownership of a deque state of a past era. We will call it the archived deque state. Since we are going to put a points-to to the array in for each past era, these resources are not technically ghost states anymore; thus we will not draw dotted lines around it.

All of the proof rules in Figure 4.2 still apply to our extended deque states, with some straightforward modifications to include the era and the circular array pointer. Dqst-Auth-Alloc may be less trivial to modify: the first era,  $\gamma_0$ , is made from the deque state allocation, i.e.  $\exists \gamma, \gamma_0$ .  $(\gamma_0, arr, L, 1, 1)^{\gamma}$ .

In addition, we use more proof rules given in Figure 5.1. Dqst-Archived-Frag-Get and Dqst-Archived-Frag-Valid are similar to Dqst-Frag-Get and Dqst-Frag-Valid but from  $\stackrel{\bullet}{\cong}$ . Dqst-Frag-Agree shows that the array pointer and the length of the array does not change within an era. Dqst-Archived-Get lets us access an  $\stackrel{\bullet}{\cong}$  of a past era from the  $\stackrel{\bullet}{\cong}$ . This has a pattern of  $P \twoheadrightarrow (Q * Q \twoheadrightarrow P)$ : this means that we have an access to Q "inside" P, and we can return this Q to get P back. Finally, Dqst-Archive lets us replace the array and advance the era, as long as the circular slice of [t..b) is preserved.

Now we change Dequelnv and OwnDeque as follows. Not much is different from the definition given in chapter 2. The differences are: (1) the ghost variable for  $\gamma_{sw}$  now contains  $\gamma_{era}$  and arr as well; (2) has additional parameters as defined above; and (3)  $C \mapsto \cdots$  is no longer persistent but only fractional.

$$\exists \gamma_q, \gamma_{sw}, \gamma_{state}, \gamma_{era}, C, arr, top, bot, arr, L, t, b, pop.$$
 
$$\begin{cases} \gamma = (\gamma_q, \gamma_{sw}, \gamma_{state}) \land p = (C, top, bot) \\ 1 \leq t \leq b < t + |L| \\ C \mapsto^{1/2} (arr, |L|) \\ \vdots \gamma_q \mapsto^{1/2} L[t..b] \mid * [\gamma_{sw} \mapsto^{1/2} (\gamma_{era}, arr, L, b, pop)] \\ \vdots (\gamma_{era}, arr, L, t, b)^{\gamma_{state}} \\ arr \mapsto^{1/2} L * top \mapsto t \\ bot \mapsto^{1/2} \begin{cases} b - 1 & \text{if } pop = true \\ b & \text{otherwise} \end{cases} \\ \exists \gamma_q, \gamma_{sw}, \gamma_{state}, \gamma_{era}, C, top, bot, arr, L, b. \\ \begin{cases} \gamma = (\gamma_q, \gamma_{sw}, \gamma_{state}) \land p = (C, top, bot) \\ C \mapsto^{1/2} (arr, |L|) \\ \vdots \gamma_{sw} \mapsto^{1/2} (\gamma_{era}, arr, L, b, \text{False}) \\ arr \mapsto^{1/2} L * bot \mapsto^{1/2} b \end{cases}$$

#### 5.3 Changes in the Verification

Now we verify each operation again. Verifying new\_deque has no interesting difference; it's still just a matter of allocating resources. Verifying pop is not much different either; no resizing takes place here. The ones that need extra proof work are push because we have to resize the array, and steal because we have to consider the case where the array is replaced by the owner.

#### 5.3.1 Push

The main difference in the verification of push is the branch where  $t + sz \le b + 1$ , in which case we resize the array. We proceed as follows:

- grow\_circle "circle" "t" "b": Apply the specification of grow with  $arr \mapsto^{1/2} L$  to step through it and obtain a new points-to  $arr_{new} \mapsto L_{new}$  such that  $L[t..b) = L_{new}[t..b)$ .
- arr "deque" <- ...: This is the part where we resize the array.
  - (1) Open Dequelnv and use Ghost-Var-Agree to prove  $(\gamma'_{era}, arr', L', b', pop') = (\gamma_{era}, arr, L, b, False)$ .
  - (2) Combine two halves of  $arr \mapsto^{1/2} L$  into  $arr \mapsto L$ , then use Dqst-Archive to consume it and update  $(\gamma_{era}, arr, L, t, b)^{\gamma_{dqst}}$  to  $(\gamma_{era-new}, arr_{new}, L_{new}, t, b)^{\gamma_{dqst}}$ .
  - (3) Update the ghost variable for  $\gamma_{sw}$  to  $[\gamma_{\underline{sw}} \Rightarrow^{1/2} (\gamma_{\underline{era}-\underline{new}}, \underline{arr_{\underline{new}}}, \underline{L_{\underline{new}}}, \underline{b}, \mathsf{False})].$
  - (4) Use Dqst-Frag-Get to get  $\overset{\circ}{\Box}$ .
  - (5) Combine  $C \mapsto^{1/2} \cdots$ , apply Store, split it back, and close Dequelnv.
- The rest is the same as the case without resizing, but uses  $\gamma_{era-new}$ ,  $arr_{new}$ , and  $L_{new}$ .

#### 5.3.2 Steal

In the full verification of steal, we open the invariant five times: (1) reading the top, (2) reading the bottom, (3) loading the array, (4) reading the value at the top, and (5) CAS-ing the top. Compared to chapter 4, there is an extra opening due to (3): loading arr from  $C \mapsto^{1/2} \cdots$  requires opening the

invariant since we cannot duplicate it and keep it locally. Again, let  $\gamma_{era-i}$ ,  $arr_i$ ,  $t_i$ ,  $b_i$ , and  $L_i$  be the era, array pointer, top, bottom, and array, respectively, when Dequelnv is opened for the *i*-th time.

Apart from that, the main difference in verification happens in (4). We open the invariant which gives  $arr_4 \mapsto^{1/2} L_4$ , but we want to read a value from the array we got in (3), so what we actually need is  $arr_3 \mapsto^{1/2} \cdots$ . We resolve this problem by a case analysis on whether the array was replaced in the meantime, i.e.  $\gamma_{era-3} = \gamma_{era-4}$ . If they are the same era, we use Dqst-Frag-Agree to prove  $arr_3 = arr_4$ , and the rest is similar to chapter 4. Otherwise,  $arr_3$  is a past array, so we retrieve a points-to for it from using Dqst-Archived-Get. Then later, we use Dqst-Archived-Frag-Valid to prove that the top has not increased if CAS succeeded.

The following is the proof detail for the case  $\gamma_{era-3} \neq \gamma_{era-4}$ :

- We are currently at "v" := !(access (Fst "circle") "t" "sz"). So far, we opened Dequelnv for the 4th time, took a snapshot for each era, and know  $t_1 \le t_2 \le t_3 \land \mathsf{Preservation}_{2,3} \land \gamma_{era-3} \ne \gamma_{era-4}$ .
  - (1) Use Dqst-Archived-Get to get  $(\gamma_{era-3}, arr_3, L_A, t_A, b_A) * arr_3 \mapsto L'$ .
  - (2) Use Dqst-Archived-Frag-Valid to prove  $t_3 \leq t_A \wedge \mathsf{Preservation}_{3,A}$ .
  - (3) Use Dqst-Archived-Frag-Get to get  $\stackrel{\triangle}{\simeq} (\gamma_{era-3}, arr_3, L_A, t_A, b_A)$ .
  - (4) Use Dqst-Frag-Agree to prove  $|L_3| = |L'|$ , and thus the value "sz" we read for  $L_3$  can still be used to read from L'.
  - (5) Read  $v := L'[t_1]$  from  $arr_3 \mapsto L'$ .
  - (6) Return  $\stackrel{\bullet}{\cong}$  back to  $\stackrel{\bullet}{\cong}$ .
  - (7) Use Dqst-Frag-Valid to prove  $t_A \leq t_4 \wedge \mathsf{Preservation}_{A,4}$ .
  - (8) Close Dequelnv.
- CAS (top "deque") "t" ("t" + #1): open Dequelnv, use  $^{\mbox{\ensuremath{$\circ$}}}$  to prove  $t_4 \leq t_5$  by Dqst-Frag-Valid, and attempt CAS with  $top \mapsto t_5$ .
- If  $t_1 = t_5$ , we succeed the CAS and we get  $top \mapsto t_1 + 1$ . Similarly to chapter 4, prove  $t_1 = t_2 = t_3 = t_4 = t_5$ ,  $t_2 < b_2$ ,  $t_3 < b_3$ ,  $\cdots$ ,  $t_5 < b_5$ , and  $L_5[t_5] = v$ . Open AU, update the ghost states, commit, and close Dequelnv.
- If  $t_1 \neq t_5$ , we fail the CAS. Open AU, immediately commit, and close Dequelnv.
- End up with  $\Phi$  and finish the proof.

## 5.4 Construction of the Full Deque State

To extend the definition of deque state in chapter 4 to support resizing, we need two key changes: (1) we need  $\stackrel{\bullet}{\cong}$  to represent the last moment of an era. In particular, if we have both  $\stackrel{\triangle}{\cong}$  and  $\stackrel{\bullet}{\cong}$  for the same era, we should be able to prove that  $\stackrel{\bullet}{\cong}$  was made later than  $\stackrel{\triangle}{\cong}$ . (2)  $\stackrel{\bullet}{\cong}$  and the corresponding points-to to the array should be stored in  $\stackrel{\bullet}{\cong}$  for each era.

The full definition of deque state is given in Figure 5.2. To represent (1), the deque state has an additional copy of the monotonic natural number identified by  $\gamma_{era}$ ; this is the purpose of representing each era as a ghost name. The numbers for  $\gamma_{tb}$  and  $\gamma_{era}$  stay identical in each era, but when we replace the array, we put the latter one into  $\stackrel{\triangle}{\cong}$  and allocate a new monotonic natural number. To represent (2), the deque state also tracks an additional ghost map identified by  $\gamma_{room}$ . This map maps each era to the array pointer and the length of the array, representing the fact that they do not change within each

$$\exists \gamma_{tb}, \gamma_{elt}, \gamma_{room}, M_t, M_e.$$

$$\gamma = (\gamma_{tb}, \gamma_{elt}, \gamma_{room}) \begin{bmatrix} \bullet \mathsf{MN} \ tbs(t,b) \end{bmatrix}^{\gamma_{tb}} * \begin{bmatrix} \bullet \mathsf{MN} \ tbs(t,b) \end{bmatrix}^{\gamma_{era}} \\ [\bullet \mathsf{MN} \ tbs(t,b) ]^{\gamma_{tb}} * \begin{bmatrix} \bullet \mathsf{MN} \ tbs(t,b) \end{bmatrix}^{\gamma_{era}} \end{bmatrix}$$

$$\forall k \in \mathsf{dom}(M_t).k \leq \begin{cases} t & \text{if } t < b \\ t-1 & \text{otherwise} \end{cases}$$

$$\gamma_{era} \notin \mathsf{dom}(M_e) \\ [\bullet \mathsf{Map} \ M_e [\gamma_{era} \leftarrow (arr, [L])]]^{\gamma_{room}} * \begin{bmatrix} \bullet \mathsf{Map} \ \gamma_{era} \mapsto \Box (arr, [L]) \end{bmatrix}^{\gamma_{room}} \\ * (\gamma_{room}) \in \mathsf{Map} \mathsf{Me} \exists L', t', b' : (\gamma', arr', L', t', b')^{\gamma} * arr' \mapsto L' \end{cases}$$

$$\exists \gamma_{tb}, \gamma_{elt}, \gamma_{room}.$$

$$\exists \gamma_{tb}, \gamma_{elt}, \gamma_{room} \\ [\bullet \mathsf{MN} \ tbs(t,b)]^{\gamma_{tb}} * \begin{bmatrix} \bullet \mathsf{MN} \ tbs(t,b) \end{bmatrix}^{\gamma_{era}} \\ t = b \vee [\mathsf{oMap} \ t \mapsto \Box \ L[t]]^{\gamma_{era}} \\ [\bullet \mathsf{Map} \ \gamma_{era} \mapsto \Box (arr, [L])]^{\gamma_{room}} \end{cases}$$

$$\Rightarrow (\gamma_{era}, arr, L, t, b)^{\gamma} := [\bullet \mathsf{MN} \ tbs(t,b)]^{\gamma_{era}} * \overset{\overset{\circ}{\triangle}}{\triangle} (\gamma_{era}, arr, L, t, b)^{\gamma}$$

Figure 5.2: The definition of full deque state.

array. Furthermore, for each mapping except for the current era, \*\* stores an \*\* and the points-to to the array.

The proof of each rule in Figure 4.2 is almost the same, and the additional rules in Figure 5.1 are proven as follows:

- Dqst-Archived-Frag-Get: trivial.
- Dqst-Archived-Frag-Valid: use Mono-Nat-Lb-Valid and Ghost-Map-Elem-Agree, similarly to how we proved Dqst-Frag-Valid.
- Dqst-Frag-Agree: use Ghost-Map-Elem-Agree.
- Dqst-Archived-Get: use Ghost-Map-Lookup to prove  $M_e[\gamma_2 \leftarrow (arr, |L|)][\gamma_1] = (arr_1, |L_1|)$ . Since  $\gamma_1 \neq \gamma_2$ , we get  $(\gamma_1 \mapsto (arr_1, |L_1|)) \in M_e$ . Use it to access  $(\gamma_1, arr_1, L, t, b) * arr_1 \mapsto L$  from the separating conjunction  $*_{(\gamma' \mapsto (arr', n')) \in M_e}$  of  $^{\bullet}$ .
- Dost-Archive:

  - (1) Prove  $(t = b \vee [\circ \mathsf{Map}\ t \mapsto \Box\ L[t]]^{\gamma_{elt}})$  is preserved using L[t..b) = L'[t..b). (2) Take a snapshot  $\triangle(\gamma, arr, L, t, b)$  by Dqst-Frag-Get and combine with  $[\bullet \mathsf{MN}\ tbs(t, b)]^{\gamma}$  to get  $\mathfrak{B}(\gamma, arr, L, t, b)$ .
  - (3) Allocate a new  $\left[\bullet MN \ tbs(t,b)\right]^{\gamma'}$  for a new ghost name  $\gamma'$  by a stronger variant of Mono-Nat-Alloc: when making a new ghost name  $\gamma'$ , we can make sure that  $\gamma'$  is different from all ghost names in some finite set. In our case, we ensure  $\gamma' \notin dom(M_e) \cup \{\gamma\}$ .
  - (4) Update the authoritative ghost map to  $[\bullet Map M_e[\gamma \leftarrow (arr, |L|)][\gamma' \leftarrow (arr', |L'|)]]^{\gamma_{room}}$  by Ghost-Map-Insert and obtain  $[\circ Map \ \gamma' \mapsto \Box (arr', |L'|)]^{\gamma_{room}}$ .

(5) Prove the separating conjunction  $*_{(\gamma'' \mapsto (arr'', n'')) \in M_e[\gamma \leftarrow (arr, |L|)]}$  of the new  $^{\bullet}$  by combining the prior \* with  $^{\bullet}(\gamma, arr, L, t, b)$ .

## Chapter 6. Verification in Extended Settings

## 6.1 Verification with Safe Memory Reclamation (SMR)

In this section, we introduce a modular verification framework for memory reclamation schemes [21] and verify the Chase-Lev deque with memory reclamation.

### 6.1.1 Background: Memory Reclamation

To prevent memory leak in the programs manipulating memory, unused parts of the memory must be reclaimed. In the case of the Chase-Lev deque, after resizing the array, the older array must be deallocated. However, it should not be deallocated *right away*: stealers might still be reading from the older array, resulting in a use-after-free bug. Deallocating is safe only after all stealers having access to the older array finish accessing it.

This can be resolved by keeping track of stealers for each array, so that the last stealer deallocates it. Of course, this method requires a large amount of additional code and verification. In the other direction, garbage collectors can be used to reclaim inaccessible memory automatically. However, it is unavailable in some low-level systems and languages, and incurs high performance overhead.

Various memory reclamation schemes [35, 34] have been proposed to strike a balance between simplicity and performance overhead. These schemes generally provide two APIs: (1) a function to *protect* a part of the memory to prevent it from being deallocated; and (2) a function to *retire* it so that it is deallocated when no threads are protecting it. This approach allows fine-grained memory control, and hides the detail of memory management at the same time.

Figure 6.1 shows the modification necessary to support hazard pointers in Chase-Lev deque. In push, after resizing and replacing the array, we retire the old array so that it can be deallocated when no stealers can access it. In steal, we protect the circular array before accessing it, and drop the protection after the access. To port the code to the SMR verification framework, there is also a slight modification to how a deque is represented: the fields are now arranged in memory layout instead of a tuple.

#### 6.1.2 Specification and Verification in SMR

To verify Chase-Lev deque under SMR schemes, the whole SMR scheme should be verified, and its specification should be incorporated into the verification of deque. The work by Jung et al. [21] fulfills this goal: it provides a modular specification for SMR in Iris. The specification is designed so that it can be seamlessly integrated into the data structures' verification without exposing the implementation details. The SMR schemes themselves have been verified in that work, so all we need is modify our Chase-Lev verification to use SMR specifications.

The specification of hazard pointers introduce three predicates: (1)  $\mathsf{Managed}(\ell, \gamma, P)$  representing the ownership of the pointer  $\ell \mapsto \cdots$  managed by hazard pointers; (2)  $\mathsf{HPSlot}(s)$  representing a shield s which can be used to protect a pointer; and (3)  $\mathsf{Protected}(s, \ell, \gamma, P)$  representing the protection of  $\ell \mapsto \cdots$  by a shield s. The predicate  $P(\ell, v, \gamma)$  is a per-location invariant used for reasoning with its contents; it is usually a ghost state with the ghost name  $\gamma$ .

```
Definition push :val := \lambda:"deque" "v",
 (if: "t" + "sz" \leq "b" + #1
   then let: "domain" := !("deque" + #qdom) in
     "deque" + #circle <- circle_grow "circle" "t" "b" "sz" ;;</pre>
     {\tt hazptr.(hazard\_domain\_retire)} \ "domain" \ "circle" \ ("sz" + \#1)
   else #()
 ) ;;
 . . .
Definition deque_steal :val := \lambda:"deque",
 let: "domain" := !("deque" + #qdom) in
 let: "shield" := hazptr.(shield_new) "domain" in
 let: "t" := !("deque" + #qtop) in
 let: "b" := !("deque" + #qbot) in
 let: "circle" := hazptr.(shield_protect) "shield" ("deque" + #circle) in
 else let: "v" := !(circ_access ("circle" + #carr) "t" "sz") in
 hazptr.(shield_drop) "shield" ;;
```

Figure 6.1: Changes in the implementation to support hazard pointers.

```
\begin{array}{c} \operatorname{Managed-New} & \operatorname{HP-Retire} \\ \ell \mapsto v * P(\ell, v, \gamma) \Longrightarrow \operatorname{Managed}(\ell, \gamma, P) & \{\operatorname{Managed}(\ell, \gamma, P)\} \, retire(\ell) \, \{\operatorname{True}\} \\ & \operatorname{Managed-Access} \\ & \{\operatorname{Managed}(\ell, \gamma, P)\} \, ! \ell \, \{v. \, P(\ell, v, \gamma) * \operatorname{Managed}(\ell, \gamma, P)\} \\ \\ \operatorname{Protectt} \\ \langle src \mapsto \ell * \operatorname{Managed}(\ell, \gamma, P) * \operatorname{HPSlot}(s) \rangle \, protect(s, src) \, \langle \ell. \, src \mapsto \ell * \operatorname{Managed}(\ell, \gamma, P) * \operatorname{Protected}(s, \ell, \gamma, P) \rangle \\ & \operatorname{Protected}(s, \ell, \gamma, P) \} \, ! \ell \, \{v. \, P(\ell, v, \gamma) * \operatorname{Protected}(s, \ell, \gamma, P)\} \end{array}
```

Figure 6.2: Selected proof rules of hazard pointers.

Figure 6.2 shows a few proof rules of hazard pointers. We can turn a points-to into a Managed, which can be used to read a value and is discarded by calling hazard\_domain\_retire. Managed also allows protecting the pointer with a shield HPSlot, which turns it into a Protected. This Protected can also be used to read a value, even after Managed is retired; the actual pointer is not deallocated until all protection to it is removed.

To combine the new SMR-related resources with Chase-Lev deque verification, we make the following changes from chapter 5:

- Since the fields of the deque are arranged in memory, we change tuples to lists, e.g. for (arr, |L|), we swap the two fields and concatenate them into [|L|] + L.
- We add  $\gamma_{hp}$  for each era, a ghost name for the ghost variable to manage hazard pointers. This is stored in  $M_e$  in addition to the array and its length.
- In Dequelnv and OwnDeque, we extend the ghost variable for  $\gamma_{era}$  to handle  $\gamma_{hp}$ .
- In Dequelnv, we replace  $C \mapsto^{1/2} [|L|] + L$  with Managed $(C, \gamma_{hp}, P) * P(C, L, \gamma_{hp})$ .
- In OwnDeque, we remove  $C \mapsto^{1/2} [|L|] + L$ . The owner can still keep track of the array contents L using the ghost variable for  $\gamma_{sw}$ .
- Now accessing the array requires Managed which is in Dequelnv, so we change the specification of grow to a LAT. The atomic precondition and postcondition are Managed $(C, \gamma_{hp}, P) * P(C, L, \gamma_{hp})$ , and the private postcondition is  $C' \mapsto [|L'|] + L'$ .

For verification, we just need to apply the specification for SMR functions, and read the array using Managed-Access and Protected-Access. When the owner wants to read the array, we open Dequelnv to get Managed( $C, \gamma_{hp}, P$ ) \*  $P(C, L, \gamma_{hp})$ . Then we apply Managed-Access which gives  $P(C, L', \gamma_{hp})$ , and use Ghost-Var-Agree on the two P resources to prove L = L'.

For the stealer, we obtain  $\mathsf{Protected}(s, C, \gamma_{hp}, P)$  when loading (and protecting) the array for the first time. Later, we read the array from  $\mathsf{Protected}$  and proceed again by case analysis on whether the era has advanced before CAS-ing. If not advanced, we follow the same procedure as above. Otherwise, we retrieve the  $\gamma_{hp}$  of the past era from  $^{\bullet \bullet}$  and use it to agree on the array.

## 6.2 Foundation for Verification in Relaxed Memory Model

In this section, we briefly discuss the verification of Chase-Lev deque in relaxed memory model. Due to the complexity of this memory model, we leave the full specification and verification to future work. Instead, we show how to verify the safety of Chase-Lev deque, which is expected to be required for full verification.

#### 6.2.1 Background: Relaxed Memory Model and iRC11

In relaxed memory model, instructions can be reordered as long as the overall behavior is preserved. For example, adjacent instructions  $X \leftarrow 1$  and  $Y \leftarrow 1$  can be swapped since they do not affect each

other. However, this kind of reordering allows more behaviors in the context of concurrency. Consider the following example, where all variables are initially 0 and two codes separated by a line are executed concurrently:

Unlike the SC memory model, it is possible to fail the assertion in the relaxed memory model: the code on the left is reordered and  $Y \leftarrow 1$  is executed, and then we enter the Y == 1 branch even though X == 1 does not hold.

Instead of directly accounting for reordering, we adopt the equivalent view-based semantics [28, 26, 14]. Here, the instructions are executed in order, but they may read past values. Specifically, each thread maintains their *views*, the read and write events it has observed so far. When reading a value from a location, the thread can read a write event for that location that it has not observed yet. Similarly, when writing to a location, the thread can insert a write event after the last one it has observed.

iRC11 [15, 14] is a separation logic for relaxed memory, formalized in Iris. It incorporates the idea of views into the logic by maintaining multiple events in each location. It introduces a points-to assertion of the form  $\ell \mapsto h$  where h is a *history*, a list of events at the location  $\ell$ , which can be used to read one of the events of h. An event is composed of a timestamp at which the event was made, a value written and a view tied into it. The timestamp is used to determine whether the thread can read the event. Upon reading an event, the thread gets the value as the result of reading and combines the view with the thread's own view.

To simplify the logic further, iRC11 supports the following access modes: (1) single-writer mode, in which only one thread can write to the location and all other threads can only read; (2) CAS-only mode, in which all threads can write to the location but only via CAS-ing; and (3) read-only mode, in which no threads can write to the location. As the interface of these modes, the following resources are defined:

- $\ell \mapsto_{\sf sw} h$ ,  $\ell \mapsto_{\sf cas} h$ , the points-to assertions with the history h, denoting that  $\ell$  is being used in the single-writer and CAS-only mode respectively;
- $\ell \mapsto_{\mathsf{ro}} n$ , the read-only points-to assertion with the value (not history) n;
- $\ell \supseteq_{sn} h$ , the history-seen observation, asserting that the thread has observed all write events in the history h;
- $\ell \supseteq_{sw} h$ , the *single-writer ownership*, asserting that the thread has an exclusive right to write to  $\ell$ , with the full history being h which has been completely observed:
- $\ell \supseteq_{\mathsf{ro}} n$ , the read-only observation, the right to read n from  $\ell \mapsto_{\mathsf{ro}} n$ .

Unlike the points-to in chapter 3, the points-to in iRC11's specialized access modes cannot be split. Instead of checking for the fractions to read or write, it additionally requires an assertion about observation. To read from or write to a location  $\ell$ , the thread must at least own  $\ell \mapsto_{\theta} h$ , where  $\theta$  is either sw or cas, and h is a history. In addition, for writing in the single-writer mode, the thread must own  $\ell \sqsubseteq_{sw} h$ . For the other operations (reading in any mode, or CAS-ing in the CAS-only mode), the thread must own  $\ell \sqsubseteq_{sn} h'$  for some history h'.

<sup>&</sup>lt;sup>1</sup>In order to ensure correctness for codes like this, a certain ordering can be enforced using release-acquire ordering, memory fences, and so on, but they are beyond the scope of this paper.

#### 6.2.2 Implementation and Safety Verification in iRC11

The implementation of Chase-Lev deque in relaxed memory model is almost the same as the SC model, except that we should appropriately insert memory fences and determine the ordering mode of each operation. However, the choice of fences and ordering does not affect safety, only functional correctness, so we skip the details here and just follow the implementation by Kang [27].

We eventually would like to verify a strong specification with Compass [16], but for now we only verify that each operation runs safely:

$$\left\{0 < n\right\} new\_deque(n) \left\{p. \exists \gamma. \overline{\mathsf{DequeInv}^{\gamma}(p)} * \mathsf{OwnDeque}^{\gamma}(p) * \mathsf{DequeLocal}^{\gamma}(p)\right\}$$
 
$$\overline{\mathsf{DequeInv}(p)} \vdash \mathsf{DequeLocal} \twoheadrightarrow \left\{\mathsf{OwnDeque}(p)\right\} push(p,v) \left\{\mathsf{OwnDeque}(p)\right\}$$
 
$$\overline{\mathsf{DequeInv}(p)} \vdash \mathsf{DequeLocal} \twoheadrightarrow \left\{\mathsf{OwnDeque}(p)\right\} pop(p) \left\{\_.\mathsf{OwnDeque}(p)\right\}$$
 
$$\overline{\mathsf{DequeInv}(p)} \vdash \mathsf{DequeLocal} \twoheadrightarrow \left\{\mathsf{True}\right\} steal(p) \left\{\_.\mathsf{True}\right\}$$

Here, DequeLocal is a persistent resource each thread maintains locally. It asserts that the thread has observed  $(\supseteq_{sn})$  some history for the top, bottom, and array, so that it can read from the corresponding points-to.

To verify the specifications above, we put points-to, history observation, and ownership into Dequelnv, OwnDeque, DequeLocal appropriately. This time, each era is represented by the timestamp  $\mathscr T$  at which the array was written to the array pointer. We maintain two ghost maps: persistent one for array pointer and the length of the array, and non-persistent one for the list H of histories for the array, i.e. the list for which each H[i] is the history for the i-th slot of the array. These ghost maps are used for synchronization between Dequelnv and OwnDeque.

Now we define the resources necessary for the specification. First, Dequelnv stores  $\mapsto_{sw}$  for the array and bottom,  $\mapsto_{cas}$  for the top, and  $\mapsto_{sw}$  and  $\sqsubseteq_{sn}$  for each of the archived array:

$$\mathsf{DequeInv}^{\gamma}(p) := \begin{array}{l} \exists \gamma_{era}, \gamma_{arr}, C, top, bot, h_t, h_b, h_C. \\ \\ \bigstar \left\{ \begin{array}{l} \gamma = (\gamma_{era}, \gamma_{arr}) \land p = (C, top, bot) \\ C \mapsto_{\mathsf{sw}} h_C \land top \mapsto_{\mathsf{cas}} h_t \land bot \mapsto_{\mathsf{sw}} h_b \\ \\ \bigstar_{\mathscr{T} \mapsto (p_{\mathscr{T}}, V_{\mathscr{T}})} \mathsf{CircleInv}(\mathscr{T}, p_{\mathscr{T}}, V_{\mathscr{T}}, \gamma_{era}, \gamma_{arr}) \end{array} \right.$$

$$\mathsf{CircleInv}(\mathcal{T}, p, V, \gamma_{era}, \gamma_{arr}) := \\ \\ \star \left\{ \begin{array}{l} p = (arr, |H|) \wedge |H| > 0 \\ \\ [ \circ \mathsf{Map} \ \mathcal{T} \mapsto^{\square} (p, |H|) ]^{\gamma_{era}} * [ \circ \mathsf{Map} \ \mathcal{T} \mapsto^{1/2} H ]^{\gamma_{arr}} \\ \\ arr \mapsto_{\mathsf{sw}} H \\ \\ \\ \star_{arr[i]=h} \ \exists h'.h' \subseteq h * arr \ \exists_{\mathsf{sn}} \ h' \end{array} \right.$$

Next, OwnDeque stores  $\supseteq_{sw}$  for the array, bottom, and each slot of the current array. It also asserts that all values ever written to the bottom are larger than 0, and the current array is the last one written to the history of the array pointer:

$$\exists \mathcal{J}_{last}, H_{last}, M_e, M_a, \gamma_{era}, \gamma_{arr}, C, top, bot, arr, h_b, h_C.$$

$$\begin{cases} \gamma = (\gamma_{era}, \gamma_{arr}) \land p = (C, top, bot) \\ \operatorname{dom}(M_e) = \operatorname{dom}(M_a) = \operatorname{dom}(h_C) \\ bot \sqsupseteq_{\operatorname{sw}} h_b * C \sqsupseteq_{\operatorname{sw}} h_C \\ \\ *_{\mathcal{J} \mapsto (b, \cdot) \in h_b} b \ge 1 \\ \mathcal{J}_{last} = \max(\operatorname{dom}(h_C)) \land h_C[\mathcal{J}_{last}] = (arr, \cdot) \\ \\ *_{arr[i] = h} arr[i] \sqsupseteq_{\operatorname{sw}} h \\ \boxed{\bullet \operatorname{Map} M_e} \qquad \qquad \bullet \text{Map} M_a \qquad \qquad \bullet \text{Map} \mathcal{J}^{\gamma_{arr}} \\ \boxed{\bullet \operatorname{Map} \mathcal{J} \mapsto \square (arr, \lfloor H_{last} \rfloor)}^{\gamma_{era}} * \boxed{\bullet \operatorname{Map} \mathcal{J} \mapsto 1/2 H_{last}}^{\gamma_{arr}} \end{cases}$$

Finally, DequeLocal stores  $\supseteq_{sn}$  for the array, top, and bottom:

$$\mathsf{DequeLocal}^{\gamma}(p) := \begin{array}{l} \exists \gamma_{era}, \gamma_{arr}, h_t, h_b, h_C. \\ \\ \bigstar \left\{ \begin{array}{l} \gamma = (\gamma_{era}, \gamma_{arr}) \land p = (C, top, bot) \\ top \; \beth_{\mathsf{sn}} \; h_t * bot \; \beth_{\mathsf{sn}} \; h_b * C \; \beth_{\mathsf{sn}} \; h_C \end{array} \right. \end{array}$$

The verification of each operation is not very interesting: it is just a matter of opening the invariant, using Ghost-Var-Agree to unify values if necessary, using  $\mapsto$  and  $\supseteq_{sn}$  or  $\supseteq_{sw}$  to read and write, and closing the invariant. However, the formalized proof is still very long. For comparison: the Coq formalization of the full verification in chapter 5 involves complicated reasoning about synchronization, the definition of deque state resources (e.g.  $\stackrel{\bullet}{w}$ ), all proof rules for the deque state along with the proof of each rule. Despite that, it is only slightly longer than the safety proof in iRC11 which skips all synchronization and doesn't even define the deque state resources.

## Chapter 7. Conclusion

## 7.1 Summary

We have formally verified Chase-Lev deque using the Iris separation logic. This is the first known verification of the Chase-Lev deque that is foundational, uses a realistic and unbounded implementation, and verifies a strong specification. We also extended the verification to incorporate safe memory reclamation techniques, and established a basis for verifying the deque in the relaxed memory model.

#### 7.2 Related Work

While various papers have introduced new work-stealing deque designs, most of them lack formal proofs or rely solely on pen-and-paper proofs to establish their correctness. Implementations without formal verification pose a risk of containing errors, even after an extensive testing, as evidenced by bug reports in commercial softwares. Chase-Lev deque is no exception to this issue. Lê et al. [33] proposed ARM and C11 implementations of Chase-Lev deque in relaxed memory model, and proved the ARM implementation correct. However, the C11 implementation lacked a formal proof and was later discovered to have a bug [40]. Also, although a proof can increase confidence in correctness to some degree, pen-and-paper proofs are still prone to human mistake and require thorough review. We can further instill confidence by checking the proof computationally, including our work.

However, prior works on mechanized verification of work-stealing deques have limitations like weaker specification, restrictive or unrealistic implementations, and larger trusted computing base, compared to our work. Here, we list such prior works and outline their limitations.

Aghai [7] verified the linearizability of the ABP work-stealing deque [8] using model checking techniques. However, they made the simplifying assumption of an infinitely large array, which is not realistic in practice. Moreover, the ABP deque itself has an inherent limitation due to its bounded capacity; in fact, Chase-Lev deque was specifically designed to address this limitation [13]. Also, their verification method was not fully mechanized. Instead of directly encoding linearizability into model checking, they checked a few basic properties and proved on pen-and-paper that they together imply linearizability.

In a later work, Kokologiannakis et al. [30] developed GenMC, a model checker for C programs under configurable memory models. As a part of the benchmark, they verified Chase-Lev deque in relaxed memory model. However, their implementation of the deque has a bounded capacity. Although it uses a circular array, it does not dynamically resize, so the owner thread fails to push its tasks when the array is full. Also, the verification targeted a weaker specification instead of linearizability [3].

Wang et al. [45] proposed a novel block-based work-stealing deque, and verified its correctness using GenMC. Similarly to the above work, they did not verify strong specifications, but only weaker guarantees like each element being popped or stolen only once.

Mutluergil and Tasiran [39] took a different approach to verify Chase-Lev deque by using layered refinement. This approach consists of multiple layers of implementation. Starting from the target implementation, the code is gradually transformed into a simpler versions that refine the previous ones. Eventually each operation becomes physically atomic, at which point logical atomicity of the target implementation is proven since it is refined by its physically atomic form. However, their implementation

assumed an infinitely large array which is again unrealistic. This also led to skipping the resizing procedure, which would simplify the synchronization reasoning compared to the real implementation.

Finally, while the aforementioned works have benefits of automation, they do not achieve foundational verification. The correctness of the whole verification process relies on trusting the verification tools themselves. For instance, although GenMC [30] has been formalized and proven to be sound and complete, the proof is pen-and-paper and the correctness of its C++ implementation is not guaranteed either. Similarly, the work by Mutluergil and Tasiran [39] uses the Civl verifier [31], which is a complex proof system consisting of multiple steps: it is built upon another verifier that incorporates an SMT solver, and Civl introduces its own processor as well.

Outside of work-stealing deques, there is another example of a foundationally verified scheduling queue. Krogh-Jespersen et al. [32] verified the Dartino queue, a scheduling data structure used in Google's Dartino virtual machine, in the Iris separation logic. As Dartino queue is implemented as a lock-based linked list, it has a simpler synchronization reasoning compared to Chase-Lev deque. The invariant of the Dartino queue mainly consists of standard properties related to node ownership.

### 7.3 Future Work

Verifying the full specification of Chase-Lev deque in the relaxed memory model poses two challenges. First, Chase-Lev deque utilizes SC memory fences, which are currently not supported by iRC11. Second, unlike the SC memory model, there is no widely accepted strong specification for data structures in the relaxed memory model [16]. Simple linearizability and logical atomicity may not be suitable for some relaxed-memory data structures such as Herlihy-Wing queue [20] and exchanger [19], because instructions in the relaxed memory model may only be synchronized with a subset of other instructions. The difficulty of designing a strong specification further complicates the verification.

To address the first point, we plan to extend iRC11 to support SC memory fences, in a way that the synchronization guarantees achieved by the fences are adequately modeled. Regarding the second point, Dang et al. [16] developed Compass, a framework for strong specifications of relaxed memory data structures built on top of Iris. It offers comprehensive support for functional correctness properties, including synchronization and FIFO guarantee for queues. Moreover, it enables modular client reasoning for these data structures. However, the specification and proof are long and complicated, in part due to the inherent complexity of the relaxed memory. To address this gap, we are working on designing a simpler interface to faciliate easier verification. In addition, we are exploring the extension of Diaframe [38], an Iris proof automation tool, to automate parts of the verification process for the relaxed memory model.

Furthermore, we plan to extend the verification of safe memory reclamation schemes [21] to the relaxed memory model. Combined with the Compass framework, we aim to achieve verified strong specifications of various data structures, including Chase-Lev deque.

In the other direction, it would be interesting to verify more recent designs of work-stealing deques. Various techniques have been proposed and demonstrated to improve synchronization overheads, such as private deques [6, 43], architecture-aware optimizations [37], relaxation of stealing guarantees [36, 12], or block-based task grouping [45]. Verifying these data structures would require more sophisticated reasoning and invariants, especially considering the novel techniques they employ.

Furthermore, the verification of work-stealing deques can be extended to verify schedulers that make use of these deques. While there have been extensive research efforts in the area of verified schedulers and

operating systems [17, 29], to the best of our knowledge, support for work-stealing scheduling schemes is not yet present. It would be intriguing to explore how the whole work-stealing strategy can be verified using the specifications for work-stealing deques.

## Bibliography

- [1] The coq proof assistant. https://coq.inria.fr/.
- [2] Crossbeam: Tools for concurrent programming in rust. https://github.com/crossbeam-rs/crossbeam. Accessed: 2023-04-19.
- [3] Genmc: Generic model checker for concurrent c programs. https://github.com/MPI-SWS/genmc. Accessed: 2023-04-14.
- [4] Iris project. https://iris-project.org/.
- [5] Taskflow: A general-purpose parallel and heterogeneous task programming system. https://github.com/taskflow/taskflow. Accessed: 2023-04-19.
- [6] U. Acar, A. Chargueraud, and M. Rainey. Scheduling parallel programs by work stealing with private deques. volume 48, pages 219–228, 02 2013. doi: 10.1145/2442516.2442538.
- [7] M. K. Aghai. Verification of work-stealing deque implementation. Master's thesis, Uppsala University, Mar. 2012.
- [8] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '98, page 119–129, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 0897919890. doi: 10.1145/277651.277678. URL https://doi.org/10.1145/277651.277678.
- [9] L. Birkedal and A. Bizjak. Lecture notes on iris: Higher-order concurrent separation logic. https://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf.
- [10] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. J. ACM, 46(5):720-748, sep 1999. ISSN 0004-5411. doi: 10.1145/324133.324234. URL https://doi.org/10.1145/324133.324234.
- [11] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, page 207–216, New York, NY, USA, 1995. Association for Computing Machinery. ISBN 0897917006. doi: 10.1145/209936.209958. URL https://doi.org/10.1145/209936.209958.
- [12] A. Castañeda and M. Piña. Fully read/write fence-free work-stealing with multiplicity, 2021.
- [13] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '05, page 21–28, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1581139861. doi: 10.1145/ 1073970.1073974. URL https://doi.org/10.1145/1073970.1073974.
- [14] H.-H. Dang. Scaling Up Relaxed Memory Verification with Separation Logics. PhD thesis, MPI-SWS, 2023. draft.

- [15] H.-H. Dang, J.-H. Jourdan, J.-O. Kaiser, and D. Dreyer. Rustbelt meets relaxed memory. Proc. ACM Program. Lang., 4(POPL), dec 2019. doi: 10.1145/3371102. URL https://doi.org/10.1145/3371102.
- [16] H.-H. Dang, J. Jung, J. Choi, D.-T. Nguyen, W. Mansky, J. Kang, and D. Dreyer. Compass: Strong and compositional library specifications in relaxed memory separation logic. In Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2022, page 792–808, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392655. doi: 10.1145/3519939.3523451. URL https://doi.org/10.1145/3519939.3523451.
- [17] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, and D. Costanzo. Certikos: An extensible architecture for building certified concurrent os kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 653–669, USA, 2016. USENIX Association. ISBN 9781931971331.
- [18] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In Proceedings of the 15th International Conference on Distributed Computing, DISC '01, page 300-314, Berlin, Heidelberg, 2001. Springer-Verlag. ISBN 3540426051.
- [19] N. Hemed, N. Rinetzky, and V. Vafeiadis. Modular verification of concurrency-aware linearizability. In Proceedings of the 29th International Symposium on Distributed Computing - Volume 9363, DISC 2015, page 371–387, Berlin, Heidelberg, 2015. Springer-Verlag. ISBN 9783662486528. doi: 10.1007/978-3-662-48653-5\_25. URL https://doi.org/10.1007/978-3-662-48653-5\_25.
- [20] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst., 12(3):463–492, jul 1990. ISSN 0164-0925. doi: 10.1145/78969.78972. URL https://doi.org/10.1145/78969.78972.
- [21] J. Jung, J. Lee, J. Choi, S. Park, and J. Kang. Modular verification of safe memory reclamation in concurrent separation logic. Submitted to Object-oriented Programming, Systems, Languages, and Applications (OOPSLA), 2023.
- [22] R. Jung. Understanding and Evolving the Rust Programming Language. PhD thesis, MPI-SWS, 2020.
- [23] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 637–650, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450333009. doi: 10.1145/2676726.2676980. URL https://doi.org/10.1145/2676726.2676980.
- [24] R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018. doi: 10.1017/S0956796818000151.
- [25] R. Jung, R. Lepigre, G. Parthasarathy, M. Rapoport, A. Timany, D. Dreyer, and B. Jacobs. The future is ours: Prophecy variables in separation logic. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019. doi: 10.1145/3371113. URL https://doi.org/10.1145/3371113.

- [26] J.-O. Kaiser, H.-H. Dang, D. Dreyer, O. Lahav, and V. Vafeiadis. Strong logic for weak memory: Reasoning about release-acquire consistency in iris. In *European Conference on Object-Oriented Programming*, 2017.
- [27] J. Kang. deque-proof. https://github.com/jeehoonkang/crossbeam-rfcs/blob/deque-proof/text/2018-01-07-deque-proof.md, 2018. Accessed: 2023-01-26.
- [28] J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer. A promising semantics for relaxed-memory concurrency. SIGPLAN Not., 52(1):175–189, jan 2017. ISSN 0362-1340. doi: 10.1145/3093333.3009850. URL https://doi.org/10.1145/3093333.3009850.
- [29] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605587523. doi: 10.1145/1629575.1629596. URL https://doi.org/10.1145/1629575.1629596.
- [30] M. Kokologiannakis, A. Raad, and V. Vafeiadis. Model checking for weakly consistent libraries. In Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, page 96–110, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367127. doi: 10.1145/3314221.3314609. URL https://doi.org/10.1145/3314221.3314609.
- [31] B. Kragl and S. Qadeer. The civl verifier. In 2021 Formal Methods in Computer Aided Design (FMCAD), pages 143–152, Oct 2021. doi: 10.34727/2021/isbn.978-3-85448-046-4\_23.
- [32] M. Krogh-Jespersen, T. Dinsdale-Young, and L. Birkedal. Verifying a concurrent data-structure from the dartino framework in iris. 2017.
- [33] N. M. Lê, A. Pop, A. Cohen, and F. Zappa Nardelli. Correct and efficient work-stealing for weak memory models. In Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, page 69–80, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450319225. doi: 10.1145/2442516.2442524. URL https://doi. org/10.1145/2442516.2442524.
- [34] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *PDCS '98*, 1998.
- [35] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, June 2004. ISSN 1045-9219. doi: 10.1109/TPDS.2004.8. URL https://doi.org/10.1109/TPDS.2004.8.
- [36] M. M. Michael, M. T. Vechev, and V. A. Saraswat. Idempotent work stealing. In Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '09, page 45–54, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583976. doi: 10.1145/1504176.1504186. URL https://doi.org/10.1145/1504176.1504186.
- [37] A. Morrison and Y. Afek. Fence-free work stealing on bounded tso processors. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages*

- and Operating Systems, ASPLOS '14, page 413–426, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450323055. doi: 10.1145/2541940.2541987. URL https://doi.org/10.1145/2541940.2541987.
- [38] I. Mulder, R. Krebbers, and H. Geuvers. Diaframe: Automated verification of fine-grained concurrent programs in iris. In Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2022, page 809–824, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392655. doi: 10.1145/3519939.3523432. URL https://doi.org/10.1145/3519939.3523432.
- [39] S. O. Mutluergil and S. Tasiran. A mechanized refinement proof of the chase—lev deque using a proof system. Computing, 101(1):59–74, jan 2019. ISSN 0010-485X. doi: 10.1007/s00607-018-0635-4. URL https://doi.org/10.1007/s00607-018-0635-4.
- [40] B. Norris and B. Demsky. Cdschecker: Checking concurrent data structures written with c/c++ atomics. ACM SIGPLAN Notices, 48, 10 2013. doi: 10.1145/2544173.2509514.
- [41] P. W. O'Hearn. Resources, concurrency, and local reasoning. Theor. Comput. Sci., 375(1-3): 271-307, apr 2007. ISSN 0304-3975. doi: 10.1016/j.tcs.2006.12.035. URL https://doi.org/10.1016/j.tcs.2006.12.035.
- [42] G. Rito and H. Paulino. Scheduling computations with provably low synchronization overheads. CoRR, abs/1810.10615, 2018. URL http://arxiv.org/abs/1810.10615.
- [43] T. van Dijk and J. van de Pol. Lace: Non-blocking Split Deque for Work-Stealing. In *MuCoCoS*, volume 8806 of *LNCS*, pages 206–217. Springer, 2014. doi: 10.1007/978-3-319-14313-2\_18.
- [44] S. F. Vindum, D. Frumin, and L. Birkedal. Mechanized verification of a fine-grained concurrent queue from meta's folly library. In *Proceedings of the 11th ACM SIGPLAN International Conference* on Certified Programs and Proofs, CPP 2022, page 100–115, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391825. doi: 10.1145/3497775.3503689. URL https: //doi.org/10.1145/3497775.3503689.
- [45] J. Wang, B. Trach, M. Fu, D. Behrens, J. Schwender, Y. Liu, J. Lei, and V. Vafeiadis. Bwos: Formally verified block-based work stealing for parallel processing. In *Proceedings of the 17th USENIX Conference on Operating Systems Design and Implementation*, OSDI'23. USENIX Association, 2023. To appear.

# Acknowledgments in Korean

저의 석사 과정을 처음부터 끝까지 이끌어주신 강지훈 교수님께 깊은 감사의 말씀을 드립니다. 늘 부족한 제가 연구를 잘할 수 있도록 지도해 주시고, 미래에 대해 걱정이 될 때마다 귀중한 조언을 해주셨습니다. 교수님의 도움 없이는 저의 연구가 세상에 나오지 못했을 것입니다.

저의 곁에 있어주신 KAIST 동시성 및 병렬성 연구실의 모든 구성원 분들께 감사드립니다. 특히 석사 과정 전부터 지금까지 동시성 검증 프로젝트를 함께 진행하며 제 연구의 방향성에 대해 큰 도움을 주신 정재황님과 이장건님, 그리고 느슨한 메모리 검증 프로젝트를 함께해 주신 박선호 님과 김재우님께 감사드립니다.

마지막으로, 저의 평생을 지켜보며 응원해 주시는 저의 가족에게 감사드립니다. 저의 결정을 늘 존중 하며 아낌없는 지원을 해주신 부모님, 제 삶의 4년 선배로서 늘 귀감이 되어준 형에게 감사합니다. 덕분에 제가 이 자리에 서게 될 수 있었습니다.