Separation Logic for Concurrency and Persistency

Simon Friis Vindum

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Separation Logic for Concurrency and Persistency

A Dissertation
Presented to the Faculty of Natural Sciences
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Simon Friis Vindum
October 31, 2023

Abstract

Ensuring correctness of programs in the presence of concurrency and persistence is extremely difficult. This thesis contributes to the field of formal verification of such programs.

Chapter 2 considers the Michael-Scott queue (MS-queue), a concurrent non-blocking queue. We use the Iris and ReLoC logics to show that the original MS-queue contextually refines a coarse-grained queue. To simplify the proof, we extend separation logic with a generally applicable *persistent points-to predicate* for representing immutable pointers. This is based on a novel resource algebra of *discardable fractions* that generalizes fractional permissions. Chapter 3 presents the first formal specification and verification of an efficient concurrent queue from Meta's C++ library Folly. We use ReLoc to formally prove that the queue is a contextual refinement of a simple coarse-grained queue. A key challenge of the MPMC queue is that it has a so-called *external linearization point*, which ReLoC has no support for reasoning about. We thus extend ReLoC with novel support for reasoning about external linearization points.

Weak persistent memory (a.k.a. non-volatile memory) is an emerging technology that offers fast byte-addressable durable main memory. Chapter 4 presents Spirea, the first concurrent separation logic for verification of programs under a weak persistent memory model. The logic combines features from Iris and Perennial with novel techniques to support high-level modular reasoning about crash-safe and thread-safe programs. We use Spirea to verify several challenging examples with modular specifications, in particular non-blocking durable data-structures with null-recovery. This is the first time durable data-structures have been verified with a program logic.

Chapter 5 proposes a novel *nextgen* modality that makes it possible to update resources in ways that are non-frame-preserving. Among other things, this is usual for reasoning about crashes. We develop the modality as an extension to Iris. We show that two existing Iris modalities are special cases of the nextgen modality. The modality can thus be seen as a generalization and simplification of the Iris base logic. To demonstrate the utility of the nextgen modality we use it to construct a program logic for a language with stack allocation and with function returns that clears entire stack frames. The nextgen modality is used to great effect in the reasoning rule for return, where a modular and practical reasoning rule is otherwise out of reach. This is the first separation logic for a high-level programming language with stack allocation.

Chapter 6 proposes a more powerful nextgen modality that can be used to reason about non-deterministic crashes. We show how the modality can be used in the model of Spirea. This is the first step towards an improved variant of Spirea that can be used to show strong specifications for durable data-structures akin to HOCAP-style or TaDA-style specifications.

All contributions and results are formalized in the Coq proof assistant.

Resumé

At sikre korrektheden af programmer i tilstedeværelsen af concurrency og persistens er enormt vanskeligt. Denne afhandling bidrager til feltet af formæl verifikation af sådanne programmer.

Kapitel 2 omhandler Michael-Scott-køen (MS-køen), en concurrent ikke-blokerende kø. Vi bruger Iris- og ReLoC-logikken til at vise det den originale MS-kø er en contextual refinement af en coarsegrained kø. For at forenkle beviset, udvider vi separationslogik med et generelt anvendeligt persistent point-to prædikat for at repræsentere uforanderlige pointere. Dette er baseret på en ny ressource algebra af kasserbare brøker, der generaliserer brøktilladelser. Kapitel 3 præsenterer den første formelle specifikation og verifikation af en effektiv concurrent kø fra Metas C++ bibliotek Folly. Vi bruger ReLoc til formelt at bevise, at køen er en contextual refinement af en simpel, coarse-grained kø. En nøgleudfordring ved MPMC-køen er, at den har et såkaldt *eksternt lineariseringspunkt*, som ReLoC har ikke understøtter at ræsonnere omkring. Vi udvider derfor ReLoC med nye egenskaber til ræsonnement om eksterne lineariseringspunkter.

Svag persistent hukkommelse (også kendt som non-volatile memory) er en ny teknologi der tilbyder hurtig byte-adresserbar persistent primær hukommelse. Kapitel 4 præsenterer Spirea, den første concurrent separationslogik til verifikation af programmer under en svag persistent hukommelsesmodel. Logikken kombinerer aspekter fra Iris og Perennial med nye teknikker til at understøtte modulær ræsonnement på et højt niveau om nedbrugssikre og trådsikre programmer. Vi bruger Spirea til at verificere flere udfordrende eksempler med modulære specifikationer, inklusiv ikke-blokerende holdbare datastrukturer med nul-gendannelse. Dette er første gang, holdbare datastrukturer er blevet verificeret med en programlogik.

Kapitel 5 foreslår en ny nextgen modalitet, der gør det muligt at opdatere ressourcer på måder, der ikke er frame-preserving. Det er blandt andet brugbart til at ræsonnere om systemnedbrud. Vi udvikler modaliteten som en udvidelse til Iris. Vi viser, at to eksisterende Iris-modaliteter er særtilfælde af nextgen-modaliteten. Modaliteten kan således ses som en generalisering og forenkling af Iris baselogikken. For at demonstrere brugbagheden af nextgen-modaliteten bruger vi den til at konstruere en programlogik for et sprog med stakallokering og med funktionsreturnering, der rydder hele stakrammer. Nextgen-modaliteten bruges med stor gevindst i ræsonnementreglen for returnering, hvor en modulær og praktisk ræsoneringsregel ellers er uden for rækkevidde. Dette er den første separationslogik for et højtniveaus

programmeringssprog med stakallokering.

Kapitel 6 foreslår en mere udtrykskraftig nextgen-modalitet, der kan bruges til at ræsonnere om ikke-deterministiske systemnedbrud. Vi viser hvordan modaliteten kan bruges i modellen af Spirea. Dette er det første skridt mod en forbedret variant af Spirea, der kan bruges til at vise stærke specifikationer for holdbare datastrukturer, tilsvarende HOCAP-stil eller TaDA-stil specifikationer.

Alle bidrag og resultater er formaliseret i Coq bevisassistenten.

Acknowledgments

I would like to thank my advisor Lars Birkedal for his support and guidance throughout my PhD. Thank you for always being generous with your time and advice, and for the great research environment you have created in Aarhus and that I have enjoyed being a part of.

Thanks to all the people in the Logic and Semantics group and the Programming Languages group. There are far too many to mention you all, but I will mention a few. Thanks to my office mate Simon Gregersen for always being helpful with Coq and many other things. Thanks to Zesen Qian for our many coffee chats—I missed you when you left. Thanks to Philipp Stassen for our winter bathing sessions and sauna conversations. Thanks to Amin Timany for always being able and willing to help me out in a pinch with the most tricky Coq problems. Thanks to Léon Gondelman for always cheering on my work and for being a great friend.

During my PhD I have had the opportunity of collaborating with some great people: Dan Frumin, Yixuan Chen, and Aïna Linn Georges. Thanks to all of you, it was a pleasure.

Thanks to François Pottier and Azalea Raad for agreeing to be on my thesis committee. I am grateful for having such deeply qualified people, whose work I admire, on the committee.

I would like to thank Viktor Vafeiadis for hosting me at MPI-SWS in Kaiserslautern. Thanks to all the wonderful people at MPI in Kaiserslautern who welcomed me and included me in their social activities. It was a pleasure to meet all of you and to be a part of your community. Thanks to your hospitality I never felt alone even though I was far away from home. Even though I only knew you for a short three months many of you have made a lasting impression. Thanks, in particular, to Léo Stefanesco who helped me out a lot during my stay and who is such a fun and interesting person to be around.

Thanks to the danish system surrounding subsidized education and PhD funding that made it possible for me to spend slightly more than 8 years at the university for free and with financial support.

Thanks to my friends and family. Most importantly to my parents, Bodil and Knud. Thank you for always believing in me, for giving me the freedom to do my own things and to pursue my own interests. I have learned a lot from both of you and continue to draw inspiration from you. Thanks to my aunt Eva who, among many things, sent me a birthday box containing danish rye bread, liver pâté, and

homemade pickled beetroot while I was in Kaiserslautern.

Last and most definitely not least, thanks to my partner Gry who was always there for me with her love and support during both the good times and the bad times. Thank you for taking an interest in my work and for listening to my—sometimes seemingly endless—ramblings about it.

Simon Friis Vindum,
Copenhagen, October, 2023.

Contents

Part I

Overview

Chapter 1

Introduction

As anyone who has ever used software can attest to: software has bugs. This can be of great wonder to laypeople. How come we can build so many amazing technological marvels, and yet even the simplest of programs often misbehave? The plain answer is that software is very intricate and complicated. Therefore, getting it right is tremendously difficult. To address this fundamental problem, many related fields of computer science are dedicated to the research of tools and techniques that can help improve the reliability and correctness of programs. One such "tool" is to mathematically prove that a given program works—for some meaning of "works". That is, similarly to how one can mathematically prove, for instance, that Pythagoras's theorem is true, one can (in theory) also prove results such as "Microsoft Word does never crash". However, as programs are immensely complicated, proofs about them tends to be just as complicated or even much more complicated. This means that carrying out such proofs is very hard and that, just as the program might have a bug, the proof itself could contain mistakes. One solution to these challenges is to use *mechanized formal program logics*. Let us unpack what this means. A *formal logic* consists of a strict set of rules that can be used to build a proof. One can imagine the rules as "puzzle pieces" that can be assembled together, and a proof is valid if all the puzzle pieces fit together. If the logic is *sound*, then whenever one has constructed a valid proof the proven thing is in fact true. One benefit of using a formal logic as that the rules in the logic can make it much easier to prove properties about program compared to using "normal" math. Another benefit is that checking whether a proof is correct is quite easy as it merely amounts to checking if the rules fit together. In fact, it is possible to create a "proof language", where one can write proofs using the rules in such a way that a computer program can check that all the rules are used properly. Such a system is called a *proof assistant* and a logic or proof that makes use of and is checked by a proof assistant is called *mechanized*. Taken together, using a mechanized formal program logic to verify programs both makes the task of proving easier, and gives the proofs a very high level of reliability as they have been machine checked.

Simplifying and summarizing a lot, this thesis contributes to the field of mecha-

nized formal program logics by:

- *Applying existing program logics* to verify complicated, intricate, efficient, and practically important data-structures. In particular, we verify the well known Michael-Scott Queue and a less well known queue from Meta's open source library Folly. Verifying these data-structures is both a significant result on its own and the verification effort contributes to a growing body of knowledge about applying program logics.
- *Developing a new program logic*, Spirea, for the purpose of reasoning about programs that use *persistent memory* (or non-volatile memory). This logic is state-of-the-art and capable of verifying programs beyond the reach of prior logics. For instance, we use it to verify several durable concurrent data-structures with null-recovery.
- As a by-product of the above two activities we contribute several innovations that are more widely and generally applicable. Most notably we introduce: *discardable fractions* and a *persistent points-to predicate*, an extension to ReLoC for external linearization points, and a *nextgen* modality that adds *non-frame-preserving* updates to separation logic.

The remainder of this introduction explains the program verification challenges pertinent to this thesis, gives an overview over existing program logics, provides a more detailed account of the overall contributions, and finishes with a vision for future work.

1.1 Challenges: Concurrency and Persistency

To illustrate the challenges one faces when programming for a modern CPU, Figure 1.1 contains a rough sketch of a CPU attached to a main memory. We use this figure to explain two key verification challenges: concurrency and persistency.

1.1.1 Concurrency

The octagons at the top left of the figure depicts *CPU cores*. For ease of illustration, the depicted CPU has 4 cores, but modern CPUs often have more. In fact, some of the most expensive CPUs available to consumers today have as many as 64 cores. If a CPU has n cores then it can carry out n tasks simultaneously. For a program to take advantage of this, it needs to be written with concurrency and parallelism in mind. One way to do this is by using *threads*, where several thread can be executed in parallel on multiple cores. The upside to doing this is that the program can be significantly faster as it make good use of all the cores available. The downside is that ensuring the correctness of such a program can be quite difficult.

To illustrate the challenges from a programmers point of view, consider the code examples in Figure 1.2. These four examples all depict a push operation implemented

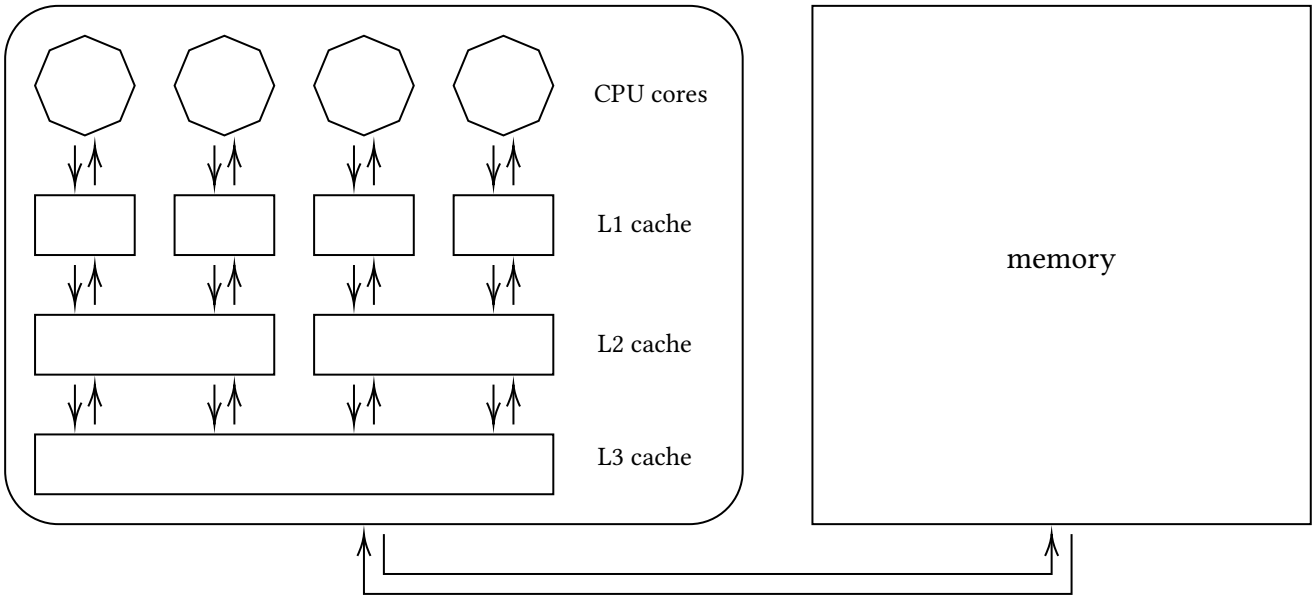


Figure 1.1: A modern CPU with memory attached

on a stack. The operation push stack v pushes the values v to the stack stack. To keep the example simple the stack is implemented as a pointer to a list data struture. The first implementation of push in Figure 1.2a works by

1. reading the stack pointer to obtain the head of the list,
2. prepending (or *consing*) the new value v to the list,
3. writing the updated list to the pointer,
4. and finally it returns unit (denoted by $()$).

In a single-threaded setting without parallelism this straightforward implementation of push is perfectly correct. However, if multiple threads might execute push in parallel, then the implementation is incorrect. In this case we can imagine the following *interleaving* where two threads execute push in parallel: Both the first and the second thread execute the first two lines of push. This means that the value they read from stack is the same. The first thread then executes the third line. Its value is now in the stack. The second thread then executes the third line. It now erased the first thread's value from the stack. Hence, after executing push operations in parallel the final effect is that only one element is added to the stack. This is incorrect, as push should only add elements to a stack, not remove elements added previously.

When several threads execute in parallel they are, in a sense, racing against each other. In the above example, which thread gets to execute which line first depends on numerous variables that can not be precisely controlled. As such, we have to assume that this order is *non-deteministic* and for a program to be correct it must be

```
push stack v ≜
  let head = ! stack in
  let newHead = v :: head in
  stack ← newHead;
  ()
```

(a) A simple implementation of push.

```
push stack v ≜
  acquire lock;
  let head = ! stack in
  let newHead = v :: head in
  stack ← v :: head
  release lock;
  ()
```

(b) A thread-safe blocking implementation of push.

```
push stack v ≜
  let head = ! stack in
  let newHead = v :: head in
  if CAS stack head newHead
  then ()
  else push stack v
```

(c) A non-blocking thread-safe implementation of push.

```
push stack v ≜
  let head = ! stack in
  let newHead = v :: head in
  flush newHead;
  fence;
  if CAS stack head newHead
  then ()
  else push stack v
```

(d) A non-blocking thread-safe and crash-safe implementation of push.

Figure 1.2: Various implementations of a push operation on a stack

correct for all interleavings. The non-determinism means that a program can work as it should and appear correct, but then occasionally fail catastrophically. Hence, in the presense of non-determinism the act of *testing* a program becomes a less reliable method for identifying bugs.

The above example highlights another important point. Whether the code in Figure 1.2a is correct depends on the context in which it is used. In a single-threaded setting it is correct, but in a multi-threaded setting it is incorrect. If a data-structure or program works in a multi-threaded setting it is said to be *thread-safe*.

One easy way to make a function thread-safe is to use a *lock*. Figure 1.2b shows how to adapt the push operation to use a lock. The code in the example assumes that there exists a lock associated with the stack with the name lock. The first line of push now *acquires* the lock and the last line *releases* the lock. The lock itself ensures that only one thread can have the lock at any given time. This means that if two threads now execute push in parallel, then whoever executes acquire first will recieve the

lock. The other thread will be *blocked* on the first line, and only execute further into the push operation once the former thread releases the lock. This ensures that only one thread can ever execute the lines in between acquire and release at a given time. This property is called *mutual exclusion*, and effectively "removes" any parallelism inside push. With the lock in place the problem we identified before is now solved, and this version of push is thread-safe.

An operation that uses a lock as above is called *blocking* since, once one thread has acquired the lock, all other threads are blocked and has to wait. In practice blocking can become a significant bottleneck and negatively impact performance. The alternative to blocking operations are *non-blocking* operations. These avoid the use of locks by instead relying on atomic *read-modify-writ* (RMW) operations. Figure 1.2c shows how to implement a non-blocking push operation by using the **CAS** RMW operation. The expression **CAS** $\ell\ v_1\ v_2$ performs, in a single atomic step, the following:

1. It reads the current value of the location ℓ.
2. It compares whether the current value is equal to v_1.
3. If the comparison is true, the v_2 is written to the location, and the operation evaluates to **true** to signify success. Otherwise, ℓ is left unchanged and **false** is returned.

In the updated push operation **CAS** is used on the third line to write to the stack. The use of **CAS** ensures that if the content of the stack changed in between reading the stack and the attempted write, then nothing will happen. In this case **CAS** evaluates to **false**, and the push operation is restarted from the beginning through the recursive call to push. This simultaneously solves the problem with the first push implementation, and avoids blocking as in the second push implementation. RWM operations operation are essential to implementing any sort of parallel construction. They are supported at the level of the CPU, and the CPU guarantees the atomicity of the operations. These lock-free operations are sometimes called *fine-grained* (as opposed to *coarse-grained* operations that make use of locks). Here fine-grained refer to the fact that parallelism is not restricted into big exclusive sections protected by locks, but only requires atomicity of "small" RMW operation. The downside to fine-grained operations is that they are much harder to get right than lock based ones and that they usually use RMW operations in very clever ways. Where the use of a lock effectively got rid of interleavings, for fine-grained operations we are back to worrying abut all possible interleavings.

1.1.2 Weak Memory

Over the last couple of decades CPUs have become much faster, but *main memory* has not seen the same level of performance increases. This means that the CPU is, comparatively, much faster than main memory on modern computers. As a

rough number to illustrate the point, on reading an address from the memory might take around the same time as executing 100 instructions on the CPU. With this discrepancy in performance the speed of most programs would be bound by the speed of the memory and the CPU would spend most of its time waiting for the slow memory. I write "would" because to (partially) avoid this problem modern CPUs pull off all sorts of clever tricks to make up for the slow the memory. One these tricks is to use *CPU caches*, these caches are small capacity storage that sits on the CPU very close to the cores as depicted in Figure 1.1. Since this storage is so close to the cores, it can be accessed much faster than the larger, but further away, main memory. CPU caches are divided into several layers with the CPU depicted in the figure having three layers of cache. With each increasing layer the cache becomes larger and slower. To make use of the cache, memory operations carried out by the CPU operate on the cache in favor of the memory if possible. For instance, to read the value of a location ℓ a core first consults the L1 cache to see if it contains the value. If the L1 cache contains the location, then the read returns this value for the location. Otherwise, the core will proceed its search up the cache hierarchy. Only if none of the caches contain the location, a *cache miss*, is a request sent to the memory to fetch the location. The CPU is responsible for managing its caches. This involves loading data from the memory into the caches, flushing data from caches into the memory, invalidating caches, searching through the cache hierarchy, and more. All of this might sound like overhead that would make operations slower, but, since the CPU is so much faster than the memory, having the CPU perform additional work is still beneficial in terms of performance if it can decrease usage of the slow memory.

Unfortunately, the CPU caches are not transparent to programs, they leak into the way in which concurrent programs might behave. Usually programmers think of the memory as one global shared heap of values. With this vies of the world, all threads reading a given location in memory will always see the same value. And if one threads writes a value to a location, then afterward all other threads will read that write. This behavior of the memory is called *sequential consistency*. However, as we can see on Figure 1.1 some of the cache layers are per-core. For instance, each core has their own L1 cache. This means that different cores may not always see the same "view" of the world. One core might have one value for ℓ in its L1 cache whereas another core might have a different value in its L1 cache. This can cause behaviors in multithreaded programs that are inconsistent w.r.t. sequential consistency. When memory exhibit behavior that does not match sequential consistency, the *memory model* is said to be *weak* or *relaxed*. A canonical example of this is the program:

$$\begin{array}{l|l} x \leftarrow 1; & y \leftarrow 1; \\ !y \quad // \, 0 & !x \quad // \, 0 \end{array}$$

In this program we assume that the location x and y both initially have the value 0. Thinking in terms of the possible interleavings when executing this program, one of the two threads must execute its write before the other thread. The thread that carries out its write last must then necessarily carry out its read after the other

thread has written to the location. Hence, we would expect that at least one of the two reads results in the value 1 and sequential consistency would ensure this expectation. Alas, in a weak memory model both threads might actually merely write the value of 1 to their own L1 cache, and each write is then invisible to the other thread. Hence, both threads might read 0 which is entirely unexpected.

CPU caches are not the only cause of weak memory behavior, another source is compiler optimizations. Consider the following program:

$$\begin{array}{l} x \leftarrow 1; \\ y \leftarrow 1; \end{array} \Bigg\| \begin{array}{l} \textbf{if } !y = 1; \\ \textbf{then assert } (!x = 1) \end{array}$$

In this example **assert** denotes a function that crashes (or gets stuck) if the given argument is not true. Assuming sequential consistency, this program would never crash, as the write to y happens after the write to x, and hence if the right thread reads the write of 1 to y it surely also reads the prior write of 1 to x. In fact, the memory on x86 CPU would also never cause this program to crash, as the CPU architecture ensures that writes by the same thread are seen in order by other threads (this is enforced as part of the total store order that x86 guarantees). However, if this program is written in a compiled programming language then in many cases (such as C or Rust) the compiler is allowed to reorder independent writes such as the two writes on the left. Due to this, the assertion is not guaranteed. Hence, weak memory models for compiled languages are often even weaker than the machine-level ones as they have to account for optimizations carried out by both the CPU and the compiler.

The examples given here are both quite simple, but, as the reader can imagine, in a larger program ensuring correctness under weak memory can be quite tricky as one needs to take into account behavior that is quite counterintuitive and unexpected.

1.1.3 Persistent Memory

On contemporary computers the main memory is *volatile*. This means that when the computer is powered off (or if it crashes and restarts) all data in the memory is lost. This is because modern memory is implemented using a technology called DRAM which requires constant power in order for the bits stored in it to remain there. For data that needs to be stored permanently, *secondary storage* is used. Secondary storage includes technology such as solid-state drives (SSDs) and hard-disk drives (HDDs). These are slower than main memory and do not allow for efficient random access, but they are *non-volatile*, or *durable*, meaning that the bits stored in them remain there even in the absence of power. [1] The implications of the above are that a program uses the main memory to store any data that it needs to access efficiently, and the secondary storage for any data that it want permanently stored. For data where both of these are required, copies are usually kept such that the data is in both the memory and in secondary storage.

[1] SSDs might in fact begin to lose bits if they are not powered on for several years though.

The above describes the *traditional* storage hierarchy. A new family of technologies called *persistent memory* (also called non-volatile memory) are disrupting this hierarchy. Non-volatile memory covers new physical ways to implement memory that are comparable to traditional main memory, in that they offer efficient random access, but, unlike traditional memory, are also non-volatile. From a programmers point of view, this means that they can now write programs that store things *permanently* in memory. This can bring great performance benefits as persistent memory is faster than other types of durable storage. Furthermore, it could simplify programs that both need efficient access and durability as they can keep just one copy of their data in the persistent memory.

Writing correct programs for persistent memory introduces to additional challenges that must be taken into account.

Crashes First of all, programs that make use of durable storage (whether that be persistent memory or something else) must be *crash-safe*. Here a crash means a full-system crash where a program is abruptly terminated, for instance due to a hardware failure. In the real world, programs need to consider the possibilities of crashes. For programs that do not store anything in persistent storage, crashes are unimportant. At a crash the program looses all of its data and if executed after the crash there is no trace of the prior execution. In contrast, when a program that *do* use durable storage is executed after a crash, data from the prior execution remains. A program that uses durable storage should of course attempt to recover data after a crash such that the data is not lost. Since a crash can *non-deterministically* happen at any time, when running after a crash the program might find the stored data to be in a state that it would otherwise never be in. Crash-safety means that the program is guaranteed not to get stuck even when executed after a crash.

Weak Persistency Secondly, the possible state of the persistent memory after a crash is affected by the CPU caches. Even with persistent memory, CPU caches are still volatile, and as we have seen, when a core carry out writes they are initially store only in the cache. This means that when something is written to the memory, it is not actually persisted until the write is eventually flushed from the caches into the memory. On present day CPUs the order in which writes reach the memory is *weak* and may happen out of order. This means that the code in Figure 1.2c as *not* crash-safe. On the second line a list with the new value is allocated, and on the third line the updated memory is assigned to the stack location. Since the two memory operations might persist in any order, if there is a crash after the **CAS** operation it might be the case that the write to stack persisted but the allocation of the new list head did not. In this case the stack location would be pointing to unknown data in the memory, and if code attempted to use the stack after a crash it would get stuck. To avoid such issues, modern CPUs contain instructions that make it possible for programs to ensure that writes persist in certain desired orders. An example of using such instructions to make the push operation stack safe appears in Figure 1.2d.

Here the use of **flush** and **fence** in between the creation of the new list and the **CAS** ensures that the former allocation is guaranteed to persist prior to the latter write. We explain these operations in detail in Chapter 4, for now the point is merely that these instructions are critical to get right in terms of correctness. On the other hand, these operations are also expensive in terms of CPU time, so they should only be used when necessary. Again, note how the setting affects whether a program is correct. As explained, Figure 1.2c is correct in a setting with only concurrency, but if we extend the setting to one with persistency, it is no longer correct.

1.2 Correctness Criteria for Concurrent and Durable Data-Structures

We have now seen some of the challenges that arises due to concurrency and persistency. We saw the example Figure 1.2a which is incorrect in a setting with concurrency and Figure 1.2c which is correct. But what precisely makes the latter correct? Concurrent and durable data-structures are so complicated that it not immediately obvious what it would mean for one to be correct. In fact, there is not just one criterion by which such a data-structure is correct. In this section we will cover some of the most common correctness criteria for concurrent and durable data structures, beginning with the most well known.

1.2.1 Linearizability

In 1987 Herlihy and Wing proposed the notion *linearizability* [HW90; HW87]. The term linearizability has a precise meaning defined in terms of histories, or traces, corresponding to executions. But, the term can also be understood and used on a more intuitive level—which it often is. On this level linearizability states that a concurrent operation should appear to take place instantaneously at some point after it is called and before it returns. At this point it should have the same effect as a sequential operation would have. The push operation in Figure 1.2a it not linearizable because, as we observed, there exist executions of this operation that does not correspond to any sequential execution of a push operation. The push operation in Figure 1.2c is linearizable. Its linearization point is at the **CAS** operation, and, due to the guarantees of the **CAS** operation, at this point the effect is certain to correspond to a sequential push operation on a stack. Linearization points can be divided into three categories [DD15]: fixed, future-dependent, and external linearization points. The linearization point in Figure 1.2c is a *fixed linearization* point as the linearization point is always exactly at the **CAS** operation. These linearization points are the simplest to handle from a verification point of view. A *future-dependent* linearization point is one where the exact location of the linearization point can only be known in hindsight. An *external* linearization point is one where the linearization point of an operation does not occur during the execution of the operation itself, but during the execution of some other operation. In Chapter 2 we see an example of

a future-dependent linearization point and in Chapter 3 we see an example of an external linearization point.

Linearizability is only about concurrency, for durability Izraelevitz et al. [IMS16] extended the notion to *durable linearizability*. This states that a durable data-structure must, in addition to it linearization point, have a *persist point* where it persists. The meaning of the persist point depends on whether the data-structure is *non-buffered* durable linearizable or *buffered* durable linearizable. The former is the strongest notion, an means that after a crash the effect of all executed persist points has persisted. The latter notion loosens this requirement by stating that only some prefix of the executed persist points must have persisted.

1.2.2 Contextual Refinement

In Figure 1.2b we saw a simple example of an implementation of push made thread-safe by using a lock. Using a lock in this way can in general be used to make simple sequential data structures thread-safe, and it is quite easy to see that the resulting implementation is correct. This is in contrast to fine-grained implementations that can be quite intricate and hard to understand. One approach to verifying tricky and intricate (but very efficient) thread-safe data structures is to show that they are in fact *contextual refinements* of the simple easy to understand ones. Loosely speaking, this means that they are "as correct"—if one accepts that the simple data structure is correct then one is also forced to accept that the complicated one is. Contextual refinement is defined as follows (this definition is slightly simplified to for presentation):

$$\vdash e_1 \precsim_{ctx} e_2 : \tau \triangleq \forall K, v.\, K[e_1] \to^* v \Rightarrow \exists v.\, K[e_2] \to^* v$$

This reads: the program e_1 is a contextual refinement of the program e_2 if any *context* K that when given e_1 terminates in some value, then K when given e_2 also terminates in some value. The use of termination is used as a proxy to determine whether a context is able to tell e_1 and e_2 apart. Since this is proven for all contexts, one can imagine K as being an adversary that tries to distinguish e_1 and e_2 by terminating when seeing one and not terminating when seeing the other. With this approach to verification one can think of the right-hand side in the definition of contextual refinement (e_2 above) as begin the specification. In terms of the push example, behaving like Figure 1.2b is what it means to be a correct concurrent push operation. In Chapter 2 and Chapter 3 we use the contextual refinement approach to verify two concurrent data structures.

Contextual refinement has not been used to verify durable data-structures, but it is quite likely that it could, if the stepping relation $\to^*$ in the definition of contextual refinement incorporates crashes. We will not consider that approach in this thesis.

1.2.3 Logical Specifications

A third correctness criteria for concurrent data-structures, is to give them a specification phrased inside a program logic. Two such approaches are *HOCAP-style* [SBP13] specifications and TaDA-style specifications [RDG14]. These approaches both capture a property of *logically atomicity*, which expresses what it means, withing the logic, for an operation to appear to take place instantaneously. One could say that these styles of specifications state which reasoning principles a *client* of a linearizable operation can use, phrased in terms of a specification in a program logic. Given this, the strength of this approach is naturally that they are readily applicable for verifying clients that use a linearizable data structure. The downside is that these specifications usually does not translate to a strong property about the data-structure itself independently of the program logic. This is in contrast to linearizability and contextual refinement which are a mathematical properties of a data-structure. In a recent paper, however, it was shown that, in a certain first-order setting (more restricted than what we consider in this thesis), a logically atomic specification for an operation does imply that the operation is linearizable [Bir+21]

Logical specifications have not been explored for durable data-structures for persistent memory.

1.3 Program Logics for Concurrency and Persistency

This section gives a brief overview of the program logics that are of particular relevance to this thesis and explains how they relate to the challenges and correctness criteria mentioned.

1.3.1 Iris

Iris is a state-of-the-art separation logic mechanized in the Coq proof assistant. Iris contains both a particular program logic that can be used to reason about concurrent programs. At the most fundamental level Iris contains a *base logic* that includes the most primitive features of the logic: the BI connectives, ownership over higher-order ghost state, the later modality, and more.

Iris has been used to give both HOCAP-style and TaDa-style specifications for concurrent data-structures. It can verify linearizable data-structures with both fixed, external, and future-dependent linearization points. The last type of linearization point is supported by a feature called *prophices* [Jun+20a]. The Iris program logic assumes sequential consistency, that is, it does not consider the weak behaviors described earlier. Various other Iris-based logics consider weak memory [Dan+20; Dan+22; Kai+17; MJP20].

1.3.2 ReLoC

ReLoC is a program logic for showing contextual refinement [FKB18; FKB20a]. ReLoC is based on Iris and makes it possible to use all the capabilities in Iris to show a *refinement judgment* that implies contextual refinement. In Chapter 2 and Chapter 3 we use ReLoC as our vehicle for the verifications that we carry out.

1.3.3 Perennial

Perennial is a program logic, based on Iris, that extends the Iris program logic with novel features that makes it possible to reason about crashes and show crash-safety. The Perennial program logic assumes a *strong* persistency model and does not consider the weak behaviors arrising with persistent memory and volatile CPU caches.

1.3.4 Owicki-Gries based Program Logics for Persistent Memory

No separation logics have been created that are able to reason about *weak* persistency models. However, two ground-breaking Owicki-Gries based program logics for the weak x86 persistency model [Bil+22; RLV20] have been created. These are the first program logics that are able to verify and reason about programs that use persistent memory.

Since these logics are not based on separation logic they lack some of the features found in separation logics. One of these features is *modularity* derived from separation logics notion of ownership. This means that these logics are not able to give self-contained specifications of durable data-structures as modules that can be used by independent clients.

1.4 Contributions and Structure

In this section I give an overview of the contributions made by this thesis, its structure, the papers and manuscript on which it is based, and my personal contributions to the individual chapters.

1.4.1 List Of Publications and Manuscripts

This thesis includes the following published papers:

[VB21] Simon Friis Vindum and Lars Birkedal. "Contextual refinement of the Michael-Scott queue (proof pearl)." In: *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*. ACM, 2021, pp. 76–90. DOI: 10.1145/3437992.3439930

[VFB22] Simon Friis Vindum et al. "Mechanized verification of a fine-grained concurrent queue from meta's folly library." In: *CPP '22: 11th ACM SIGPLAN*

International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022. Ed. by Andrei Popescu and Steve Zdancewic. ACM, 2022, pp. 100–115. DOI: 10.1145/3497775.3503689

[VB23c] Simon Friis Vindum and Lars Birkedal. "Spirea: A Mechanized Concurrent Separation Logic for Weak Persistent Memory." In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (Oct. 2023). DOI: 10.1145/3622820

Additionally, the thesis includes the following manuscript:

- Simon Friis Vindum, Aïna Linn Georges, and Lars Birkedal. "The Nextgen Modality: A Modality for Non-Frame-Preserving Updates in Separation Logic." In submission.

1.4.2 Contributions

The contributions made by each chapter in Part II are as follows.

Note: Some of the description of the contributions are derived from the text in the individual papers.

Chapter 2

This chapter uses the ReLoC logic to verify the well-known Michael-Scott queue. The contributions in this chapter includes:

- A new resource algebra called *discardable fractions*. This generalizes the resource algebra of fractions to allow for discarding of a non-zero fraction in order to option a duplicable element proving that a fraction was discarded.
- A new persistent points-to predicate based on discardable fractions. This is a general extension of separation logic that makes it simpler and more convenient to express locations that are *immutable*.
- A proof that the Michael-Scott queue is a contextual refinement of a simple lock-based coarse-grained queue. This is the first time that a result of this strength for the full (non-simplified) MS-queue and in a higher-order setting has been shown and mechanized.
- A claim that the proof is particularly elegant as it makes use of the persistent points-to predicate and a notion of *reachability* that concisely captures the invariants maintained by the various nodes in the MS-queue.
- All the contributions are mechanized in Coq.

After the publication of the paper underlying this chapter, discardable fractions has replaced the use of fractions in many (or perhaps most) places in the Coq mechanization of Iris. Furthermore, discardable fractions and persistent points-to predicates has been used to great effect in several other bodies of research [Geo+23; MP22; MWB24; Tim+21; VP23].

Chapter 3

This chapter uses ReLoC to verify a high-performance queue, called the MPMC queue, from the open source library Folly developed and used at scale in production by the company Meta. The contributions in this chapter includes:

- An informal analysis of the MPMC queue and description of how one of its linearization points is external.
- The MPMC queue consists of two submodules: a turn sequencer and a single-element queue. Hoare-style specifications are given for the two submodules.
- A proof that the MPMC queue contextually refines a coarse-grained queue. Since the MPMC queue, unlike the MS queue, might block in some circumstances, the coarse-grained queue used here it not exactly the same as in the last chapter.
- An explanation of why prior versions of ReLoC could not handle external linearization points and an extension of ReLoC, both on paper and in Coq, with support for reasoning about external linearization points. With this addition ReLoC is now able to verify data-structures with all the three types of linearization points.
- All the results in this paper are formalized in the Coq proof assistant.

Chapter 4

This chapter introduces the logic Spirea, a separation logic for reasoning about progarms under a weak memory and persistency model. The contributions in this chapter includes:

- A *resource changing posts crash modality* that can account for the non-deterministic changes in resources at crashes under weak persistency. This post-crash modality supports rules of the form $R \vdash \langle \mathsf{PC} \rangle\, R'$, where R' reflects how R is non-deterministically affected by the crash.
- *Crash-aware invariants*, which, in contrast to Iris-style and GPS-style invariants, are sound under weak persistency. Soundness of Spirea crash-aware invariants relies on having novel proof rules for transfer of resources in and out of invariants. Our Spirea invariants are *crash-aware*, meaning that they can be preserved under our post-crash modality and thus facilitate resource transfer between code executing before and after a crash. This is the first time a separation logic contains invariants that can be used to this end.
- An assortment of features to handle persistent memory instructions: *Post-fence modalities*, a *post-crash flush modality*, and *state lower-bounds* w.r.t. fences. These work in tandem to reason about weak flushes and synchronous and asynchronous fences.

Chapter 5

This chapter a novel *nextgen* modality is proposed and a case study using it is presented. The contributions in this chapter includes:

- An extension to the Iris base logic with a new modality: the nextgen modality. This modality makes it possible to make non-frame-preserving changes to resources in Iris which was previously not possible.
- An extension to the Iris implementation in Coq with the new modality.
- A program logic for a language StackLang which contains a call-stack and stack allocated values. The nextgen modality is used in the proof rule for returns and the result is a proof rule that is simple and easy to use.
- The program logic and examples using it are formalized in Coq.

Chapter 6

This chapter introduces a more advanced nextgen modality, intended for Spirea. The contributions in this chapter includes:

- An explanation of why the post-crash modality used in Spirea in Chapter 4 is insufficient for proving stronger specifications for durable data-structures.
- A more powerful nextgen modality that ties the generational transformation to separation logic resources.
- A new model for BaseSpirea that uses the nextgen modality towards achieving an improved version of the logic that reaps the benefit of the nextgen modality.

1.4.3 Statement of Personal Contributions

Below I list the correspondence between each chapter in Part II of this thesis and the mentioned papers and manuscripts. Additionally, I list my own personal contributions to each of the chapters and the respective paper and projects they relate to.

- Chapter 2 is a verbatim inclusion of Vindum and Birkedal [VB21]. For this paper I am the main author with my co-author PhD Lars Birkedal serving an advisory role. I carried out the research, in particular the proof and its mechanization in Coq. I wrote the paper with feedback and editing by my co-author.
- Chapter 3 is a verbatim inclusion of Vindum et al. [VFB22]. I am the main author of the paper, with Dan Frumin serving a minor role, and Lars Birkedal an advisory role. I lead the research, in particular I verified the data-structure and created the extension to ReLoC. I mechanized the proof in Coq with minor

help from Dan. Dan translated the C++ implementation of the MPMC queue in Folly into HeapLang and mechanized the extension to ReLoC in Coq with minor suggestions from me. I lead the writing of the paper and wrote the majority of the paper. Lars wrote parts of Section 3.1 and Dan wrote parts of Section 3.7. Both co-authors provided feedback and editing of the paper.

- Chapter 4 is a based on of Vindum and Birkedal [VB23c]. The text is unchanged except that I have integrated the appendices from the original publications into the main text and included Figure 4.6 and Figure 4.9 that did not fit with the space requirements of the published paper. For this paper I am the main author with my co-author Lars Birkedal serving an advisory role. I carried out the research and the mechanization effort. I wrote the paper with feedback and editing by my co-author.
- Chapter 5 is unchanged compared to above-mentioned manuscript. I lead the research leading to the created of the nextgen modality with advice from Lars. I mechanized the modality in Coq. Aïna Linn Georges lead the research regarding the language with stack allocation described in Section 5.4 and mechanized it in Coq. I lead the writing of the paper except for Section 5.4 which was written by Aïna. All co-authors provided feedback and editing of the paper.
- Chapter 6 is not based on a publication. It is part of a larger ongoing project carried out together with Yixuan Chen and Lars Birkedal. The part of the project covered in the chapter was lead by me (with other parts, not mentioned in the chapter, being lead by Yixuan). I carried out the research and mechanization with feedback from Yixuan and Lars. The text in the chapter is written by me.

Part II

Publications

Chapter 2

Contextual Refinement of the Michael-Scott Queue

Abstract

The Michael-Scott queue (MS-queue) is a concurrent non-blocking queue. In an earlier pen-and-paper proof it was shown that a simplified variant of the MS-queue contextually refines a coarse-grained queue. Here we use the Iris and ReLoC logics to show, for the first time, that the original MS-queue contextually refines a coarse-grained queue. We make crucial use of the recently introduced prophecy variables of Iris and ReLoC. Our proof uses a fairly simple invariant that relies on encoding which nodes in the MS-queue can reach other nodes. To further simplify the proof, we extend separation logic with a generally applicable *persistent points-to predicate* for representing immutable pointers. This relies on a generalization of the well-known algebra of fractional permissions into one of *discardable* fractional permissions. We define the persistent points-to predicate entirely inside the base logic of Iris (thus getting soundness "for free").

We use the same approach to prove refinement for a variant of the MS-queue resembling the one used in the `java.util.concurrent` library.

We have mechanized our proofs in Coq using the formalizations of ReLoC and Iris in Coq.

2.1 Introduction

The Michael-Scott queue (MS-queue) is a fast and practical fine-grained concurrent queue [MS96b]. We prove that the MS-queue is a *contextual refinement* of a coarse-grained concurrent queue. The coarse-grained queue, shown in Figure 2.1, is implemented as a reference to a functional list and uses a lock to sequentialize concurrent accesses to the queue. We thus prove that in any program we may replace uses of the coarse-grained, but obviously correct, concurrent queue with the faster, but more intricate, MS-queue, without changing the observable behaviour of the program. We recall that, formally, an expression e contextually refines another

```
dequeue_CG lock list () ≜
  sync (lock) {
    match !list with
      nil ⇒ none
      x :: xs ⇒ list ← xs; some x }

enqueue_CG lock list x ≜ sync (lock) { list ← (!list ++ [x]) }

queue_CG ≜ Λ.
  let lock = newlock ()
      list = ref nil in
  (λ_. dequeue_CG lock list (), λx. enqueue_CG lock list x)
```

Figure 2.1: The coarse-grained queue.

expression e', denoted $\Delta; \Gamma \vdash e \precsim_{ctx} e' : \tau$, if for all contexts K, of ground type, whenever $K[e]$ terminates with a value there exists an execution of $K[e']$ that terminates with the same value. One should think of e as the *implementation* (in our case the MS-queue), e' as the *specification* (in our case the coarse-grained queue), and K as a *client* of a queue implementation.

Note that the contextual refinement implies that the internal states of the two queues are *encapsulated* and hidden from clients who could otherwise tell the difference between the two implementations. Contextual refinement is also related to *linearizability*, a popular correctness criterion considered for concurrent data structures. Linearizability has mostly been considered for first-order programming languages (without higher-order functions and abstract types). For a particular first-order language and under strong assumptions, Filipovic *et al.* [Fil+10a] showed that linearizability and contextual refinement coincide. Recently, Murawski and Tzevelekos [MT19] proposed a notion of linearizability for a programming language with higher-order functions, and they also proved that their notion of linearizability is *sound*, that is, that it implies contextual refinement. To the best of our knowledge, no sound notion of linearizability has been developed for the very rich programming language we consider (with higher types, abstract types, general references, and fork-based concurrency), so instead of using linearizability, we follow the approach of Turon et. al., and show contextual refinement directly [Tur+13b].

Turon et. al. showed how the proof technique of *logical relations* can be used to prove contextual refinement of fine-grained concurrent data structures [Tur+13b]. They also gave pen-and-paper proofs of contextual refinement for a simplified variant of the MS-queue. Here we present a mechanized proof of contextual refinement for the original MS-queue. This is more challenging, since proving refinement for it requires, among other things, the use of prophecy variables. The implementation of the MS-queue for which we prove refinement is *faithful* to the original, in the sense that we do not simplify or change it.

To carry out the proof we use ReLoC [FKB20b], a logic for reasoning about

contextual refinement defined on top of Iris, a state-of-the-art higher-order concurrent separation logic framework [Jun+18a]. Our mechanization uses the Coq implementations of ReLoC and Iris and the proof mode for Iris [Kre+18; KTB17a].

A key insight in our proof is to use a notion of *reachability* as a unifying concept that concisely captures both the roles of the nodes in the MS-queue, the protocol for how the queue may be modified, and the invariants that the queue maintain. This is arguably simpler than the approach used in [Tur+13b].

Like many data structures, the MS-queue contains locations that are never mutated after a certain point. To further simplify our proof we thus extend separation logic, in particular Iris, with better support for reasoning about locations that never change, by representing them as *immutable* pointers in the logic. To explain what this means at a high level, recall the points-to predicate $\ell \hookrightarrow v$, which has been present in separation logic since its inception for reasoning about shared *mutable* state [Rey02]. The points-to predicate denotes ownership over location ℓ and the knowledge that ℓ points to the value v. It has been generalized to the *fractional* points-to predicate $\ell \hookrightarrow^q v$ where one can own a fraction, $q \in (0, 1] \cap \mathbb{Q}$, of a points-to predicate [Bor+05; Boy03]. Changing a pointer is only possible when $q = 1$, whereas reading a location is possible with any fraction. This makes it possible to split access to a location and later reassemble it for *further mutation*. One can existentially quantify over the fraction ($\exists q.\ell \hookrightarrow^q v$) which makes it impossible to reassemble the entire fraction. This predicate, however, is only *duplicable* whereas we seek a predicate that is *persistent*—a strictly stronger notion [BB18]. Hence neither of these existing points-to predicates gives a satisfying way to reason about locations that arrive at a final value, after which they never change. To support reasoning about such locations, we generalize the points-to predicate further and introduce a *persistent* points-to predicate, $\ell \hookrightarrow^\square v$. In contrast to the beforementioned points-to predicates, our new persistent points-to predicate does not represent ownership over a resource; it only denotes the *knowledge* that ℓ always points to v. Since this predicate is persistent in the Iris-technical sense, it satisfies additional properties in comparison to the standard (fractional) points-to predicate and reasoning about immutable locations therefore becomes simpler when this predicate is used. We show that one can obtain a persistent points-to predicate by generalizing the notion of fractional permissions to one that allows *discarding* a fraction. One can then discard a fraction of the fractional points-to predicate and obtain a persistent points-to predicate; intuitively this makes sense since changing a location requires the entire fraction of the points-to predicate.

In summary, we make the following contributions:

- We show how the invariants maintained by the MS-queue can be expressed in a simple and unifying way by a notion of *reachability*.
- We show that a faithful implementation of the original MS-queue contextually refines a coarse-grained queue.

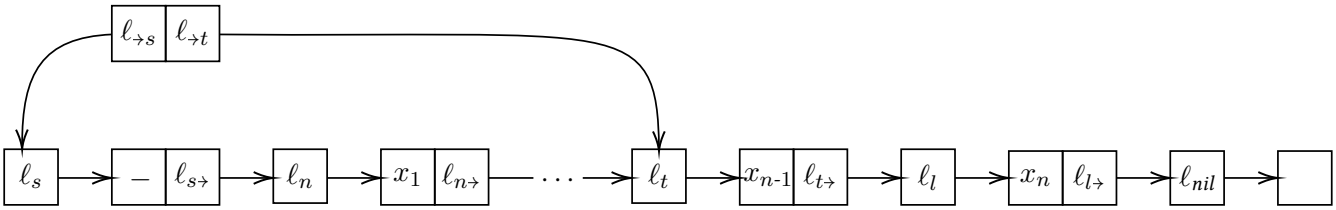


Figure 2.2: The MS-queue consists of a singly linked list. Here the tail pointer is lagging as it points to the second to last node.

- We extend separation logic (Iris and ReLoC in particular) with a persistent points-to predicate and demonstrate how it simplifies reasoning about the MS-queue.
- We show how the persistent points-to predicate and its associated proof rules can be defined and proven entirely inside the Iris base logic.
- To define the persistent points-to predicate we construct two novel resource algebras. The resource algebra of *discardable fractions*, which generalizes the well-known notion of fractions in separation logic, and the authoritative resource algebra with projections.
- Based on our formal proof, we discover that the use of *consistent snapshots* in the MS-queue is not necessary for the correctness of the algorithm in a garbage collected language.
- Finally, we use the same approach based on reachability to prove refinement for a variant of the MS-queue resembling the one used in the `java.util.concurrent` library.

All our results are formalized in Coq and we have extended the Coq implementation of Iris and ReLoC to support the persistent points-to predicate [VB20].

Outline We explain the fine-grained MS-queue algorithm and its implementation in Section 2.2 and then proceed to describe the structure of a refinement proof in ReLoC in Section 2.3, where we also present the coarse-grained queue that serves as a specification. The persistent points-to predicate and its proof rules are introduced in Section 2.4. Here we also further motivate why we seek a points-to predicate that is persistent and not merely duplicable. In Section 2.5 we detail the key ideas of the refinement proof and the invariant used. In Section 3.5 we present the actual refinement proof. In Section 2.7 we observe that the so-called consistent snapshots used in the MS-queue can be omitted without compromising the correctness of the algorithm, and in Section 2.8 we quickly comment on how we have used the same proof technique to prove refinement for a variant of the MS-queue. Finally, in Section 2.9 we detail how the persistent points-to predicate and its properties are actually defined and proved in the Iris base logic, by introducing two novel resource

$$
\begin{aligned}
\tau &::= \alpha \mid 1 \mid \mathsf{bool} \mid \mathsf{int} \mid \tau \times \tau \mid \tau + \tau \mid \tau \to \tau \\
&\mid \forall\alpha.\tau \mid \exists\alpha.\tau \mid \mu\alpha.\tau \mid \mathsf{ref}\ \tau \\
v &::= i \in \mathbb{Z} \mid \ell \in \mathit{Loc} \mid \mathbf{true} \mid \mathbf{false} \mid (v, v) \mid \mathbf{inj}_1\, v \mid \mathbf{inj}_2\, v \\
&\mid \mathbf{rec}\, f(x) = e \mid \Lambda.e \mid \mathbf{pack}\, v \mid \mathbf{fold}\, v \\
e &::= x \mid v \mid \mathbf{if}\, e\, \mathbf{then}\, e\, \mathbf{else}\, e \mid (e, e) \mid \pi_1\, e \mid \pi_2\, e \mid \mathbf{inj}_1\, e \mid \mathbf{inj}_2\, e \\
&\mid \mathbf{match}\, e\, \mathbf{with}\, \mathbf{inj}_1\, x \Rightarrow e \mid \mathbf{inj}_2\, x \Rightarrow e \mid e\, e \mid e\langle\rangle \\
&\mid \mathbf{pack}\, e \mid \mathbf{unpack}\, e\, \mathbf{in}\ x.e \mid \mathbf{fold}\, e \mid \mathbf{unfold}\, e \\
&\mid \mathbf{ref}\, e \mid\ !e \mid e \leftarrow e \mid \mathbf{CAS}(e, e, e) \mid \mathbf{fork}\ \{e\} \mid \ldots
\end{aligned}
$$

Syntactic sugar

$$
\mathsf{Option}\ \tau \triangleq 1 + \tau \quad \mathsf{none} \triangleq \mathbf{inj}_1\, 1 \quad \mathsf{some}\ v \triangleq \mathbf{inj}_2\, v
$$

$$
\lambda x.\, e \triangleq \mathbf{rec}\ \_\ x = e \quad \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \triangleq (\lambda x.\, e_2)\, e_2
$$

Figure 2.3: Syntax of the types and terms of HeapLang.

algebras. While we do recall the notion of a resource algebra, some familiarity with the Iris notion of resource algebras is probably needed to understand the details of (only) this section. We end by discussing related work in Section 3.8.

2.2 The MS-Queue

As depicted in Figure 2.2, the MS-queue consists of a singly linked list that contains the values ($x_1, \ldots, x_n$ in the figure) in the queue. The first node (ℓ_s) is called the *sentinel* and its content is not a value in the queue. The queue maintains two pointers, the *sentinel pointer* ($\ell_{\to s}$), which points to the sentinel, and the *tail pointer* ($\ell_{\to t}$), which points to the *tail* (ℓ_t). The tail is either equal to the *last node* (ℓ_l) or the second to last node. In the latter case, we say that the tail pointer is *lagging behind*. Note that $\ell_t = \ell_l$ when the tail pointer is *not* lagging behind.

We adopt the following naming convention: If ℓ_n is a location representing a node, then a location pointing *into* that node is denoted $\ell_{\to n}$ and the location pointing *out* from that node to the next node is denoted $\ell_{n\to}$. If ℓ_n is a node and ℓ_m its successor, then the pointer between the nodes can be denoted both $\ell_{n\to}$ or $\ell_{\to m}$ depending on the circumstances.

The implementation of the MS-queue is shown in Figure 2.7. It is written in HeapLang, a language included in the mechanization of Iris and which ReLoC extends with a type system to facilitate refinement proofs. The syntax of the language is presented in Figure 4.1, it is a λ-calculus with impredicative polymorphism, iso-recursive types, higher-order store, and thread-based concurrency. The language and its type system are standard; further details can be found in [FKB20b].

We have kept our implementation as faithful as possible to the original implementation. In order to emphasize this, we have annotated the code with line

numbers in direct correspondence with the line numbers in Michael and Scott's original code [MS96b]. All differences are minor and stem from inherent differences between HeapLang and the C-like language used in the original.

Initialization The $\mathsf{queue}_{\mathrm{MS}}$ function is the constructor for the queue and the entry point to the implementation. It uses a type abstraction, Λ, such that the queue is generic in the type of elements that it stores. This lambda also serves to ensure that the internal state of the queue is encapsulated in a closure. The initialization allocates an initial node, a sentinel pointer, and a tail pointer. The latter two points to the initial node. A newly constructed queue is illustrated in Figure 2.4.

A node is a pointer to either none or some of a pair of a value and a pointer to the next node. The pointer serves to make nodes comparable by pointer equality such that pointers to nodes can be changed with **CAS**.

Since there is no value to put in the initial sentinel, which $\mathsf{queue}_{\mathrm{MS}}$ must construct, none is used. All other nodes contain an actual value v and hence contains some v. Thus we often need to get the value of an Option which is known to be a some. This is the purpose of the getValue function.

Dequeue Dequeue reads the sentinel pointer and then the pointer to the sentinel's successor. If no successor exists the queue is empty and none is returned. If a succeeding node is found, dequeue attempts to change the sentinel pointer to the succeeding node with **CAS**. If the **CAS** is successful, the value in the new sentinel is returned. If the **CAS** is unsuccessful the operation is restarted. Figure 2.5 shows how successfully dequeuing an element from a non-empty queue swings the sentinel pointer forward.

The implementation contains prophecy annotations on line D4b and D5. These do not affect the execution of the program and can be ignored for now.

Enqueue Enqueue constructs a new node with the value that is to be enqueued. It then reads the tail pointer and obtains a node that may be the last. To determine if it is, enqueue checks whether or not the node has a successor. If a successor exists the tail pointer is lagging behind, and enqueue attempts to move the tail pointer forward with a **CAS** after which it restarts. If no successor exists then the node is currently the last. By means of a **CAS** enqueue then attempts to change the outgoing pointer of the node such that it points to the new node. If the **CAS** is successful, the tail pointer now lags behind, and enqueue attempts to advance the tail pointer to the new node. If, on the other hand, the **CAS** is unsuccessful, the operation restarts, and the tail pointer is read anew. Figure 2.6 illustrates how a successful enqueue inserts a new node and then swings the tail pointer forward.

Highlights We highlight a few aspects of the MS-queue that are of particular interest in terms of the verification.

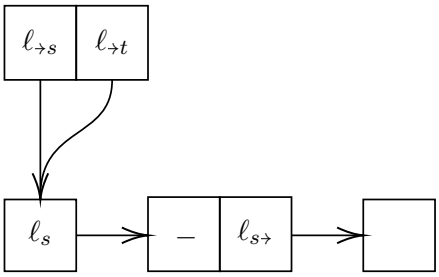


Figure 2.4: A newly constructed queue.

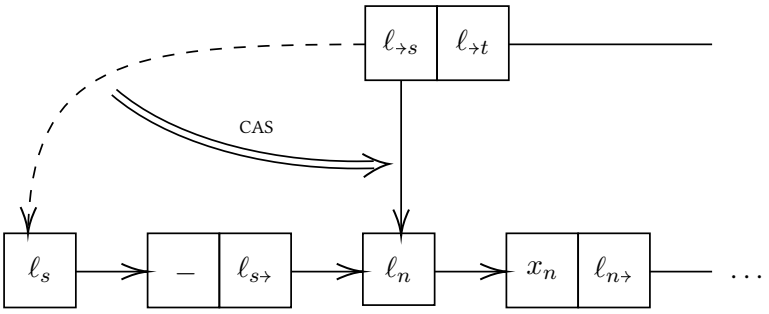


Figure 2.5: dequeue on the MS-queue.

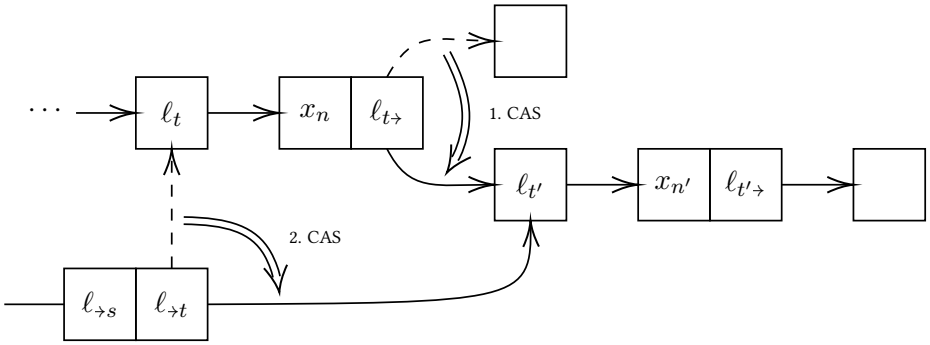


Figure 2.6: enqueue on the MS-queue.

On D6 the sentinel and tail are compared to each other. This is a rather indirect way of checking whether or not the queue is empty. If they are equal the queue is either empty or the tail pointer lags behind. Otherwise, the **else** branch on line D13 assumes that the queue is guaranteed to be non-empty. In our proof, we must formalize why this assumption is correct.

On line D5, a so-called *consistent snapshot* is performed: the value of toSent read on line D2 is compared to a newly read value of toSent. This ensures that toSent has not changed in the meantime and is intended to ensure that the values of tail and next are consistent. Similarly, enqueue performs a consistent snapshot on line E7.

Line D7 checks whether the next node is none or not. If it is not, then the tail pointer is lagging behind because an unfinished enqueue operation has not yet

updated it. Dequeue then attempts to update the tail pointer on D10. Likewise, on E13 enqueue also detects a lagging tail and attempts to update it. These are instances of *helping*, a pattern where the execution of one operation helps another.

As we will see, a contextual refinement proof for a fine-grained concurrent data-structure involves finding its *linearization points*. It is fairly clear that enqueue's linearization point is the **CAS** on E9 and that dequeue has a linearization point on line D13. What is less obvious is that when dequeue finds the queue empty and returns none on D8, its linearization point is at the load on D4c. However, line D4c is only a linearization point if next points to none *and* if the consistent snapshot on the *next line* succeeds. Because of this, it was conjectured by Morten Krogh-Jespersen[1] that one would need some kind of prophecy variables to reason about this; and indeed, in our proof, to know whether or not the check on the next line succeeds we use the recently introduced *prophecy variables* of Iris and ReLoC.

2.3 Structure of a Refinement Proof

In this section, we describe how to carry out a refinement proof of a fine-grained concurrent data-structure such as the MS-queue using ReLoC. We first consider the ingredients that such a proof consists of.

Persistently modality Iris has a persistently modality $\Box$ and $\Box\, P$ means that P *always holds*. A proposition P is per definition *persistent* if $P \vdash \Box\, P$, *i.e.,* if one from P alone can show that P always holds. Therefore persistent propositions represent *knowledge*. Propositions that are not persistent are called *ephemeral*—they represent ownership over resources. To show a goal of the form $\Box\, P$ one can only use persistent assumptions (PERSISTENT-$\Box$ in Figure 2.8). The intuition being that to show that something always holds one can only depend on other facts that always hold.

Specification In a proof of refinement, the *specification* should be a simple implementation of the same *interface* that the *implementation* is intended to implement. As mentioned in the Introduction, our specification is a coarse-grained concurrent queue, implemented using a pointer to a functional list and where the operations are guarded by a lock, which is included in ReLoC. The official definition of the coarse-grained queue is given in Figure 2.9; the version shown in the Introduction used a modicum of syntactic sugar.

Refinement judgment To prove a contextual refinement ReLoC offers a *refinement judgment* $\models e_1 \precsim e_2 : \tau$ which denotes that e_1 refines e_2 at the type τ. The ReLoC soundness theorem states that if such a judgment holds inside the logic, then the corresponding contextual refinement holds in the surrounding meta-logic.

[1] When he attempted to verify the MS-queue in 2014 using the iCap logic, a precursor to Iris. Private communication.

```
        getValue x ≜ match x with none ⇒ () () | some v ⇒ v

1:      queue_MS ≜ Λ.
2:        let node = ref (some(none, (ref (ref none))))
3:            tail = ref node
4:            sent = ref node
5:        in (dequeue_MS sent tail, enqueue_MS tail)

D1:     dequeue_MS toSent toTail ≜ rec loop () =
D2:       let sent = !toSent
D3:           tail = !toTail
D4a:          toNext = π2 (getValue !sent)
D4b:          p = NewProph
D4c:          next = !toNext in
D5:       if sent = Resolve(!toSent, p, ()) then
D6:         if sent = tail then
D7:           match !next with
D8:             none ⇒ none
D10:            some _ ⇒ CAS toTail tail next; loop ()
D11:        else
D13:          if CAS toSent sent next
D14:          then some (getValue(π1(getValue !next)))
D15:          else loop ()
D16:      else loop ()

        enqueue_MS toTail x ≜
E1-E3:    let node = ref (some (some x, ref (ref none))) in
E4:       (rec loop() =
E5:         let tail = !toTail
E6a:            toNext = π2 (getValue !tail)
E6b:            next = !toNext in
E7:         if tail = !toTail then
E8:           match !next with
E9:             none ⇒ if CAS toNext next node
E17:                   then CAS toTail tail node; ()
E11:                   else loop ()
E13:            some _ ⇒ CAS toTail tail next; loop ()
E14:        else loop ()) ()
```

Figure 2.7: Implementation of the MS-queue in HeapLang.

$$\textsc{$\Box$-sep-and}\ \frac{\Box(P \land Q)}{\Box(P * Q)} \qquad \textsc{$\Box$-exists}\ \frac{\exists x.\, \Box P}{\Box\, \exists x.\, P} \qquad \textsc{persistent-$\Box$}\ \frac{P \text{ persistent} \qquad P \vdash Q}{P \vdash \Box Q} \qquad \textsc{inv-alloc}\ \frac{P}{\mathcal{E} \Rrightarrow_{\mathcal{E}} \boxed{P}^{\iota}}$$

$$\textsc{löb}\ \frac{Q \land \triangleright P \vdash P}{Q \vdash P}$$

Structural rules

$$\textsc{rel-return}\ \frac{[\![\tau]\!]_\Delta(v_1, v_2)}{\Delta \models v_1 \precsim v_2 : \tau} \qquad \textsc{rel-tlam}\ \frac{\forall R : \textsc{Val} \times \textsc{Val} \to \mathsf{iProp}.\ \Box\left([\alpha := R], \Delta \models e_1 \precsim e_2 : \tau\right)}{\Delta \models \Lambda.e_1 \precsim \Lambda.e_2 : \forall \alpha.\tau}$$

$$\textsc{rel-pair}\ \frac{\Delta \models e_1 \precsim e_2 : \tau \qquad \Delta \models e_1' \precsim e_2' : \sigma}{\Delta \models (e_1, e_1') \precsim (e_2, e_2') : \tau \times \sigma}$$

$$\textsc{rel-rec}\ \frac{\Box\left(\forall v_1, v_2.\, [\![\tau]\!]_\Delta(v_1, v_2) \mathrel{-\!\!*} \Delta \models (\mathbf{rec}\ f_1(x_1) = e_1)\ v_1 \precsim (\mathbf{rec}\ f_2(x_2) = e_2)\ v_2 : \sigma\right)}{\Delta \models (\mathbf{rec}\ f_1(x_1) = e_1) \precsim (\mathbf{rec}\ f_2(x_2) = e_2) : \tau \to \sigma}$$

Symbolic execution rules

$$\textsc{rel-pure-r}\ \frac{e_2 \overset{\mathsf{pure}}{\rightsquigarrow} e_2' \qquad \Delta \models_{\mathcal{E}} e_1 \precsim K[\,e_2'\,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\,e_2\,] : \tau}$$

$$\textsc{rel-load-r}\ \frac{\ell \hookrightarrow_{\mathsf{s}} v \qquad \ell \hookrightarrow_{\mathsf{s}} v \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} e_1 \precsim K[\,v\,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\,!\ell\,] : \tau}$$

$$\textsc{rel-store-r}\ \frac{\ell \hookrightarrow_{\mathsf{s}} - \qquad \ell \hookrightarrow_{\mathsf{s}} v \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} e_1 \precsim K[\,()\,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\,\ell \leftarrow v\,] : \tau}$$

Rules for prophecy variables

$$\textsc{rel-newproph-l}\ \frac{\forall v, p.\, \mathsf{Proph}_1(p, v) \mathrel{-\!\!*} \Delta \models K[\,p\,] \precsim e_2 : \tau}{\Delta \models K[\,\mathsf{NewProph}\,] \precsim e_2 : \tau}$$

$$\textsc{rel-resolveproph-l}\ \frac{\mathsf{Proph}_1(p, v) \qquad \mathsf{wp}\ e\ \{u.\, v = (u, w) \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} K[\,v\,] \precsim e_2 : \tau\}}{\Delta \models K[\,\mathit{Resolve}(e, p, w)\,] \precsim e_2 : \tau}$$

Figure 2.8: Selected rules from ReLoC (some are simplified for the sake of presentation).

```
dequeue'_CG list ≜
  match !list with
    none ⇒ none
    some p ⇒ list ← (π2 p); some (π1 p)
dequeue_CG lock list () ≜
  acquire lock; let v = dequeue'_CG list in release lock; v

enqueue'_CG ≜ rec loop x list =
  match list with
    none ⇒ some (x, none)
    some p ⇒ some (π1 p, loop x (π2 p))
enqueue_CG lock list x ≜
  acquire lock; list ← enqueue'_CG x !list; release lock

queue_CG ≜ Λ.
  let lock = newlock ()
      list = ref none in
  (λ_. dequeue_CG lock list (), λx. enqueue_CG lock list x)
```

Figure 2.9: Implementation of the coarse-grained queue.

ReLoC provides high-level rules for working with these refinement judgments that result in simpler proofs than other approaches (*e.g.*,directly using logical relations). The *structural rules* apply when each side of the refinement is of the same syntactic form—it then suffices to show refinement of the sub-expressions that constitute the constructions. One such rule is REL-PAIR, which states that to show that two pairs are related it suffices to show that they are pair-wise related. Note that to show that two functions are related, using REL-REC, one must do so persistently, that is, without relying on any ephemeral resources. This is because a context could call a function an arbitrary number of times, and thus the functions must *always* be related at any point in the future.

When the two sides of the refinement are not of the same syntactic form, one must use *symbolic execution rules* to step either side forward. Note that the i and *s* in the points-to predicates denote if they are for the implementation or the specification.

Invariants As mentioned, to show that two functions are related one can only use persistent propositions. Non-persistent propositions can be made persistent by establishing an *invariant* using the rule INV-ALLOC. The proposition $\boxed{P}^{\iota}$ denotes knowledge of an invariant with the name ι and is persistent even if P is not. During a refinement proof, one can *open* an invariant around a single atomic expression e on the left-hand side. The contents of the invariant can be used to symbolically execute e, but, afterward it is an obligation to close the invariant by showing that it still holds. Crucially this restriction does not apply to the right-hand side, here it is allowed to take several steps of symbolic execution with an invariant open. The

$$\mathrm{I}_{\mathrm{CG}}(\ell_{cg}, lk, xs) \triangleq \ell_{cg} \hookrightarrow_{\mathsf{s}} \mathsf{isList}(xs) * \mathsf{isLocked}(lk, \mathsf{False})$$
$$\mathsf{isList}([]) \triangleq \mathsf{none}$$
$$\mathsf{isList}(x :: xs) \triangleq \mathsf{some}\ (x, \mathsf{isList}(xs))$$

$$\frac{\mathrm{I}_{\mathrm{CG}}(\ell_{cg}, lk, []) \qquad \mathrm{I}_{\mathrm{CG}}(\ell_{cg}, lk, []) \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} e_1 \precsim K[\,\mathsf{none}\,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\,\mathsf{dequeue}_{\mathrm{CG}}\ lk\ \ell_{cg}\ ()\,] : \tau}\ \textsc{dequeue}_{\mathrm{CG}}\textsc{-nil-r}$$

$$\frac{\mathrm{I}_{\mathrm{CG}}(\ell_{cg}, lk, x :: xs) \qquad \mathrm{I}_{\mathrm{CG}}(\ell_{cg}, lk, xs) \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} e_1 \precsim K[\,\mathsf{some}\ x\,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\,\mathsf{dequeue}_{\mathrm{CG}}\ lk\ \ell_{cg}\ ()\,] : \tau}\ \textsc{dequeue}_{\mathrm{CG}}\textsc{-cons-r}$$

$$\frac{\mathrm{I}_{\mathrm{CG}}(\ell_{cg}, lk, xs) \qquad \mathrm{I}_{\mathrm{CG}}(\ell_{cg}, lk, xs +\!\!+ [x]) \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} e_1 \precsim K[\,()\,] : \tau}{\Delta \models_{\mathcal{E}} e_1 \precsim K[\,\mathsf{enqueue}_{\mathrm{CG}}\ lk\ \ell_{cg}\ x\,] : \tau}\ \textsc{enqueue}_{\mathrm{CG}}\textsc{-r}$$

Figure 2.10: Right-hand side relational specification for the coarse-grained queue.

way the above restrictions are enforced is rather technical, so we omit the details, but note that the modality $\Rightarrow\!\!\!\!\!\Rightarrow$ is used to denote when invariants can be opened.

Linearization points During a refinement proof, one must maintain a link between the state of the implementation and the specification such that upon termination one can show that the two values are related. For a fine-grained concurrent data-structure, such as the MS-queue, operations "take effect" at specific points, namely the linearization points. At these points, the specification should be symbolically executed from start to end; this is possible even while an invariant is open per the above. To this end we use the rules for the coarse-grained queue shown in Figure 2.10; these are easy to prove using the lock specification that ReLoC includes, and our definition of the *representation predicate* I_{CG} for the coarse-grained queue, also shown in the figure. The representation predicate states that the physical state of the coarse-grained queue (the pointer to a list and the lock) corresponds to a logic-level sequence.

Prophecy variables For the MS-queue in particular we also need *prophecy variables*. These are a recent addition to Iris and ReLoC [FKB20b; Jun+20b]. Recall how the load at D4c may be a linearization point depending on the result of the load on the next line, D5. Hence, when we symbolically execute the load at D4c we need to know the result of a *future* expression. This is what prophecy variables make possible. They rely on code annotations, which do not affect the execution of the program but aids in reasoning. A prophecy is created with NewProph and

$$\textsc{Mapsto-intro-}\Box\ \frac{\ell \hookrightarrow_{\mathsf{i}}^{q} v}{\Rightarrow\!\!\!\!\Rightarrow \ell \hookrightarrow_{\mathsf{i}}^{\Box} v} \qquad \textsc{Mapsto-agree-}\Box\ \frac{\ell \hookrightarrow_{\mathsf{i}}^{\Box} v \qquad \ell \hookrightarrow_{\mathsf{i}}^{\Box} v'}{v = v'} \qquad \textsc{persistent}\ \frac{\ell \hookrightarrow_{\mathsf{i}}^{\Box} v}{\Box\, \ell \hookrightarrow_{\mathsf{i}}^{\Box} v}$$

$$\textsc{Ht-load-}\Box\ \frac{}{\{\ell \hookrightarrow_{\mathsf{i}}^{\Box} u\}\, !\ell \{v.v = u\}} \qquad \textsc{rel-load-r-}\Box\ \frac{\ell \hookrightarrow_{\mathsf{i}}^{\Box} v \qquad \Delta \models K[\,v\,] \precsim e_2 : \tau}{\Delta \models K[\,!\,\ell\,] \precsim e_2 : \tau}$$

$$\textsc{rel-cas-l}\ \frac{{}_{\top}\!\Rightarrow\!\!\!\!\Rightarrow_{\mathcal{E}} \exists v. \begin{pmatrix} \big(v \neq v_1 \mathrel{-\!\!*} \\ \quad (\ell \hookrightarrow_{\mathsf{i}}^{\Box} v * \Delta \models_{\mathcal{E}} K[\,\mathbf{false}\,] \precsim e_2 : \tau) \vee \\ \quad \exists q.\, (\ell \hookrightarrow_{\mathsf{i}}^{q} v * \\ \qquad (\ell \hookrightarrow_{\mathsf{i}}^{q} v \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} K[\,\mathbf{false}\,] \precsim e_2 : \tau))\big) \wedge \\ \big(v = v_1 \mathrel{-\!\!*} \\ \quad (\ell \hookrightarrow_{\mathsf{i}} v * (\ell \hookrightarrow_{\mathsf{i}} v_2 \mathrel{-\!\!*} \Delta \models_{\mathcal{E}} K[\,\mathbf{true}\,] \precsim e_2 : \tau))\big) \end{pmatrix}}{\Delta \models K[\,\mathbf{CAS}(\ell, v_1, v_2)\,] \precsim e_2 : \tau}$$

Figure 2.11: Rules for the persistent points-to predicate.

per REL-NEWPROPH-L it results in a resource $\mathsf{Proph}_1(p, v)$ where p is the name of the prophecy and v is a value. Intuitively, v is equal to the value which the prophecy is eventually resolved to. A prophecy is resolved with an *atomic prophecy resolution*: $\mathit{Resolve}(e, p, w)$. This expression behaves computationally exactly as the atomic expression e. Its rule REL-RESOLVEPROPH-L requires $\mathsf{Proph}_1(p, v)$, and hence one can think of this resource as giving one the right to resolve the prophecy. It then states that v is equal to (u, w) where u is that value that e evaluates to. In our case we create a prophecy at D4b, hence at this point we get a value v that can be thought of as the result of the future expression !toSent.

Given these ingredients, the overall structure of a refinement proof is: *(a)* Decide on a specification and prove right-hand side lemmas for each operation (Figure 2.10 in our case). *(b)* Define an invariant that relates the state of the specification to that of the implementation (Section 2.5) *(c)* Use symbolic execution rules to step through the initialization of each side. *(d)* Establish the invariant and use structural rules to get the goals to show that each operation is related. *(e)* Show that each operation is related by using the invariant; at each linearization point apply the corresponding lemma for the specification.

2.4 Persistent Points-To Predicate

Consider the depiction of the MS-queue in Figure 2.2 on page 22. All the pointers, except $\ell_{\to s}$, $\ell_{\to t}$, and $\ell_{l\to}$, are never changed, and, once $\ell_{l\to}$ is changed it is never changed again. As we will see, expressing precisely which parts of the MS-queue

change, and which do not, is central to our approach. Since data-structures with locations that are or become immutable are common, it makes sense to develop a generally applicable tool for reasoning about immutable pointers. To this end, we introduce the *persistent points-to predicate*, denoted $\ell \hookrightarrow_{\mathsf{i}}^{\square} v$ as mentioned in the Introduction. In contrast to the normal points-to predicate, which allows for mutation but no sharing, the persistent points-to predicate allows for free sharing but no mutation.

The reader may wonder whether there is an already existing alternative to a new persistent points-to predicate. Perhaps $\exists q.\, \ell \hookrightarrow_{\mathsf{i}}^{q} v$? This predicate, however, is only duplicable whereas we want a points-to predicate that is persistent. This is because persistence is a strictly stronger notion and persistent propositions enjoy additional properties. The persistent modality commutes with all the logical connectives (*e.g.*,□-EXISTS) and under it conjunction and separating conjunction coincides (□-SEP-AND). Hence persistent propositions form a sublogic with non-substructural properties. This is not the case for duplicable propositions: for instance, $\ell \hookrightarrow v$ is *not* duplicable but $\exists q.\ell \hookrightarrow^{q} v$ is. Persistent propositions are utilized to great effect in the Coq mechanization of Iris, see [KTB17a].

Maybe one could remedy this issue by wrapping the existentially quantified fractional points-to predicate in an invariant, that is, use $\boxed{\exists q.\ell \hookrightarrow^{q} v}^{\iota}$? This would result in a persistent predicate, but, we want a persistent points-to predicate that can be used as a normal points-to predicate, including being put inside invariants, and with this definition, we would be led to nested invariants. And while Iris does support nested invariants, reasoning about such would involve the later modality and, as a result, it would make the use of the persistent points-to predicates more restrictive.

Other approaches to modeling immutable locations exist, e.g., one may use a combination of invariants and additional ghost state, as done in [KTB17a], but this approach is more complex and our points-to predicate would have simplified the proofs in [KTB17a].

A selection of the rules for the persistent points-to predicate is shown in Figure 2.11. Since the persistent points-to predicate represents locations that never change, it is persistent (PERSISTENT). Given *any fraction* of a normal points-to predicate, one can obtain a persistent points-to predicate (MAPSTO-INTRO-□)—one can think of the fractional points-to predicate as being discarded in exchange for a persistent points-to predicate. The modality $\Rightarrow$ is there because discarding the fraction requires updating ghost state. Persistent points-to predicates for the same location must point to the same value (MAPSTO-AGREE-□). Finally, the predicate can be used for read-only operations, such as loading a pointer (HT-LOAD-□).

In Section 2.9 we show how to define the persistent points-to predicate and derive its rules entirely *within the Iris base logic*. This automatically guarantees soundness of the rules. We have additionally extended the Coq formalization of Iris and ReLoC to support the persistent points-to predicate as seamlessly as they support the normal points-to predicate. Among other things, this means that the tactics in the proof mode automatically use the persistent points-to predicate when

possible.

The last rule in Figure 2.11, REL-CAS-L, is an improved version of a corresponding rule in ReLoC [FKB20b]. It now allows using the persistent points-to predicate to show that a failed **CAS** is safe. This makes sense since it is sufficient to have read-only access to a location as long as one is not actually successful in mutating it. The other change to the rule is in the ordering of connectives. This change is subtle but makes the rule more complete. The original rule for **CAS** in ReLoC is structured as

$$\exists v.\, \ell \hookrightarrow_{\mathsf{i}} v * ((v \neq v_1 \mathrel{-\!\!*} \dots) \lor (v = v_1 \mathrel{-\!\!*} \dots))$$

whereas our rule allows one to first offer a witness v, then assume either $v = v_1$ or $v \neq v_1$, and then *use this (in)equality* to show the points-to predicate. This turns out to be essential in the proof of refinement of enqueue.

2.5 Invariant for the Refinement Proof

We now present the invariant used in the refinement proof.

2.5.1 Reachability

A key insight of our approach is how the invariants that the MS-queue maintains can be expressed in terms of which nodes are *reachable* from other nodes. Reachability is expressed with an inductive predicate:

$$\begin{aligned}\ell_n \rightsquigarrow \ell_m \triangleq \exists \ell_{n\rightarrow}, v.\, \ell_n \hookrightarrow_{\mathsf{i}}^{\square} \mathsf{some}(v, \ell_{n\rightarrow}) * \\ (\ell_n = \ell_m \lor \exists \ell_p.\, \ell_{n\rightarrow} \hookrightarrow_{\mathsf{i}}^{\square} \ell_p * \ell_p \rightsquigarrow \ell_m)\end{aligned}$$

It is persistent as the definition uses the persistent points-to predicate to express that the sequence of nodes is immutable.

Reachability is a preorder on nodes in the sense that for all ℓ_n and ℓ_m:

$$\ell_n \hookrightarrow_{\mathsf{i}}^{\square} \mathsf{some}\ (v, \ell_{n\rightarrow}) \mathrel{-\!\!*\!\!*} \ell_n \rightsquigarrow \ell_n \qquad \text{(reachable-reflexive)}$$

$$\ell_n \rightsquigarrow \ell_m \mathrel{-\!\!*} \ell_m \rightsquigarrow \ell_o \mathrel{-\!\!*} \ell_n \rightsquigarrow \ell_o \qquad \text{(reachable-transitive)}$$

Note, that $\ell_n \rightsquigarrow \ell_n$ is not trivial, it implies that ℓ_n is actually a node, in the sense that it points to some of a pair. More generally, $\ell_n \rightsquigarrow \ell_m$ implies that both ℓ_n and ℓ_m are nodes.

2.5.2 Abstract Reachability

A crucial property of the MS-queue is that the sentinel and tail pointers are only moved *forward* to succeeding nodes. Additionally, the linked list is never mutated except when new nodes are added at the very end. This implies that if a node can reach the *current* sentinel, tail, or last node then it can reach any *future* sentinel, tail, or last node.

$$\textsc{abs-reach-alloc}\ \frac{}{\ell_n \rightsquigarrow \ell_n}\qquad \exists\gamma_n.\, \gamma_n \Mapsto \ell_n * \ell_n \dashrightarrow \gamma_n$$

$$\textsc{abs-reach-concr}\ \frac{\ell_n \dashrightarrow \gamma_m \qquad \gamma_m \Mapsto \ell_m}{\ell_n \rightsquigarrow \ell_m * \gamma_m \Mapsto \ell_m}$$

$$\textsc{abs-reach-abs}\ \frac{\ell_n \rightsquigarrow \ell_m \qquad \gamma_m \Mapsto \ell_m}{\Rightarrow (\ell_n \dashrightarrow \gamma_m * \gamma_m \Mapsto \ell_m)}$$

$$\textsc{abs-reach-advance}\ \frac{\gamma_m \Mapsto \ell_m \qquad \ell_m \rightsquigarrow \ell_o}{\Rightarrow (\gamma_m \Mapsto \ell_o * \ell_o \dashrightarrow \gamma_m)}$$

Figure 2.12: Rules for abstract reachability.

To model this we use three ghost variables, γ_s, γ_t, and γ_l, as *abstract nodes* that give fixed names to the idea of the "current" sentinel, tail, and last node respectively. We then introduce *abstract reachability*, $\ell_n \dashrightarrow \gamma_m$, capturing that the *physical* node ℓ_n can reach the *abstract* node γ_m. To realize this intention, our invariant will *tie* the three abstract nodes to the locations that are currently the sentinel, tail, and last nodes. This is done using a predicate $\gamma_n \Mapsto \ell_m$ representing that the abstract node γ_n is currently tied to the physical node ℓ_m.

These predicates satisfy the rules given in Figure 2.12. The first rule serve as an introduction rule for abstract reachability. The second and third rule state that given $\gamma_m \Mapsto \ell_m$ one can go from $\ell_n \rightsquigarrow \ell_m$ to $\ell_n \dashrightarrow \gamma_m$, and vice versa. The last rule makes it possible to change which physical node an abstract node is tied to as long as the new node is reachable from the current node.

For the reader familiar with Iris resource algebras we remark that the above can be realized using the resource algebra $\textsc{Auth}(\mathscr{P}(\textsc{Loc}))$ and the following definitions:

$$\ell_n \dashrightarrow \gamma_m \triangleq \boxed{\circ\{\ell_n\}}^{\gamma_m} \qquad \gamma_n \Mapsto \ell_n \triangleq \exists s.\, \boxed{\bullet s}^{\gamma_n} * \mathop{\Large *}_{\ell_m \in s} \ell_m \rightsquigarrow \ell_n$$

Here $\mathscr{P}(A)$ denotes the resource algebra of sets of A, with union as the operation, and the core being the identity function.

2.5.3 The Invariant

The top-level invariant in Figure 2.13 is parameterized by a value relation, τ_i, and the values that the implementation and specification consist of. It states the existence of two mathematical lists xs_i and xs_s that, through $\mathrm{I_{MS}}$ and $\mathrm{I_{CG}}$, are related to the physical representation of each queue. The big separating conjunction relates the lists pair-wise by τ_i. This way of relating the implementation and specification is arguably simpler than the approach used in [KTB17a; Tur+13b], which would have intermingled the physical representations of the two queues with the pair-wise relatedness of the elements in the queues.

$\mathrm{I_{CG}}$ is as previously seen and $\mathrm{I_{MS}}$ states the existence of ℓ_s, ℓ_t, and ℓ_l and ties the abstract nodes to these. It contains the points-to predicates for the three mutable locations in the queue. It states that the sentinel can reach the abstract tail: $\ell_s \dashrightarrow \gamma_t$.

Top-level invariant

$$I(\tau_i, \ell_{\to s}, \ell_{\to t}, \ell_{cg}, lk) \triangleq \exists xs_i, xs_s.\ \mathrm{I_{MS}}(\ell_{\to s}, \ell_{\to t}, xs_i) * \mathrm{I_{CG}}(\ell_{cg}, lk, xs_s)\ * \mathop{\Large *}_{(x_i,x_s)\in(xs_i,xs_s)} \tau_i(x_i, x_s)$$

Invariant for the MS-queue

$$\begin{aligned}\mathrm{I_{MS}}(\ell_{\to s}, \ell_{\to t}, xs_i) \triangleq\ & \exists \ell_s, \ell_{s\to}, \ell_t, \ell_{t\to}, \ell_l, \ell_{l\to}.\\ & \ell_{\to s} \hookrightarrow_{\mathsf{i}} \ell_s * \ell_{\to t} \hookrightarrow_{\mathsf{i}} \ell_t * \mathrm{isQueue_{MS}}(\ell_{l\to}, \ell_{s\to}, xs_i) *\\ & \gamma_s \Rrightarrow \ell_s * \ell_s \hookrightarrow_{\mathsf{i}}^{\square} \mathsf{some}\ (-, \ell_{s\to}) * \ell_s \dashrightarrow \gamma_t *\\ & \gamma_t \Rrightarrow \ell_t * \ell_t \hookrightarrow_{\mathsf{i}}^{\square} \mathsf{some}\ (-, \ell_{t\to}) * \ell_t \dashrightarrow \gamma_l *\\ & \gamma_l \Rrightarrow \ell_l * \ell_l \hookrightarrow_{\mathsf{i}}^{\square} \mathsf{some}\ (-, \ell_{l\to}) * \ell_{l\to} \hookrightarrow_{\mathsf{i}} \ell_n * \ell_n \hookrightarrow_{\mathsf{i}}^{\square} \mathsf{none}\end{aligned}$$

$$\mathrm{isQueue_{MS}}(\ell_{l\to}, \ell_{\to n}, [\,]) \triangleq \ell_{l\to} = \ell_{\to n}$$

$$\begin{aligned}\mathrm{isQueue_{MS}}(\ell_{l\to}, \ell_{\to n}, x :: xs) \triangleq\ & \exists \ell_n, \ell_{n\to}.\\ & \ell_{\to n} \hookrightarrow_{\mathsf{i}}^{\square} \ell_n * \ell_n \hookrightarrow_{\mathsf{i}}^{\square} \mathsf{some}\ (\mathsf{some}\ x, \ell_{n\to}) *\\ & \mathrm{isQueue_{MS}}(\ell_{l\to}, \ell_{n\to}, xs)\end{aligned}$$

Figure 2.13: The invariant and auxiliary definitions.

This knowledge is key to proving the **else** branch in dequeue starting on line D13, which we previously discussed. In fact, the reason why the check on D6 ensures that the queue is empty is exactly that the tail pointer can not fall behind the sentinel pointer. Additionally, $\ell_t \dashrightarrow \gamma_l$ ensures that the tail can reach the abstract last node. Finally, $\mathrm{isQueue_{MS}}$ relates the linked list to the mathematical list xs_{i}.

Note how the only non-persistent things in $\mathrm{I_{MS}}$ are the three points-to predicates and the resource tieing the abstract nodes to the physical nodes. Clearly, these can not be persistent. Hence, our invariant precisely captures and separates the changing parts of the MS-queue from the unchanging parts.

Before moving on to the refinement proof, we demonstrate how the invariant and abstract reachability is used by proving a lemma which is to be used whenever the MS-queue attempts to swing the tail pointer forward.

Lemma 2.5.1. *Swing tail pointer forward.*

$$\frac{\boxed{I(\dots)}^{\iota} \qquad \ell_n \rightsquigarrow \ell_m \qquad \forall v.\ \models K[\,v\,] \precsim e : \alpha}{\models K[\,\textbf{\textit{CAS}}\ \ell_{\to t}\ \ell_n\ \ell_m\,] \precsim e : \alpha}$$

Proof. We apply REL-CAS-L and open the invariant. Since the invariant contains $\ell_{\to t} \hookrightarrow \ell_t$ for some ℓ_t we offer the witness ℓ_t. If the **CAS** fails we can simply close the invariant again. If the **CAS** succeeds we know that $\ell_n = \ell_t$ and we now get

$\ell_{\to t} \hookrightarrow \ell_m$. When we close the invariant we supply ℓ_m as the witness for ℓ_t. To do that we have to show

$$\gamma_t \Rrightarrow \ell_m * \ell_m \hookrightarrow_{\mathsf{i}}^{\square} \mathsf{some}\ (-, \ell_{m\to}) * \ell_m \dashrightarrow \gamma_l$$

The middle conjunction follows from $\ell_n \rightsquigarrow \ell_m$. We have $\gamma_t \Rrightarrow \ell_n$ and $\ell_n \rightsquigarrow \ell_m$ which per the last rule in Figure 2.12 gets us the rest. □

2.6 Refinement Proof of the MS-Queue

We now prove that the MS-queue contextually refines the coarse-grained queue:

$$\models \mathsf{queue}_{\mathsf{MS}} \precsim \mathsf{queue}_{\mathsf{CG}} : \forall\alpha.(1 \to \mathsf{Option}\ \alpha) \times (\alpha \to 1)$$

Since both $\mathsf{queue}_{\mathsf{MS}}$ and $\mathsf{queue}_{\mathsf{CG}}$ are type abstractions we apply REL-TLAM to show that in a context extended with α interpreted using any value relation R. We symbolically execute the code on the left-hand side to the resources:

$$\ell_{nil} \hookrightarrow_{\mathsf{i}} \mathsf{none} * \ell_{s\to} \hookrightarrow_{\mathsf{i}} \ell_{nil} *$$
$$\ell_s \hookrightarrow_{\mathsf{i}} \mathsf{some}(\mathsf{none}, \ell_{s\to}) * \ell_{\to s} \hookrightarrow_{\mathsf{i}} \ell_s * \ell_{\to t} \hookrightarrow_{\mathsf{i}} \ell_s$$

From stepping through the right-hand side we get

$$\ell_{list} \hookrightarrow_{\mathsf{s}} \mathsf{none} * \mathsf{isLocked}(lk, \mathsf{False}).$$

Together with ABS-REACH-ALLOC this is enough to establish the invariant. We thus now have $\boxed{I(\tau_i, \ell_{\to s}, \ell_{\to t}, \ell_{cg}, lk)}^{\iota}$ in the context.

Both sides step to a pair and we apply the structural rule REL-PAIR. We are then required to show that the fine-grained dequeue and enqueue are logical refinements of their coarse-grained counterparts. We do this in the next two sections.

2.6.1 Dequeue

We are to show the logical refinement:

$$[\alpha := R] \models \mathsf{dequeue}_{\mathsf{MS}}\ \ell_{\to s}\ \ell_{\to t} \precsim \mathsf{dequeue}_{\mathsf{CG}}\ lk\ \ell_{CG} : 1 \to \mathsf{Option}\ \alpha.$$

Since both sides are functions we use REL-REC and have to show that for any two values v_1 and v_2, where $[\![1]\!]_\Delta(v_1, v_2)$, it is the case that the left-hand side applied to v_1 is related to the right-hand side applied to v_2. Since v_1 and v_2 are related at the type 1 they must both be equal to the unit value $()$. Hence we are to show

$$[\alpha := R] \models \mathsf{dequeue}_{\mathsf{MS}}\ \ell_{\to s}\ \ell_{\to t}\ () \precsim \mathsf{dequeue}_{\mathsf{CG}}\ lk\ \ell_{CG}\ () : \mathsf{Option}\ \alpha.$$

As the left-hand side is a recursive function we apply the LÖB rule. This gives us the induction hypothesis that the refinement holds for any recursive calls. We then

apply structural rules to symbolically execute the left implementation until we arrive at the first load:

$$\mathsf{sent} = !\ell_{\to s}$$

The yellow background indicates the expression currently being symbolically executed and which we open the invariant around. We open the invariant and from the points-to predicate for $\ell_{\to s}$ we know that the load steps to some ℓ_s and that we can assume the following persistent propositions for some $\ell_{s\to}$ and v:

$$\ell_s \hookrightarrow_{\mathsf{i}}^{\square} \mathsf{some}\,(v, \ell_{s\to}) * \ell_s \dashrightarrow \gamma_s * \ell_s \dashrightarrow \gamma_t * \ell_s \dashrightarrow \gamma_l \tag{2.1}$$

On the next line, the tail is loaded.

$$\mathsf{tail} = !\ell_{\to t}$$

By opening the invariant, we can conclude that the load evaluates to some ℓ_t. We know that ℓ_s can reach the current tail ($\ell_s \dashrightarrow \gamma_t$ in Equation (2.1)) and that ℓ_t is the current tail ($\gamma_t \Mapsto \ell_t$ from the invariant) hence per ABS-REACH-CONCR we get $\ell_s \rightsquigarrow \ell_t$.

On the next line (D4a) ℓ_s is read:

$$\mathsf{toNext} = \pi_2\ (\mathsf{getValue}\ !\ell_s)$$

We can evaluate this, without opening the invariant, using the points-to predicate from Equation (2.1). Thus, the load evaluates to some $(v, \ell_{s\to})$. With this information, we can symbolically execute the getValue and the projection.

We then arrive at the creation of the prophecy variable at line D4b. Using REL-NEWPROPH-L we get the prophecy assertion $\mathsf{Proph}_1(p, v)$. Since the prophecy variable is resolved with !toSent on line D5, the value v is, intuitively, equal to the result of that load. Hence, whether or not v is equal to ℓ_s, determines the outcome of the check on line D5. If they are equal, we will be able to show that the check succeeds, and otherwise, the check will fail. We consider these two cases separately. In the latter case, where $v \neq \ell_s$, dequeue restarts and we only have to show that the execution up to the recursive call on the last line is safe. This is straightforward so we consider only the first case where $v = \ell_s$.

We proceed to the next load on D4c:

$$\mathsf{next} = !\ell_{s\to}$$

This load reads the pointer *out* of the *sentinel*. Intuitively, if this leads to none then the queue must be empty and the pointer read is the mutable pointer that $\mathsf{enqueue}_{\mathsf{MS}}$ may modify. Hence, if this is the case, this is a linearization point and we must then conclude that the queue is empty.

To do this, we open the invariant and introduce the existentially quantified locations with the names ℓ_s, ℓ_t and ℓ_l as $\ell_{s'}, \ell_{t'}$ and $\ell_{l'}$ respectively. Using $\ell_s \dashrightarrow \gamma_s$ from Equation (2.1) and ABS-REACH-CONCR we can determine that ℓ_s can reach all these nodes:

$$\ell_s \rightsquigarrow \ell_{s'} * \ell_s \rightsquigarrow \ell_{t'} * \ell_s \rightsquigarrow \ell_{l'} \tag{2.2}$$

Since ℓ_s can reach $\ell_{l'}$ they are either equal or ℓ_s has a successor node which can reach $\ell_{l'}$.

First case: We have $\ell_s = \ell_{l'}$. The sentinel read earlier is equal to the current tail. Then all the nodes in Equation (2.2) reachable from ℓ_s are reachable from $\ell_{l'}$. But, $\ell_{l'}$ has no successors ($\ell_{l'\to}$ points to none) hence any node it can reach must be itself:

$$\ell_s = \ell_t = \ell_{l'} = \ell_{s'} \tag{2.3}$$

Per MAPSTO-AGREE-$\square$ this implies that $\ell_{s\to} = \ell_{l'\to}$. We thus find that the pointer being loaded is $\ell_{l'\to}$ and the points-to predicates

$$\ell_{l'\to} \hookrightarrow_{\mathsf{i}} \ell_{nil} * \ell_{nil} \hookrightarrow_{\mathsf{i}}^{\square} \mathsf{none}$$

are in the invariant. Hence the load results in ℓ_{nil}.

By combining the above with the following fact

$$\begin{aligned}&\mathrm{isQueue}_{\mathrm{MS}}(\ell_{s\to}, \ell_{s\to}, xs) \twoheadrightarrow \\ &\quad \ell_{s\to} \hookrightarrow_{\mathsf{i}} \ell_{nil} \twoheadrightarrow \ell_{nil} \hookrightarrow_{\mathsf{i}}^{\square} \mathsf{none} \twoheadrightarrow xs = [].\end{aligned}$$

we conclude that $xs_{\mathsf{i}} = []$ and hence also (from the big separating conjunction in I) that $xs_{\mathsf{s}} = []$. Using $xs_{\mathsf{s}} = []$ we can now apply DEQUEUE$_{\mathrm{CG}}$-NIL-R. After this our goal is to show the refinement where K represents the code of dequeue from line D5.

$$[\alpha := R] \models K[\,!\ell_{s\to}\,] \precsim \mathsf{none} : \mathsf{Option}\ \alpha.$$

We must show that the left-hand side steps to none which we can do as follows: On line D5 we know that the check in the if-statement is **true** since we know that the prophecy variable is resolved to ℓ_s. Hence symbolic execution proceeds to line D6 where ℓ_s is compared to ℓ_t. From Equation (2.3) we know that these are equal. On line D7 the location ℓ_{nil} is loaded; it points-to none and thus the function returns none on line D8.

Second case: There exists a node ℓ_n for which we have

$$\ell_{s\to} \hookrightarrow_{\mathsf{i}}^{\square} \ell_n * \ell_n \hookrightarrow_{\mathsf{i}}^{\square} \mathsf{some}\ (v, \ell_{n\to}) * \ell_n \rightsquigarrow \ell_{l'}.$$

The load evaluates to ℓ_n and we close the invariant.

On line D6 the location ℓ_s is compared to ℓ_t and we case on whether or not these locations are equal:

Case $\ell_s = \ell_t$***:*** The **if**-statement succeeds, we step to D7 which loads ℓ_n and thus evaluates to a some. Therefore the **match** takes the second branch to D10:

$$\mathbf{CAS}\ \ell_{\to t}\ \ell_t\ \ell_n\,;\ \mathsf{loop}\ ()$$

Here we apply Lemma 2.5.1, and for the last expression we apply the induction hypothesis.

Case $\ell_s \neq \ell_t$***:*** We step to D13 where dequeue attempts to swing the sentinel pointer forward:

$$\mathbf{if}\ \mathbf{CAS}\ \ell_{\to s}\ \ell_s\ \ell_n$$

We know that the **CAS** is safe since the invariant contains the points-to predicate $\ell_{\to s} \hookrightarrow_{\mathsf{i}} \ell_{s'}$ for some $\ell_{s'}$.

If the **CAS** fails we have not changed anything and can simply close the invariant, step to D15, and apply the induction hypothesis.

If the **CAS** succeeds then $\ell_s = \ell_{s'}$ and this is a linearization point. After the **CAS** we have $\ell_{\to s} \hookrightarrow_{\mathsf{i}} \ell_n$. Since ℓ_s is equal to $\ell_{s'}$ the pointer out of $\ell_{s'}$ must be equal to $\ell_{s\to}$. As such we have $\mathsf{isQueue}_{\mathrm{MS}}(\ell_{s\to}, \ell_{l\to}, \mathit{xs}_{\mathsf{i}})$ from the invariant for some xs_{i}.

If xs_{i} was $[]$ then ℓ_s would be equal to the last node, which points to none. But, this is in contradiction with the knowledge that ℓ_s is succeeded by ℓ_n. Hence xs_{i} cannot be $[]$. Thus there exists x_i and xs'_i such that $\mathit{xs}_{\mathsf{i}} = x_i :: \mathit{xs}'_i$; and x_s and xs'_s such that $\mathit{xs}_{\mathsf{s}} = x_s :: \mathit{xs}'_s$. For these:

$$\tau_i(x_i, x_s) * \mathop{\Large *}_{(x_i,x_s)\in(\mathit{xs}'_i,\mathit{xs}'_s)} \tau_i(x_i, x_s)$$

Moreover, x_i must be exactly the value in the node ℓ_n (i.e., $v = \mathsf{some}\ x_i$).

With the knowledge that the list is non-empty we can use $\textsc{dequeue}_{\mathrm{CG}}\textsc{-cons-r}$ after which we get $\mathrm{I}_{\mathrm{CG}}(\ell_{\mathit{cg}}, \mathit{lk}, \mathit{xs}'_{\mathsf{i}})$ and must show the refinement:

$$[\alpha := R] \models_{\mathcal{E}} K[\,\mathbf{true}\,] \precsim \mathsf{some}\ x_s : \tau$$

When we close the invariant we offer ℓ_n as a witness for the existentially quantified variable ℓ_s. To do this we must show $\gamma_s \Rrightarrow \ell_n$ and $\ell_n \dashrightarrow \gamma_t$—this is fairly easy.

After the **CAS** we arrive at D14. We know that the load evaluates to $\mathsf{some}\,(\mathsf{some}\ x_s, \ell_{n\to})$. Hence the entire expression on line D14 steps to some x_s and we are to show

$$[\alpha := R] \models_{\mathcal{E}} \mathsf{some}\ x_i \precsim \mathsf{some}\ x_s : \tau$$

which we can do because we have $\tau_i(x_i, x_s)$.

2.6.2 Enqueue

To conclude the proof we show refinement of enqueue:

$$[\alpha := R] \models \mathsf{enqueue}_{\mathrm{MS}}\ \ell_{\to t} \precsim \mathsf{enqueue}_{\mathrm{CG}}\ \mathit{lk}\ \ell_{\mathit{list}} : \alpha \to 1.$$

As both sides of the refinement are lambda-values we must show that these are related when applied to any two values, x_{i} and x_{s}, related by τ_i.

We first step over the construction of the new node on line E1. This gives us the resources:

$$\ell_n \hookrightarrow_{\mathsf{i}} \mathsf{some}\,(\mathsf{some}\ x_{\mathsf{i}}, \ell_{n\to}) * \ell_{n\to} \hookrightarrow_{\mathsf{i}} \ell_{\mathit{nil}} * \ell_{\mathit{nil}} \hookrightarrow_{\mathsf{i}} \mathsf{none}$$

Line E4 is an application of a recursive function. We therefore apply the LÖB rule as we did in the proof of dequeue.

To step over the load of $\ell_{\to t}$ on line E5 we open the invariant which contains the points-to predicate $\ell_{\to t} \hookrightarrow_{\mathsf{i}} \ell_t$ for some ℓ_t. The load evaluates to ℓ_t and when we close the invariant we keep the following persistent knowledge:

$$\ell_t \dashrightarrow \gamma_l * \ell_t \hookrightarrow_{\mathsf{i}}^{\square} \mathsf{some}\ (v, \ell_{t\to}), \tag{2.4}$$

for some v and $\ell_{t\to}$. The persistent points-to predicate for ℓ_t is used for the load on the next line, E6a. Since its contents match the operations applied to it, we can symbolically execute the rest of the line, and toNext is assigned to the value $\ell_{t\to}$.

The next line (E6b) loads $\ell_{t\to}$ and we open the invariant again. The invariant contains $\gamma_l \Mapsto \ell_l$ for some ℓ_l. By using ABS-REACH-CONCR we get $\ell_t \rightsquigarrow \ell_l$. We case on whether or not ℓ_t is equal to ℓ_l.

First case, $\ell_t = \ell_l$: We rewrite with the equality in the points-to predicate in Equation (2.4) and get $\ell_l \hookrightarrow_{\mathsf{i}}^{\square} \mathsf{some}\ (v, \ell_{t\to},)$. From the invariant we have $\ell_l \hookrightarrow_{\mathsf{i}}^{\square} \mathsf{some}\ (v', \ell_{l\to})$ and thus, by MAPSTO-AGREE-$\square$, we get $\ell_{t\to} = \ell_{l\to}$. From the invariant we further have

$$\ell_{l\to} \hookrightarrow_{\mathsf{i}} \ell_{nil} * \ell_{nil} \hookrightarrow_{\mathsf{i}}^{\square} \mathsf{none} \tag{2.5}$$

Hence we can conclude that the load evaluates to ℓ_{nil}. We close the invariant.

Symbolic execution continues to line E7. On this line $\ell_{\to t}$ is loaded again. We have already seen how the invariant ensures that such a load is safe. The newly read value is then compared to the old value read at line E5. If these are not equal symbolic execution proceeds to line E14 where we can conclude the proof by applying the induction hypothesis. If they are equal execution proceeds to line E8 where ℓ_{nil} is loaded. We use the points-to predicate from Equation (2.5) and conclude that the load evaluates to none.

Therefore, the **match** takes the first branch to the **CAS** on line E9:

if CAS $\ell_{t\to}$ ℓ_{nil} ℓ_n

To show that the **CAS** is safe we must have a points-to predicate for $\ell_{t\to}$. We can open the invariant and get a points-to predicate $\ell_{l'\to} \hookrightarrow_{\mathsf{i}} \ell_{nil}$ for some $\ell_{l'\to}$. Intuitively, if the **CAS** succeeds it is because $\ell_{t\to}$ is still the last node in the linked list and in that case $\ell_{t\to}$ is equal to $\ell_{l'\to}$.

This is where we apply our novel REL-CAS-L, which is quite subtle. This rule asks us to supply a witness which we must later show that ℓ_t points-to. To find such a witness observe that ℓ_t can reach $\ell_{l'}$. If they are equal then $\ell_{t\to}$ is equal to $\ell_{l'\to}$ and $\ell_{t\to}$ points to ℓ_{nil}. If they are not equal then $\ell_{t\to}$ must point to some other node. In both cases $\ell_{t\to}$ points to something, but in the first case the reasoning relies on the resource $\ell_{l'\to} \hookrightarrow_{\mathsf{i}} \ell_{nil}$. Hence, by giving up this resource we can conclude that there exists some ℓ_m such that

$$\begin{aligned} &\exists \ell_{m\to}.\ \ell_{t\to} \hookrightarrow_{\mathsf{i}}^{\square} \ell_m * \ell_m \hookrightarrow_{\mathsf{i}}^{\square} \mathsf{some}\ (-, \ell_{m\to}) * \ell_{l\to} \hookrightarrow_{\mathsf{i}} \ell_{nil} \\ &\quad \vee\ \ell_{t\to} \hookrightarrow_{\mathsf{i}} \ell_m * \ell_t = \ell_{l'} * \ell_m = \ell_{nil}. \end{aligned} \tag{2.6}$$

We offer this ℓ_m as a witness. We now have two cases corresponding to whether the **CAS** fails or succeeds and to the disjunction in REL-CAS-L.

CAS succeeds If the **CAS** succeeds then this is a linearization point. We must show the *full* points-to predicate (not just a persistent points-to) for $\ell_{t\to}$, but we only have the full points-to predicate in one of the disjuncts in Equation (2.6). Fortunately, from the rule we can assume that ℓ_m is equal to ℓ_{nil}, which points to none. This leads to a contradiction in the first disjunct in Equation (2.6) which states that ℓ_m points to a some. We can therefore assume the last disjunct. This does not only give us the full points-to predicate we need, it also tells us that ℓ_t is equal to the current last node $\ell_{l'}$ which is important to ensure that our change affects the queue correctly. Notice the subtlety involving equality, used to conclude that we had the full points-to predicate. Since we have now changed $\ell_{l'\to}$ we can use $\mathsf{isQueue}_{\mathrm{MS}}(\ell_{l'\to}, \ell_{s\to}, xs_{\mathsf{i}})$ to show $\mathsf{isQueue}_{\mathrm{MS}}(\ell_{l'\to}, \ell_{s\to}, xs_{\mathsf{i}} +\!\!+ [x])$. We have changed the last node from ℓ_t into ℓ_n. So we need to change $\gamma_l \Mapsto \ell_t$ into $\gamma_l \Mapsto \ell_n$. Clearly $\ell_t \rightsquigarrow \ell_n$, so we can use ABS-REACH-ADVANCE to achieve this.

Since this is the linearization point we use $\text{ENQUEUE}_{\mathrm{CG}}$-R to step the specification forward. We then have everything needed to close the invariant.

We continue to E17 where we apply Lemma 2.5.1 to show that the attempt at advancing the tail pointer is safe. The final expression is then $()$ which matches the right-hand side at this point.

CAS fails In this case we, can assume that $\ell_{l'} \neq \ell_l$. Following the rule REL-CAS-L we have to provide either a persistent or fractional points-to predicate for $\ell_{t\to}$. And from Equation (2.6) we know that we have one of these. We therefore consider each case in the disjunction and pick the corresponding case to show. This shows that the **CAS** is safe, and since nothing changed, it is trivial to close the invariant again. Execution steps to E11 where we apply the induction hypothesis.

Second case, $\ell_t \neq \ell_l$: . In this case, the tail pointer was lagging behind when we read it and there exists a node ℓ_m for which we have

$$\ell_{t\to} \hookrightarrow_{\mathsf{i}}^{\square} \ell_m * \ell_m \hookrightarrow_{\mathsf{i}}^{\square} \mathsf{some}\,(v', \ell_{m\to}) * \ell_m \rightsquigarrow \ell_l.$$

Hence the load evaluates to ℓ_m. We close the invariant.

Line E7 is handled as before. The load is safe, and if the two locations are not equal we apply the induction hypothesis at line E14. If the locations are equal we proceed to line E8 where ℓ_m is loaded. Since ℓ_m points to a some we step to E13. At E13 we apply Lemma 2.5.1 and then the induction hypothesis.

2.7 Consistent Snapshots Can Be Omitted

Recall the consistent snapshots in dequeue (line D5) and enqueue (line E7). The consistent snapshots are meant to solve the ABA problem by ensuring that the values

read are still up-to-date. However, with the insights gained from our formal proof, it becomes evident that these snapshots are actually *not* needed for correctness: from the way we have constructed the invariant we do not need to use the information gained from these checks. This is because the instance of the ABA problem that the consistent snapshot solves does in fact not occur in a garbage collected setting. And since the semantics of the language of our implementation, HeapLang, models a garbage collected language, we can formally prove that the atomic snapshots are not needed.

In the Coq formalization of our proofs, we have shown that the MS-queue without the consistent snapshots still contextually refines the coarse-grained queue. We have also shown that the coarse-grained queue refines the MS-queue both with and without the consistent snapshots. This implies that the coarse-grained queue is contextually *equivalent* to both queues, and, per transitivity of contextual refinement, that the MS-queue with consistent snapshots is contextually equivalent to one without.

We speculate that omitting the consistent snapshots may result in better performance as dequeue may still succeed even if the consistent snapshot fails. Hence this can lead to earlier success. As one can see in our Coq formalization, for the refinement proof of the MS-queue without the consistent snapshots it is not necessary to use prophecy variables in the proof.

2.8 Lagging-Tail MS-Queue

Our Coq formalization also contains a HeapLang implementation and a refinement proof for what we name the *lagging-tail* MS-queue. It resembles how the queue included in the Java standard library works and is a slightly more realistic version of the queue covered in [Tur+13b]. This variant is quite different from the original MS-queue in that it allows the tail pointer to lag behind arbitrarily, a change affecting both how dequeue and enqueue works: Dequeue can no longer rely on the sentinel being able to reach the tail and enqueue must read the tail pointer and, to account for the lagging tail, then iterate through the linked list until it finds the last node. While this is in many ways a simpler algorithm to prove correct, we find it remarkable that our notion of reachability also suffices to prove contextual refinement for this, very different, variant with only a very small change to the invariant. As the tail pointer may lag behind arbitrarily, it may, in particular, be further behind than even the sentinel pointer. Hence to prove contextual refinement for this variant we can no longer include $\ell_s \dashrightarrow \gamma_t$ in the invariant. However, by simply changing this part to $\ell_s \dashrightarrow \gamma_l$, we can prove refinement of the variant. No other changes are required to the invariant!

2.9 Defining the Persistent Points-To Predicate

This section describes how we implement the persistent points-to predicate. In Iris, Hoare triples, the weakest precondition, and the points-to predicate are not primitives in the logic. Instead, they are defined *inside the logic*, using what is called the Iris *base logic*. Hence, we can implement the persistent points-to predicate entirely inside Iris, by changing a definition (*heapCtx* below) that is used in the weakest precondition. An advantage of this approach is that soundness of the rules for the persistent points-to predicate follows directly from soundness of the Iris base logic.

The biggest challenge in adding the persistent points-to predicate is to ensure that it satisfies MAPSTO-INTRO-□. The existing points-to predicate is defined as ownership of some ghost state. Hence to make this rule true we need to use a resource algebra (RA) that supports a frame-preserving update from the ghost state owned by the normal points-to predicate to the ghost state owned by the persistent points-to predicate. We solve this by introducing the *discardable fractions RA*.

For space reasons, in the rest of this section we assume that the reader is familiar with ghost state and resource algebras in Iris. For the details, we refer to [Jun+18a].

Encoding of the heap To extend Iris as described we need to change two existing definitions: *heapCtx* and $\hookrightarrow_q$. The former is a predicate on heaps

$$\mathit{heapCtx} : (\textsc{Loc} \xrightarrow{\text{fin}} \textsc{Val}) \to \mathsf{iProp}.$$

which is part of the state interpretation used in the definition of the weakest precondition. For every step of execution, starting in a heap σ and ending in heap σ', $\mathit{heapCtx}(\sigma)$ holds before and $\Rightarrow\!\!\!\!\mid \mathit{heapCtx}(\sigma')$ holds after the step.

In the current version of Iris, *heapCtx* is defined using the RA

$$\textsc{Auth}(\textsc{Loc} \xrightarrow{\text{fin}} (\mathbb{Q}_{01} \times \textsc{Ag}(\textsc{Val}))), \tag{2.7}$$

where $\mathbb{Q}_{01}$ is the RA of fractions with the carrier $(0, 1]$, and the following definitions[2]:

$$\mathit{heapCtx}(\sigma) = \boxed{\bullet\sigma}^{\gamma_{\text{heap}}} \qquad \ell \hookrightarrow_q v = \boxed{\circ[\ell \leftarrow (q, \mathsf{ag}(v))]}^{\gamma_{\text{heap}}}$$

We note that $\ell \hookrightarrow_q v$ is not persistent since $\mathbb{Q}_{01}$ has no core. Updates to the heap are possible since $1 \in \mathbb{Q}_{01}$ is exclusive (it has no frame).

Recall that we want MAPSTO-INTRO-□ to hold *without depending on heapCtx*. This is because *heapCtx* is internal to the definition of weakest precondition and not exposed to clients of it. We therefore need to use an RA that makes it possible to make a frame-preserving update from the ghost state owned by $\hookrightarrow_q$ to the ghost state owned by $\hookrightarrow_\square$. The core should be undefined for the former while defined for the latter. We define such an RA in the next section. But, even with such an RA we have the problem that $\hookrightarrow$ denotes ownership of a fragment, and with the authoritative RA it is not clear how to make a suitable frame-preserving update from a fragment. We therefore also need to introduce a generalized authoritative RA.

[2]This is simplified—but covers what is relevant for our purpose.

Discardable fractions RA We introduce the RA of *discardable fractions*, which is a generalization of the normal fractional RA. Whereas elements of the fractional RA denote *ownership* over some strictly positive fraction, elements of the discardable fractional RA can additionally denote *knowledge* about a fraction having been discarded.

Let $\mathbb{Q}_{>0}$ denote the set of strictly positive rationals. The carrier for the RA is:

$$\textsc{DFrac} \triangleq \mathsf{own}(q) \mid \mathsf{disc}(p) \mid \mathsf{both}(q,p) \qquad q,p \in \mathbb{Q}_{>0}$$

One should think of this as pairs where one, *but not both*, of the values might be absent. The element $\mathsf{own}(q)$ is equivalent to an element of the normal fractional RA and the element $\mathsf{disc}(p)$ denotes the knowledge that the fraction p has been discarded.

The valid elements are those where the sum of the two numbers are less than or equal to 1:

$$\mathcal{V}(\mathsf{own}(p)) \triangleq p \leq 1 \qquad \mathcal{V}(\mathsf{disc}(q)) \triangleq q \leq 1$$
$$\mathcal{V}(\mathsf{both}(q,p)) \triangleq q + p \leq 1$$

The operation adds together the owned fractions and takes the maximum of the fractions known to be discarded. We do not specify all cases in the operation, the remaining cases are determined by the requirement that the operation is commutative and associative.

$$\mathsf{disc}(p) \cdot \mathsf{disc}(p') \triangleq \mathsf{disc}(\max(p,p'))$$
$$\mathsf{own}(q) \cdot \mathsf{own}(q') \triangleq \mathsf{own}(q+q')$$
$$\mathsf{own}(q) \cdot \mathsf{disc}(p) \triangleq \mathsf{both}(q,p)$$

The core of an element is the discarded part of the element if any. This ensures that knowledge about discarded fractions is persistent.

$$|\mathsf{disc}(p)| = \mathsf{disc}(p) \qquad |\mathsf{own}(q)| = \bot \qquad |\mathsf{both}(q,p)| = \mathsf{disc}(p)$$

We now have the following frame-preserving update.

Lemma 2.9.1. *Discarding is possible: own*$(q) \rightsquigarrow$ *disc*(q).

Proof. Suppose $\mathsf{own}(q) \cdot \mathsf{both}(q',p')$ is valid. Then $q + q' + p' \leq 1$, which implies that $q' + \max(q + p') \leq 1$ showing that $\mathsf{disc}(q) \cdot \mathsf{both}(q',p')$ is valid. The remaining cases are similar. □

Heap RA We would now like to replace the use of the fractional RA in Equation (2.7), the RA currently used for the heap, with the discardable fractional RA. However, this alone is not enough because, as mentioned, the authoritative RA does not make it possible to make the frame-preserving update from a fragment that we need.

We therefore need a slightly generalized variant of the authoritative RA that allows us to update the discardable fraction in fragments. For RA's A and B and a function $\pi : B \to A$ we define

$$\begin{aligned}
\textsc{PAuth}(A, B, \pi) &= \textsc{Ex}(A)^? \times B \\
\mathcal{V}((\bot, b)) &= \mathcal{V}(b) \\
\mathcal{V}((a, b)) &= \mathcal{V}(a) \land \mathcal{V}(b) \land \pi(b) \preccurlyeq a \\
(a, b) \cdot (a', b') &= (a \cdot a', b \cdot b') \\
|(a, b)| &= \begin{cases} (\bot, |b|) & \text{if } |b| \neq \bot \\ \bot & \text{otherwise} \end{cases}
\end{aligned}$$

The full and fragmental view is defined as usual.

$$\bullet a \triangleq (a, \varepsilon) \qquad\qquad \circ b \triangleq (\bot, b)$$

For this construction to satisfy the laws of a RA π must be expansive with respect to the inclusion order.

The difference between this construction and the normal authoritative RA is that the authoritative and fragmental view can contain two different RA's and that in the definition of validity $\pi(b)$, and not b itself, should be included in a.

To model the heap we then instantiate the above construction by using

$$\textsc{PAuth}(\textsc{Loc} \xrightarrow{\text{fin}} \textsc{Ag}(\textsc{Val}), \textsc{Loc} \xrightarrow{\text{fin}} (\textsc{DFrac} \times \textsc{Ag}(\textsc{Val}), \pi_2).$$

The definitions for the heap are then

$$\begin{aligned}
\mathit{heapCtx}(\sigma) &\triangleq \boxed{\bullet\sigma}^{\gamma_{\text{heap}}} \\
\ell \hookrightarrow_q v &\triangleq \boxed{\circ[\ell \leftarrow (\mathsf{own}(q), v))]}^{\gamma_{\text{heap}}} \\
\ell \hookrightarrow_\square v &\triangleq \exists p. \boxed{\circ[\ell \leftarrow (\mathsf{disc}(p), v)]}^{\gamma_{\text{heap}}}
\end{aligned}$$

This ensures that the fraction in the fragment is independent of the full authoritative view and hence that it can be updated without the full authoritative view.

Lemma 2.9.2. *If $q, q' \in$ DFrac and $q \rightsquigarrow q'$ then $\circ[k \leftarrow (q, v)] \rightsquigarrow \circ[k \leftarrow (q', v)]$.*

Finally, from Lemma 2.9.1 and Lemma 2.9.2 we have the frame-preserving update

$$\circ[\ell \leftarrow (\mathsf{own}(q), v))] \rightsquigarrow \circ[\ell \leftarrow (\mathsf{disc}(p), v)]$$

and can thus show Mapsto-intro-$\square$.

2.10 Related Work

We now discuss related work that has not already been treated in the paper. The only related work that directly shows contextual refinement is the already mentioned

pen-and-paper proof by Turon et. al. However, they only consider a simplification of the less challenging lagging-tail MS-queue. Their approach relies on assigning to each node a *state* in a state transition system. However, they have no notion of reachability, which appears to be necessary for reasoning about the original MS-queue. And since reachability is a *relationship* between two nodes and not a state of one particular node, it is not clear how to extend their approach to the MS-queue. Our approach on the other hand applies to both the MS-queue and the lagging-tail MS-queue.

We now cover related work that shows *linearizability* of the MS-queue. Doherty *et al.* proved that a slightly modified MS-queue is linearizable by using a simulation proof formalized in the PVS proof system [Doh+04]. Their simulation proof makes use of both a forward simulation and a backwards simulation; this is comparable to our use of prophecy variables. They make several changes to the queue which they argue improve performance. Their changes preserve the future dependent linearization point, but they also remove the check on line D6, which we found challenging in our proof. Schellhorn *et al.* later showed that backwards simulation suffices to show linearizeability of the MS-queue [SDW14].

Vafeiadis proposed an automatic verification procedure for proving linearizability for first-order programs [Vaf10]. His approach handles certain non-fixed linearization points, namely those that are *pure*, meaning that the linearization points do not change the state of the queue. The non-fixed linearization point in dequeue in the MS-queue is pure, as dequeueing an element from an empty queue does not change the state of the queue. Vafeiadis's approach depends on this to obtain a verification procedure for proving linearizability which can handle the MS-queue. His approach is also based on prophecy variables. As mentioned in the Introduction, this notion of linearizability does not imply contextual refinement for our rich higher-order language. We further remark that ReLoC also supports future dependent linearization points even when these are not pure.

Liang and Feng propose a program logic to verify linearizability [LF13a]. They use their approach to verify an impressive number of concurrent data structures, with the MS-queue being one of them. To handle the non-fixed linearization points they use speculation. This approach is related to prophecy variables and does not rely on annotations in the implementation. The program logic and their verification of the MS-queue are not mechanized.

There exists several other approaches to verifying linearizability which can handle non-fixed linearization points, and which should therefore also be able to verify the MS-queue. For these, we refer to the excellent survey [DD14].

Related to the persistent points-to predicate, Charguéraud and Pottier showed how to extend separation logic with a general read-only modality [CP17]. This modality makes it possible to temporarily give read-only access to a points-to predicate, without having to keep track of fractions as one needs to do with the fractional points-to predicate. However, even though they remark that it should be possible to construct a predicate for immutable data, they explicitly do not do that. Their approach is for *temporarily* making locations read-only while ours is for *permanently*

making locations read-only.

Chapter 3

Mechanized Verification of a Fine-Grained Concurrent Queue from Meta's Folly Library

Abstract

We present the first formal specification and verification of the fine-grained concurrent multi-producer-multi-consumer queue algorithm from Meta's C++ library Folly of core infrastructure components. The queue is highly optimized, practical, and used by Meta in production where it scales to thousands of consumer and producer threads. We present an implementation of the algorithm in an ML-like language and formally prove that it is a contextual refinement of a simple coarse-grained queue (a property that implies that the MPMC queue is linearizable). We use the ReLoC relational logic and the Iris program logic to carry out the proof and to mechanize it in the Coq proof assistant. The MPMC queue is implemented using three modules, and our proof is similarly modular. By using ReLoC and Iris's support for modular reasoning we verify each module in isolation and compose these together. A key challenge of the MPMC queue is that it has a so-called *external linearization point*, which ReLoC has no support for reasoning about. Thus we extend ReLoC, both on paper and in Coq, with novel support for reasoning about external linearization points.

3.1 Introduction

It is well-known that it is challenging to program, specify, and verify fine-grained concurrent algorithms, and in recent years we have seen much progress on program logics for specifying and verifying such algorithms, *e.g.*,[Din+10; FKB18; JP11; Jun+15a; LF13b; SNB15; SB14; TDB13; Tur+13a; TW11; Vaf08; VP07]. In this paper, we present the first formal specification and verification of the highly efficient and practical concurrent multi-producer-multi-consumer queue algorithm found in Meta's open-source library Folly (or, simply, the MPMC queue in the rest of the paper).

The Folly library is an open-source collection of key infrastructure components implemented in C++ and used extensively in production at Meta [Met21]. The library contains, among many other things, the MPMC queue.[1] The queue was originally developed by Nathan Bronson to connect two thread pools inside TAO, Meta's distributed data store for their social graph [Bro+13]. One of the key ideas used in the algorithm is to improve scalability by decreasing the contention found in other lock-free algorithms, such as the Michael-Scott queue [MS96a], by striping the queue across q "smaller" sub-queues. To avoid the overhead of maintaining q sub-queues, the striping is taken to the extreme by letting each sub-queue store only a single element. These *single-element queues* can then be simpler and faster. In fact, they are implemented merely as a reference to a value and a so-called *turn sequencer*. The latter is a synchronization mechanism used by the single-element queue to guard access to its value. The enqueue and dequeue operations on the MPMC queue are delegated to one of the single-element queues by taking a ticket from one of two ticket dispensers using an atomic increment (**FAA**). After receiving a ticket, up to q separate enqueue or dequeue operations can proceed in parallel, completely independent of each other as they operate on different single-element queues. The **FAA** instruction thus becomes the main point of contention, but since an **FAA** instruction (unlike **CAS**) always succeeds, this design, in the words of Bronson, "makes contention count" as its cost always pays off in significant progress being made in the algorithm [Bro20]. Altogether, this makes the queue scalable to hundreds of thousands of producer and consumer threads.

More concretely, in this paper we present an implementation of the MPMC queue and all its components in an ML-like language with concurrency primitives. The implementation captures the essence and the key verification challenges of the algorithm while eliding some of the low-level details of the original C++ implementation. We prove that the MPMC queue *contextually refines* a coarse-grained concurrent queue. The coarse-grained queue uses a lock to ensure that only one thread at a time access the queue. We take this simple queue to be the *specification* of a queue and the MPMC queue to be an *implementation* of the specification. Informally, the contextual refinement property then means that in any program we may replace uses of the "obviously correct" coarse-grained concurrent queue with the more efficient, but also more complicated, MPMC queue, without changing the observable behavior of the program. More precisely, an expression e contextually refines another expression e', if for all contexts C of a ground type, if $C[e]$ terminates with a value, then there exists an execution of $C[e']$ that terminates with the same value.

We prove the contextual refinement using the recently proposed relational logic ReLoC [FKB18; FKB20a], which builds on top of the Iris separation logic [Jun+16; Jun+18b; Jun+15a; Kre+17a] and greatly simplifies proofs of contextual refinement by offering rules that allows one to reason about refinements at a high level of abstraction. Additionally, it is mechanized in Coq and allowed us to develop our

[1] The source code is available online at `https://github.com/facebook/folly/blob/main/folly/MPMCQueue.h`.

mechanized proof interactively using the Iris proof mode [Kre+18; KTB17b].

To verify a fine-grained concurrent algorithm, one of the key steps is to identify the *linearization points* of its operations: the point during execution where the operation "appears to take place". In our analysis of the MPMC queue we discover that, in some cases, the linearization point of the dequeue operation is *external*. A linearization point is external if it happens during the execution of another operation. For dequeue, its linearization point may happen within the execution of an enqueue operation, which is not immediately obvious by looking at the code. As we explain in detail later, the external linearization points arise because the algorithm, in contrast to other fine-grained concurrent queues, is not entirely non-blocking: if all the single-element queues are full (resp. empty) then enqueue (resp. dequeue) is blocked.

One may categorize linearization points into three classes [DD15]: fixed, future-dependent, and external. The first version of ReLoC [FKB18] had support for reasoning about fixed linearization points only, and ReLoC Reloaded [FKB20a] added support for future-dependent linearization points, through its use of Iris-style prophecy variables [Jun+20a]. However, we observe that neither version of ReLoC supports reasoning about external linearization points. The high-level reason is that ReLoC ties the state of the implementation with the state of the specification as a single judgment. At an external linearization point (in our case in dequeue) the state of the specification must be transferred to the other operation where the linearization point takes place (in our case enqueue). This is not possible with ReloC's existing rules. Hence, to verify the MPMC queue we extend ReLoC with new proof rules and generalize its existing proof rules to be able to reason about external linearization points. The extension is simple but elegant and "completes the picture" by making ReLoC able to handle all three classes of linearization points. External linearization points often occur due to *helping* and our extension makes ReLoC able to handle these concurrent data structures with helping as well.

As mentioned, the MPMC queue is implemented as three submodules: the MPMC queue is implemented using the single-element queue, which is implemented using the turn-sequencer. A strength of our approach is that our contextual refinement proof is similarly modular: it makes use of (unary) Hoare-style specifications of the turn-sequencer and the single-element queue. Here we leverage the fact that ReLoC allows for compositional reasoning and that it, following [TDB13], includes a proof rule that allows one to use Hoare-style specifications, written in the Iris program logic, to simplify reasoning about the left-hand side in a relational proof [FKB18, Section 7.4]. We thus end up not only with a refinement proof of the MPMC queue but also with reusable specifications for the single-element queue and the turn sequencer.

To arrive at sufficiently composable specifications we make use of a proof pattern involving a resource algebra over *infinite* sets, to keep track of which turns are "still available" (see Section 3.4). The idea of using infinite structures to improve composability is well-known in the context of functional programming [Hug89]. In our case, the specification of the turn-sequencer supplies its client with an *infinite* set of turns and the specification of the single-element queue gives its client two

infinite sets of tickets. This approach greatly simplifies the proofs and makes it possible to reason about the single-element queue in the refinement proof with the details of the turn sequencer having been abstracted away. We believe this proof pattern could also be used to simplify reasoning about other algorithms based on these components, and have used it to additionally verify a ticket lock based on the turn sequencer.

Another challenge in verifying the MPMC queue is that its physical state (*i.e.,* the actual content in the underlying array) does not immediately determine the *abstract state* of the queue (*i.e.,* the state that is observable through the queue interface). In particular, a value may be present in the physical state of the queue without actually being in the queue (*i.e.,* not observable with a dequeue operation), and vice versa. This lies in contrast with other data structures, even those with non-fixed linearization points (such as the Herlihy-Wing queue [HW90] and the Michael-Scott queue [MS96a]).

In summary, we believe that verifying the MPMC queue serves as an interesting case study, as it is challenging to verify, used at scale in the industry, has not been treated in the literature before, and it provides motivation for extending ReLoC with support for external linearization points.

Outline and contributions.

- Since the MPMC queue has not been treated in the literature before, we give a detailed description of it (Section 3.2).
- We informally analyze the linearization points of the MPMC queue and observe that one of them is external (Section 3.3).
- We define and prove Hoare-style specifications for the turn sequencer and single-element queue (Section 3.4).
- We show that the MPMC queue contextually refines a coarse-grained queue. (Sections 3.5 and 3.6). Our proof is *modular* and makes use of the aforementioned Hoare-style specifications for the submodules.
- We explain why prior versions of ReLoC can not handle external linearization points and extend ReLoC, both on paper and in Coq, with support (including tactics) for reasoning about external linearization points (Section 3.7).
- We have formalized all the results in this paper, and two additional examples of algorithms with external linearization points in the Coq proof assistant [VFB21]. The formalization is part of the ReLoC git repository and can be found online at `https://gitlab.mpi-sws.org/iris/reloc/-/tree/master/theories/examples/folly_queue`. The version that we specifically refer to in this paper corresponds to the commit with the git hash `b6df47f9`.

We discuss related and future work in Section 3.8.

3.2 The Folly MPMC queue

We now describe the three data structures, starting with the turn sequencer and proceeding bottom-up.

3.2.1 Turn Sequencer

A turn sequencer is a data structure that implements mutual exclusion by *sequentializing* access to a critical section among threads *ordered* by a monotonically increasing turn. The turn sequencer implementation is shown in Figure 3.1a.

The turn sequencer provides two operations: wait and complete. These are similar to the acquire and release operations on a lock, but they take an additional natural number as an argument. The natural number specifies which *turn* to wait for or to complete. The turn sequencer guarantees that if a thread waits for the nth turn, then it will only proceed once all the preceding turns have been completed. For this to hold, the turn sequencer assumes that its clients never wait for the same turn several times. As such, it is the responsibility of clients to manage the turns, *i.e.,* which natural numbers they wait for. Compared to a lock, this places a greater demand on the client, but in return the client is given precise control over the order in which threads run their critical sections.

We implement the turn sequencer as a pointer ts to a number, which represents the current turn. The function wait ts n simply spins on that pointer until its value is equal to n. The implementation of complete ends the current turn by incrementing ts.

3.2.2 Single-Element Queue

A single-element queue (SEQ) is a queue with a capacity of one. Our implementation is shown in Figure 3.1b. It is a *blocking* queue: if it is empty (full) then any subsequent dequeue (enqueue) is blocked until the queue becomes non-empty (non-full).

Similarly to the turn sequencer, the SEQ's operations take a turn as an argument, however the turns are separate for enqueue and dequeue. The turn argument specifies the order of the operations: an enqueue or dequeue operation is carried out only after all operations with a lower number have been carried out. For an enqueue and a dequeue operation with the same turns, the enqueue is carried out first. This ordering ensures that when an enqueue operation is carried out, the queue is always empty, and when dequeue is run the queue is non-empty.

The SEQ is implemented as a reference to an Option type, protected by a single turn sequencer. To ensure that the turn sequencer operations are called with correct turns, the implementations of the enqueue and dequeue operations adhere to the following discipline. The even turns of the turn sequencer correspond to the enqueue operations and the odd turns correspond to the dequeue operations. Hence when $\mathsf{enqueue}_{SEQ}$ ($\mathsf{dequeue}_{SEQ}$, respectively) is called with turn n, the corresponding turn for the turn sequencer is $2n$ ($2n + 1$, respectively). Not only does this allow for a

```
newTS : 1 → ref int
newTS () = ref 0

complete : ref int → int → 1
complete ts turn =
  ts ← turn + 1;
  ()

wait : ref int → int → 1
wait ts turn =
  let turn' = ! ts in
  if turn' = turn
  then ()
  else wait ts turn
```

(a) Turn sequencer.

```
type SEQ α = ref int × ref (Option α)

queue_SEQ : ∀α. 1 → SEQ α
queue_SEQ () = (newTS (), ref none)

enqueue_SEQ : SEQ α → int → α → 1
enqueue_SEQ (ts, r) enqTurn v =
  let turn = enqTurn * 2 in
  wait ts turn;
  r ← some(v);
  complete ts turn

dequeue_SEQ : SEQ α → int → α
dequeue_SEQ (ts, r) deqTurn =
  let turn = deqTurn * 2 + 1 in
  wait ts turn;
  let v = match ! r with
    | some(x) ⇒ x
    | none ⇒ assert(false)
  in complete ts turn; v
```

(b) Single-element queue.

```
newQueue : ∀α. int → (1 → α) × (α → 1)
newQueue q = Λ.
  let slots = arrayInit q queue_SEQ in
  let pushTicket = ref 0 in
  let popTicket = ref 0 in
  (λv. enqueue slots q pushTicket v,
   λx. dequeue slots q popTicket)

enqueue : array (SEQ α) →
  int → int → α → 1
enqueue slots q pushTicket v =
  let t = FAA(pushTicket, 1) in
  let idx = t mod q in
  let ticket = t/q in
  enqueue_SEQ (slots[idx]) ticket v

dequeue : array (SEQ α) → int → int → α
dequeue slots q popTicket =
  let t = FAA(popTicket, 1) in
  let idx = t mod q in
  let ticket = t/q in
  dequeue_SEQ (slots[idx]) ticket v
```

(c) MPMC queue.

```
queue_CG : ∀α. (1 → α) × (α → 1)
queue_CG = Λ.
  let w = (newlock (), ref []) in
  (λv. enqueue_CG w v,
   λx. dequeue_CG w)

enqueue_CG : lock × list α → α → 1
enqueue_CG (lk, hd) v =
  let rec go v ls =
    match ls with
    | [] ⇒ [v]
    | h :: t ⇒ h :: go v t
  in acquire lk;
  hd ← go v (! hd);
  release lk

dequeue_CG : lock × list → α
dequeue_CG (lk, hd) =
  acquire lk;
  match ! hd with
  | [] ⇒ assert(false)
  | h :: t ⇒ hd ← t;
              release lk;
              h
```

(d) Coarse-grained queue.

Figure 3.1: Implementation of the various data structures.

single turn sequencer to provide turns for both of the operations, it also ensures that the enqueue and dequeue operations are carried out in the correct order. The first enqueue gets the first even turn, 0, the first dequeue gets the first odd turn, 1, and so on. Hence the enqueue and dequeue operations alternately get access to the pointer, and the dequeue operation can be sure that a value is present when it reads the pointer.

3.2.3 MPMC queue

The MPMC queue is a blocking queue of a fixed capacity q. The implementation of the MPMC queue is shown in Figure 3.1c. The binary operator "mod" denotes modulo (or remainder) and "/" denotes *integer* division (*i.e.,* $3/2 = 1$). The Λ is a type abstraction (or a generic) making the queue polymorphic in the type of values it can store.

Upon initialization, an array of length q is created, with each entry containing a SEQ. The function arrayInit constructs an array of the given length, calls the given function once for each entry, and sets the entry to the result. In addition to the array, the queue contains two *ticket dispensers* (references to natural numbers): $pushTicket$ and $popTicket$. The first keep track of tickets for the enqueue operation, and the second does the same for the dequeue operation.

The enqueue operation first takes a ticket by incrementing the value of $pushTicket$ with **FAA**, which atomically increments the ticket and leaves enqueue with a ticket t. From this ticket, we calculate an index $(t \bmod q)$ in the array for a SEQ. Then, enqueue writes an element into the SEQ by using the turn $\lfloor t/q \rfloor$. The dequeue operation proceeds in a similar way. It atomically increments $popTicket$ and calculates an index and a turn in the same way. It dequeues a value from the SEQ and returns this value.

3.2.4 Relationship to original C++ code

Our implementation of the MPMC queue in ReLoC's ML-like language is faithful to the original algorithm, but does omit some low-level details of the original C++ implementation.

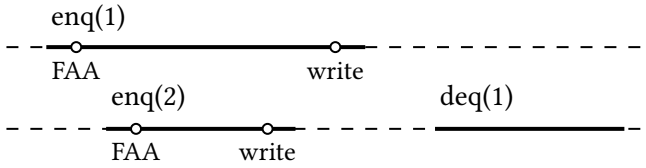
- The C++ implementation takes into account the C++ relaxed-memory model whereas the memory model of ReLoC's ML-like language is sequentially consistent. ReLoC does not support weak memory so verifying the MPMC queue in a weak memory setting would have required a different verification methodology.
- The C++ turn sequencer gracefully handles integer overflow of the turn counter. As the ML-like language included with ReLoC only support unbounded integers our implementation does not handle overflow.
- When waiting for a turn, the C++ turn sequencer uses a heuristic consisting of spinning with a back-off and suspending the thread (using futexes [FRK02])

for increased performance. Our implementation only uses spinning. This difference only affects efficiency and not the safety or linearizability of the algorithm. Additionally, to manage the use of futexes the integer in the turn sequencer stores not only the current turn but also uses some bits to manage sleeping threads. Due to this, the turn is incremented using compare-and-set and not **FAA** as in our implementation.

- The C++ implementation supports additional operations in addition to the queue operations dequeue and enqueue. For instance, an enqueue operation that fails instead of blocking when the queue is full.
- The use of closures in our implementation can be seen as corresponding to the use of objects in C++.

3.3 Linearizability of the MPMC queue

In this section, we analyze the MPMC queue informally and identify its linearization points. As a first guess, one might think that the linearization point for enqueue is when enqueue writes its value into the SEQ and, similarly, for dequeue when it reads the value from the SEQ. After all, these are the points where a value is physically inserted into or read from the data structure. However, placing the linearization points in this way does not work, as the following example shows:



This diagram represents two threads executing operations on the queue. The filled segments represent the duration of the operations. In the example, the first enqueue executes its **FAA** and receives ticket 0. Afterward the second enqueue executes its **FAA**, receives ticket 1, and writes its value to the queue. Then the first enqueue writes its value. Finally, a dequeue executes; gets ticket 0, and therefore returns 1. To make this consistent, the linearization point of the first enqueue should happen before the linearization point of the second enqueue. But, the second enqueue writes its value into the queue *before* the first enqueue does so. Hence, making the linearization points at that time in enqueue is too late.

As the example suggests, the linearization point of the enqueue operation happens at the **FAA**. If an enqueue operation receives a ticket i, then clearly the value that it inserts into the queue is eventually read and returned by the dequeue operation that also receives the ticket i. This means that exactly when the **FAA** in enqueue is executed, it is determined where in the queue its value is inserted. It thus makes sense to place the linearization point at the **FAA**. Following this line of argument, we say that the enqueue that receives ticket i is the ith enqueue. Moreover, we call

the dequeue that receives ticket i the ith dequeue, and we say that the ith enqueue and the ith dequeue *correspond* to each other.

It might seem that the linearization point in dequeue is similarly at the **FAA** operation. This, however, does not always work, as the following example shows:

The crux of the example is that dequeue receives ticket 0 before the corresponding enqueue takes its ticket. It is therefore not consistent to put the linearization point of dequeue at its **FAA**, as dequeue would then take place before the value it returns is enqueued in the first place. However, in general, one can not place the linearization point at when dequeue reads the value either, as that would lead to the same problems as for enqueue.

Thus, the linearization point of the dequeue operation is not always fixed. Looking at the example, we see that we could place the linearization point for the waiting dequeue *just after* the linearization point for the enqueue operation that unblocks it. This means that the linearization point of dequeue happens during the execution of enqueue — an external linearization point.

In summary, we conclude the following. If the ith dequeue arrives *after* its corresponding enqueue then it has a fixed linearization point at its **FAA**. If, on the other hand, it arrives *before* its corresponding enqueue then it has an external linearization point, which happens right after the corresponding enqueue's linearization point. Observe that even with the external linearization point, it is the case that the ith dequeue always has its linearization point before the $(i + 1)$'th dequeue.

Abstract state. Given the placement of the linearization points as above, we can talk about the *abstract state* of the queue, which is determined by the linearized order of the operations. Note that as soon as enqueue receives a ticket, the enqueued element becomes a part of the abstract state, before it is even written into the array. Symmetrically, when a dequeue receives a ticket, it removes an element from the logical queue, even though that value is still present in the physical queue. Thus, the physical state of the underlying array does not determine the abstract state of the queue, *e.g.,* the queue might physically contain no values, while logically it contains arbitrarily many values (and vice versa).

Calculating the abstract state of the queue is important in the refinement proof (Sections 3.5 and 3.6), but it is not related directly to the physical state of the array. The abstract state is, however, directly related to the values of *pushTicket* and *popTicket*. If $popTicket \leq pushTicket$, then there are exactly $pushTicket - popTicket$ elements in the logical queue. Otherwise the queue is empty and there are $popTicket - pushTicket$ dequeue operations that have arrived before their corresponding enqueue. We will

Turn Sequencer

$\{R(0)\}\ \mathsf{newTS}\ ()\ \{v.\ \exists\gamma.\ \mathsf{isTS}(\gamma, R, v) * \mathsf{turns}(\gamma, \mathbb{N})\}$

$\{\mathsf{isTS}(\gamma, R, v) * \mathsf{turn}(\gamma, n)\}\ \mathsf{wait}\ v\ n\ \{R(n) * \mathsf{close}(v, n)\}$

$\{\mathsf{isTS}(\gamma, R, v) * R(n+1) * \mathsf{close}(v, n)\}\ \mathsf{complete}\ v\ n\ \{\mathsf{True}\}$

Single-Element Queue

$$\{\mathsf{True}\}\ \mathsf{queue}_{SEQ}\ ()\ \left\{v.\exists\gamma.\ \begin{array}{c}\mathsf{isSEQ}(\gamma, Q, v) * \\ \mathsf{turns}_e(\gamma, 0) * \mathsf{turns}_d(\gamma, 0)\end{array}\right\}$$

$\{\mathsf{isSEQ}(\gamma, Q, v) * \mathsf{turn}_e(\gamma, n) * Q(n, x)\}\ \mathsf{enqueue}_{SEQ}\ v\ n\ x\ \{\mathsf{True}\}$

$\{\mathsf{isSEQ}(\gamma, Q, v) * \mathsf{turn}_d(\gamma, n)\}\ \mathsf{dequeue}_{SEQ}\ v\ n\ \{x.Q(n, x)\}$

Figure 3.2: Unary specifications for turn sequencer and SEQ.

see how these considerations are formalized as part of the refinement proof in Section 3.6.

3.4 Specifications for the Turn Sequencer and the Single-Element Queue

In this section, we define suitable Hoare triple specifications for the turn sequencer and the SEQ. We also sketch how these are proved. We emphasize that the proof of the SEQ only uses the *specification* (and not the implementation) of the turn sequencer. Similarly, when we prove contextual refinement for the MPMC queue, we only make use of the specification for the SEQ. Thus our specifications and proofs are *modular*, and we observe that to prove contextual refinement for the MPMC queue, a unary specification for the SEQ suffices.

3.4.1 Turn Sequencer

As mentioned earlier, the turn sequencer is a mechanism for mutual exclusion. Therefore, our specification of the turn sequencer (shown in Figure 3.2) is an extension of a typical concurrent separation logic specification for a lock [BB21; Got+07], and the verification process is similar to the verification of a ticket-based lock [MS91, Section 2.2]. There are two key differences though. The first difference is that it is up to the client of the turn sequencer to ensure that the turns are used correctly. For instance, wait should never be invoked with a past turn. The second difference is that the resource protected by the turn sequencer is indexed by a turn number, which allows for a more dynamic treatment of resources protected behind a critical

$$\textsc{turn-alloc}\ \frac{X \subseteq \mathbb{N}}{\Rrightarrow \exists \gamma.\, \mathsf{turns}(\gamma, X)} \qquad \textsc{turn-disj}\ \frac{\mathsf{turns}(\gamma, X) \qquad \mathsf{turns}(\gamma, Y)}{X \cap Y = \emptyset}$$

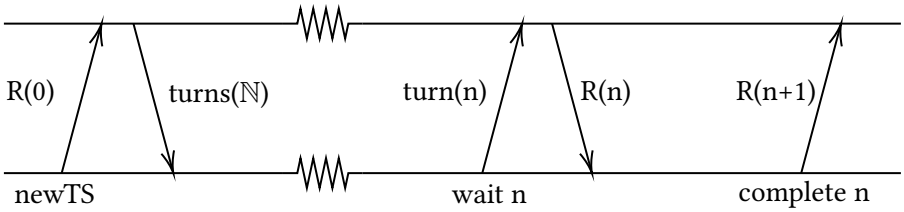
$$\textsc{turn-sep}\ \frac{X \cap Y = \emptyset}{\mathsf{turns}(\gamma, X) * \mathsf{turns}(\gamma, Y) \dashv\vdash \mathsf{turns}(\gamma, X \cup Y)}$$

Figure 3.3: Rules for turns.

section. In some sense, this makes the specification for the turn sequencer stronger than that for a lock, and in our Coq formalization we have implemented and verified a lock based on the turn sequencer.

The specification uses two predicates "close" and "isTS", which are abstract to clients of the specification (as in [BBT07; PB05]). The latter, "isTS", is the representation predicate. It is *persistent*, which intuitively means that, unlike other separation logic propositions, it is freely duplicable and not consumed by preconditions.

The predicate R describes the resource that the turn sequencer protects. Whereas a lock protects a resource $R : \mathsf{iProp}$, the turn sequencer protects a $\mathbb{N}$-indexed *family* of resources, that is, $R : \mathbb{N} \to \mathsf{iProp}$, where the index represents the current turn. This generalization of the protected resource is possible since the turn sequencer guarantees to run clients in the order of their turns. When it becomes a client's turn to enter its critical section, it can rely on all earlier turns having been carried out. This allows for "threading" the resource through all the clients, as depicted in the diagram below where the turn sequencer is at the top and its clients at the bottom.



The $R(0)$ in the precondition of newTS ensures that when a turn sequencer is created, the turn sequencer owns the resource for the initial turn. When wait is called with turn n, the client receives the resource for that turn, $R(n)$. When completing the turn, the client must give back $R(n+1)$ and not $R(n)$. This makes it possible for the turn sequencer to give $R(n+1)$ to the next thread in line (which is waiting for the turn $n+1$).

We now consider the handling of turns in the specification. To represent turns we use *ghost state*, an Iris feature also found in other separation logics [Din+13; Jun+15a; Nan+14]. Ghost state are resources that do not correspond to any physical state of the program. In our case, we want a resource representing ownership over

turns—where owning the turn n implies that one has the "right" to wait for the nth turn. For that purpose, we use a predicate $\mathsf{turns}(\gamma, X)$ that denotes ownership over the set of turns $X \subseteq \mathbb{N}$, and the singular $\mathsf{turn}(\gamma, n) \triangleq \mathsf{turns}(\gamma, \{n\})$ that denotes ownership over a turn $n \in \mathbb{N}$. These turns can be manipulated, for instance by a client of the turn sequencer, using the rules in Figure 3.3. The *update modality*, $\Rightarrow$, in these rules represents the possibility of updating ghost state and can safely be ignored. The rule TOKENS-ALLOC states that for any set of natural numbers one can construct a resource for them with a fresh *ghost name* γ. The ghost name can be thought of as a location or variable for the ghost state. Ownership over two sets of turns implies that the sets are disjoint (TURN-DISJ). Ownership over two disjoint sets of turns is equivalent to ownership of their union (TURN-SEP).

As depicted in the diagram above, when a client creates a new turn sequencer, it acquires ownership over *all* turns: $\mathsf{turns}(\gamma, \mathbb{N})$. To call wait for a turn n the client must own $\mathsf{turn}(\gamma, n)$, the ownership of which is then transferred into the turn sequencer, ensuring that the client can only wait for the same turn once. This is necessary for safety of the turn sequencer, as previously mentioned.

Finally, when a client acquires the current turn, it gets $\mathsf{close}(v, n)$, an exclusive resource giving permission to complete the turn.

Proof of Specification (Sketch). To prove that the implementation of the turn sequencer meets the specification, we use the following definitions of the predicates:

$$\mathsf{close}(\ell, n) \triangleq \ell \overset{1/2}{\hookrightarrow} n$$

$$\mathsf{isTS}(\gamma, R, \ell) \triangleq \boxed{\begin{array}{c}\exists n.\, \ell \overset{1/2}{\hookrightarrow} n * \mathsf{turns}(\gamma, \{m \in \mathbb{N} \mid m < n\}) * \\ (R(n) * \mathsf{close}(\ell, n) \lor \mathsf{turn}(\gamma, n))\end{array}}^{\mathcal{N}}$$

The predicate "isTS" is defined as an *invariant*. An invariant $\boxed{P}^{\mathcal{N}}$ represents the knowledge that the proposition P always holds. Since an invariant is knowledge and not a resource that one owns, this definition satisfies the previously mentioned property that "isTS" is persistent.

With these definitions, we now sketch how the specifications are proved.

For newTS, we have the resource $R(0)$ from the precondition and we obtain $\ell \hookrightarrow 0$ from stepping through the implementation. We can then allocate the ghost state $\mathsf{turns}(\gamma, \mathbb{N})$ using TURN-ALLOC. This allows us to establish the invariant by picking the left disjunct therein.

For wait, we open the invariant around the load. We then have the points-to predicate for the location, and can consider whether the value stored in the location is equal to the turn that wait was called with. In the latter case, we can use induction to handle the recursive call when the check in if fails. In the former case, the $\mathsf{turn}(\gamma, n)$ in the right disjunct in the invariant leads to a contradiction, due to the $\mathsf{turn}(\gamma, n)$ in the precondition. We thus have the resources in the left disjunct which we can use to show the postcondition, and then close the invariant by showing the right disjunct.

Finally, for complete, we use close(ℓ, n) in the precondition to conclude that n is still the current turn, *i.e.,* the existential is equal to n. This is the case since close(ℓ, n) is in fact half of the points-to predicate for ℓ. We then have a contradiction in the right disjunct in the disjunction, and symmetrically to what we did for wait, we "flip" the disjunction when we close the invariant.

3.4.2 Single-Element Queue

Similar to the specification for the turn sequencer, in the specification for the SEQ (shown in Figure 3.2) we must ensure that no two dequeue or enqueue operations are performed with the same turn. As such, creating a new SEQ gives ownership over two sets of turns: one for enqueue and another one for dequeue. These, $\text{turns}_e(\gamma, n)$ and $\text{turns}_d(\gamma, n)$, denote ownership over all the turns for enqueue and dequeue, respectively, except for the first n such turns. Additionally, $\text{turn}_e(\gamma, n)$ and $\text{turn}_d(\gamma, n)$ represent ownership over the nth turn for enqueue and dequeue, respectively. When calling $\mathsf{enqueue}_{SEQ}$ or $\mathsf{dequeue}_{SEQ}$ with n, the specification requires the corresponding turn.

The representation predicate isSEQ is parameterized by a predicate $Q : \mathbb{N} \to \textsc{Val} \to \mathsf{iProp}$. If x is the nth value added to the queue, then $Q(n, x)$ should hold. Correspondingly, the specification for $\mathsf{enqueue}_{SEQ}$ requires this in its precondition. This in turn allows the specification for $\mathsf{dequeue}_{SEQ}$ to ensure, in its postcondition, that the returned value satisfies the predicate.

Proof of Specification (Sketch). First, we consider the definition of $\text{turns}_e(,$ a)nd $\text{turns}_d(,$.) These are defined to be ownership over all the *even* and the *odd* turns, respectively, except for the first n even or odd numbers:

$$\begin{aligned}
\text{turns}_e(\gamma, n) &\triangleq \text{turns}(\gamma, \{m \in \mathbb{N} \mid \mathsf{even}(m) \wedge 2n \quad\;\; \leq m\}) \\
\text{turns}_d(\gamma, n) &\triangleq \text{turns}(\gamma, \{m \in \mathbb{N} \mid \mathsf{odd}(m) \;\; \wedge 2n+1 \leq m\}) \\
\text{turn}_e(\gamma, n) &\triangleq \text{turn}(\gamma, 2n) \\
\text{turn}_d(\gamma, n) &\triangleq \text{turn}(\gamma, 2n+1)
\end{aligned}$$

Notice how these definitions are only possible because the specification for the underlying turn sequencer allows for ownership over any *infinite* sets of turns.

Next, we define the representation predicate isSEQ by instantiating the turn sequencer specification:

$$\begin{aligned}
R_{\text{SEQ}}(Q, \ell)(n) &\triangleq \begin{cases} \ell \hookrightarrow \mathsf{none} & \text{if } \mathsf{even}(n) \\ \exists v.\, \ell \hookrightarrow \mathsf{some}\ v * Q(\frac{n-1}{2}, v) & \text{otherwise} \end{cases} \\
\text{isSEQ}(\gamma, Q, v) &\triangleq \exists ts, \ell.\, v = (ts, \ell) * \text{isTS}(\gamma, R_{\text{SEQ}}(Q, \ell), ts)
\end{aligned}$$

The predicate $\text{isSEQ}(\gamma, Q, v)$ states that the value v making up the SEQ is a pair of a location ℓ and a turn sequencer ts. The representation predicate for the underlying

$$\textsc{Ghost-alloc}\ \frac{a \in \mathcal{V}}{\Rrightarrow \exists \gamma.\, \boxed{a}^{\gamma}} \qquad \textsc{Own-op}\ \frac{}{\boxed{a}^{\gamma} * \boxed{b}^{\gamma} \dashv\vdash \boxed{a \cdot b}^{\gamma}} \qquad \textsc{Own-valid}\ \frac{}{\boxed{a}^{\gamma} \vdash a \in \mathcal{V}}$$

$$\textsc{set-alloc}\ \frac{X \subseteq \mathbb{N}}{\Rrightarrow \exists \gamma.\, \boxed{\mathbf{1}_X}^{\gamma}} \qquad \textsc{set-sep}\ \frac{X \cap Y = \emptyset}{\boxed{\mathbf{1}_X}^{\gamma} * \boxed{\mathbf{1}_Y}^{\gamma} \dashv\vdash \boxed{\mathbf{1}_X \cup \mathbf{1}_Y}^{\gamma}}$$

$$\textsc{set-disj}\ \frac{X \cap Y \neq \emptyset \qquad \boxed{\mathbf{1}_X}^{\gamma} \qquad \boxed{\mathbf{1}_Y}^{\gamma}}{\mathsf{False}}$$

Figure 3.4: Rules for ghost state and the resource algebra of (infinite) sets.

turn sequencer is instantiated with the resource R_{SEQ}, which states that if the current turn is even, then the location points to **None**, and otherwise it points to a **Some** v. Since the n given to R_{SEQ} is a turn for the turn sequencer, we must convert it to get a turn for the SEQ. This is why R_{SEQ} applies Q to $(n-1)/2$.

With these definitions, the SEQ specification can be derived from the turn sequencer specification.

3.4.3 Ghost state for Turns and Tickets.

We now detail the construction of the ghost state used to represent turns. This section can be skipped—understanding the derived rules presented in the previous two sections suffices for the rest of the paper.

In Iris ghost state is represented using a form of partial commutative monoids called a *resource algebra*. The monoid operation $(\cdot)$ combines elements of the resource algebra and a subset of elements $\mathcal{V}$ are *valid*. In the logic the ownership assertion $\boxed{a}^{\gamma}$ denotes ownership over an element a of some resource algebra for a ghost name γ. Any valid element can be allocated for a fresh ghost name γ (Ghost-alloc), ownership of two elements combine into ownership of their combination per the operation (Own-op), and owned elements are always valid (Own-op).

We want to represent ownership of, potentially, infinite sets of turns. Since ownership of a turn should be exclusive, we want the combination of two sets to be invalid if the sets are not disjoint. The naive approach of letting the elements of the resource algebra be sets and defining the operation as

$$A \cdot B \triangleq \begin{cases} A \cup B & \text{if } A \cap B = \emptyset \\ \bot & \text{otherwise} \end{cases}$$

where $\bot$ is invalid, does not work The operation must be computable, but determining

if two arbitrary infinite sets intersect is not.[2]

Instead, we represent sets using a function resembling a characteristic function. We assemble a resource algebra using three standard resource algebras: the exclusive resource algebra, the option resource algebra, and the resource algebra of functions.

$$\textsc{InfSet}(X) = X \to \textsc{Option}(\textsc{Ex}(1))$$

For the resource algebra of functions, the operation is defined point-wise as $(f \cdot g)(a) = f(a) \cdot g(a)$. Its elements are valid $f \in \mathcal{V}$ if and only if $f(a) \in \mathcal{V}$ for all a in the function's domain. The codomain $\textsc{Option}(\textsc{Ex}(1))$ has two valid elements none and $\mathsf{some}(\mathsf{ex}())$; and one invalid element $\mathsf{some}(\bot)$. These combine in the following way:

$$\begin{aligned}\mathsf{none} \cdot \mathsf{some}(\mathsf{ex}()) &= \mathsf{some}(\mathsf{ex}())\\ \mathsf{some}(\mathsf{ex}()) \cdot \mathsf{some}(\mathsf{ex}()) &= \mathsf{some}(\mathsf{ex}() \cdot \mathsf{ex}()) = \mathsf{some}(\bot)\end{aligned}$$

For any $A \subseteq X$ we can define an element $\mathbf{1}_A \in \textsc{InfSet}(X)$ as

$$\mathbf{1}_A(a) = \begin{cases}\mathsf{some}(\mathsf{ex}()) & \text{if } a \in A\\ \mathsf{none} & \text{if } a \notin A\end{cases}$$

The idea being that $\mathbf{1}_A$ serves as a sort of characteristic function. Given two *disjoint* sets A and B it is then the case that $\mathbf{1}_A \cdot \mathbf{1}_B = \mathbf{1}_{A \cup B}$. On the other hand, given A and B that are not disjoint, then for $a \in A \cap B$ it is the case that $(\mathbf{1}_A \cdot \mathbf{1}_B)(a) = \bot$ and hence the combination is invalid. The three last rules in Figure 3.4 then follow immediately.

With this in place "turns" is defined simply as

$$\mathsf{turns}(\gamma, X) \triangleq \boxed{\mathbf{1}_X}^{\gamma}.$$

With this definition the previously seen rules for turns in Figure 3.3 follow from the rules in Figure 3.4.

3.5 Proof of Contextual Refinement

As mentioned, our main result is that the MPMC queue is a contextual refinement of a coarse-grained queue. We can succinctly state this as the following ReLoC proposition:

$$\models \mathrm{newQueue} \precsim \mathsf{queue}_{\mathsf{CG}} : \forall\alpha.(1 \to \alpha) \times (\alpha \to 1)$$

[2]This type of ghost state was used in the GPS logic [TVD14] to verify a ticket lock. But, since the proof was on paper only, the non-computability of the operation was not an issue. A variant of the GPS proof was later mechanized in the iGPS logic [Kai+17], using a type of ghost state based on cofinite sets and not arbitrary infinite sets.

$$\textsc{rel-lam}\ \frac{\square\left(\forall v_1, v_2.\ [\![\tau]\!]_\Delta(v_1, v_2) \mathrel{-\!\!*} \Delta \models (\lambda x_1.\ e_1)\ v_1 \precsim (\lambda x_2.\ e_2)\ v_2 : \sigma\right)}{\Delta \models (\lambda x_1.\ e_1) \precsim (\lambda x_2.\ e_2) : \tau \to \sigma}$$

$$\textsc{rel-load-r}\ \frac{\ell \hookrightarrow_{\mathsf{s}} v \qquad \ell \hookrightarrow_{\mathsf{s}} v \mathrel{-\!\!*} \models_{\mathcal{E}} e_1 \precsim K[\,v\,] : \tau}{\models_{\mathcal{E}} e_1 \precsim K[\,!\,\ell\,] : \tau}$$

$$\textsc{rel-load-l}\ \frac{\ell \hookrightarrow v \qquad \ell \hookrightarrow v \mathrel{-\!\!*} \Delta \models K[\,v\,] \precsim e_2 : \tau}{\Delta \models K[\,!\,\ell\,] \precsim e_2 : \tau} \qquad \textsc{rel-queue-r}\ \frac{\forall w.\ \mathrm{I}_{\mathrm{CG}}(w, \vec{x}) \mathrel{-\!\!*} \models t \precsim K[\,w\,] : \tau}{\models t \precsim K[\,(\mathsf{newlock}\ (), \mathbf{ref}\ [])] : \tau}$$

$$\textsc{rel-dequeue-r}\ \frac{\mathrm{I}_{\mathrm{CG}}(w, v :: \vec{x}) \qquad (\mathrm{I}_{\mathrm{CG}}(w, \vec{x}) \mathrel{-\!\!*} \models t \precsim K[\,v\,] : \tau)}{\models t \precsim K[\,\mathsf{dequeue}_{\mathrm{CG}}\ w\,] : \tau}$$

Figure 3.5: ReLoC rules (selection).

for any $q > 0$. In such a *refinement judgment* the left expression is called the *implementation* and the right expression the *specification*.

A refinement judgment is manipulated using ReLoC's high-level rules. While the details are not important, a few such rules appear in Figure 3.5. The key principle is that the implementation and specification can be *symbolically executed*, similarly to how it is done in a unary program logic with a Hoare triple or a weakest precondition judgment. The rules REL-LOAD-L and REL-LOAD-R show how to symbolically execute a load operation in the implementation and specification respectively. When the implementation and specification are both values one must show that the values are related. What this means depends on the type of the values; for integers, for instance, it means that they are equal.

Proofs of refinements, like the one above, consist of three parts: (a) Symbolically execute the initialization (*i.e.,* the constructor) of the implementation and specification, and collect the resources. (b) Establish an invariant, using the resources obtained from the first step. The invariant typically relates the internal states of the data structures on the both sides of the refinement. Picking the right invariant is the key to the proof, and we discuss it in details in Section 3.6. (c) Using the invariant, verify refinement of each operation that is part of the data structure. In this stage we verify separately that MPMC's dequeue operation refines the coarse-grained queue's dequeue operation, and similarly for the enqueue operation.

For the first step, due to the polymorphic type $\forall\alpha.(1 \to \alpha) \times (\alpha \to 1)$ of the queue, we must assume a binary predicate R that represents what it means for values of a type τ to be related. Then, we symbolically execute the initialization code for

the MPMC queue and obtain the resources:

$$(\ell_{push} \hookrightarrow 0) * (\ell_{pop} \hookrightarrow 0) * (\ell_{arr} \hookrightarrow_* \text{map } \pi_2\ \mathit{SEQs}) *$$
$$\ast_{(v,\gamma)\in \mathit{SEQs}} \text{isSEQ}(\gamma, Q, v) * \text{turns}_e(\gamma, 0) * \text{turns}_d(\gamma, 0).$$

The first two points-to predicates are from allocating $pushTicket$ and $popTicket$, while the remaining are from allocating the array and the SEQs it contains. We obtain a pointer ℓ_{arr} to the array. For each element of the array, we invoke queue_{SEQ} and, using its specification from Figure 3.2, obtain the value v in the array that satisfies $\text{isSEQ}(\gamma, Q, v) * \text{turns}_e(\gamma, 0) * \text{turns}_d(\gamma, 0)$ for some ghost name γ and a predicate Q of our choosing. The list *SEQs* contain the value and ghost name for each SEQ. We describe the appropriate choice of the predicate Q in the next section.

For the coarse-grained queue, we symbolically execute its initialization using REL-QUEUE-R and obtain the resource $\text{I}_{\text{CG}}(w, [])$. The abstract predicate $\text{I}_{\text{CG}}(w, xs)$ states that w is a coarse-grained queue containing the elements xs.

With these resources we have to prove the remainder of the refinement:

$$\begin{aligned}[\alpha := R] \models\ & (\lambda v.\ \mathsf{enqueue}\ \ell_{arr}\ q\ \ell_{push}\ v, \lambda x.\ \mathsf{dequeue}\ \ell_{arr}\ q\ \ell_{pop}) \\ \precsim\ & (\lambda v.\ \mathsf{enqueue}_{\text{CG}}\ w\ v, \lambda x.\ \mathsf{dequeue}_{\text{CG}}\ w) \\ :\ & (1 \to \alpha) \times (\alpha \to 1).\end{aligned}$$

Naturally, it suffices to show that each operation refines its coarse-grained counterpart. To this end, we use the rule REL-LAM, which intuitively states that two functions are related if they *always* (indicated by the $\square$) evaluate to related values when given related input. This reflects that for two implementations to be related they have to be indistinguishable in any context – including a context that calls the functions several times, potentially in parallel. However, the resources that we obtained from the initialization process cannot be used "as is" as they are ephemeral resources that do not always hold. Hence, we delegate those resources to an *invariant*. The refinement proof of the operations can then proceed by symbolically executing the implementation. Every step can assume and must preserve the invariant. The specification side is stepped forward only at linearization points as it is at these points that the implementation changes its abstract state and hence what specification side queue it corresponds to. At the linearization points we thus apply rules such as REL-DEQUEUE-R. We do not explain the refinement proof of the operations in any more detail as defining a suitable invariant is the most challenging part of the refinement proof and is explained in the next section. However, in Section 3.7 we explain how the external linearization point is handled using our extension to ReLoC.

3.6 Invariant for Refinement Proof

The invariant we use is shown in Figure 3.6. It is non-trivial and key to the refinement proof so we devote this section to explain its parts. Overall, the invariant keeps

$$I(R, \gamma_t, \gamma_m, \gamma_l, q, \ell_{pop}, \ell_{push}, \ell_{arr}, \mathit{SEQs}, w) \triangleq \exists xs_{\mathsf{i}}, xs_{\mathsf{s}} \in \textsc{List}(\textsc{Val}), \mathit{popTicket}, \mathit{pushTicket} \in \mathbb{N}, m \in \textsc{List}(\textsc{Val}).$$

$$\overbrace{\begin{array}{l}\ell_{pop} \hookrightarrow \mathit{popTicket} * \\ \ell_{push} \hookrightarrow \mathit{pushTicket} * \\ \ell_{arr} \hookrightarrow^{\square}_{*} \text{map } \pi_2\ \mathit{SEQs}\end{array}}^{\textbf{Physical state}} * \overbrace{\begin{array}{c}\mathsf{list}^{\gamma_l}(m) * |m| = \mathit{pushTicket} * \\ \mathrm{drop}(\mathit{popTicket}, m) = xs_{\mathsf{i}}\end{array}}^{\textbf{Ghost list}} * \overbrace{|\mathit{SEQs}| = q * \left(\bigstar_{i=0}^{q} I_{SEQ}(i, \mathit{SEQs}_i)\right)}^{\textbf{Invariants for the SEQs}} *$$

$$\underbrace{\mathsf{tokensFrom}^{\gamma_t}(\max(\mathit{popTicket}, \mathit{pushTicket})) * \mathsf{ids}^{\gamma_m}(\mathit{popTicket}) * \left(\bigstar_{i=0}^{\mathit{pushTicket}-1} \mathrm{enqueueObl}(i)\right) * \left(\bigstar_{i=\mathit{pushTicket}}^{\mathit{popTicket}-1} \exists \mathit{id}.\ \mathsf{idsAt}^{\gamma_m}(i, \mathit{id}) * \models - \precsim_{\mathit{id}} \mathsf{dequeue}_{\mathsf{CG}}\ w : -\right)}_{\textbf{Handling of external linearization points}} * \underbrace{\mathrm{I}_{\mathrm{CG}}(w, xs_{\mathsf{s}}) * \bigstar_{(x_i, x_s) \in (xs_{\mathsf{i}}, xs_{\mathsf{s}})} R(x_i, x_s)}_{\textbf{Relation to the coarse-grained queue}}$$

where

$$\begin{aligned}
\mathrm{enqueueObl}(i) &\triangleq \mathsf{token}^{\gamma_t}(i) \vee (\exists \mathit{id}, v_{\mathsf{i}}, v_{\mathsf{s}}.\ \mathsf{idsAt}^{\gamma_m}(i, \mathit{id}) * \mathsf{listAt}^{\gamma_l}(i, v_{\mathsf{i}}) * R(v_{\mathsf{i}}, v_{\mathsf{s}}) * (\models - \precsim_{\mathit{id}} v_{\mathsf{s}} : -)) \\
I_{SEQ}(i, (\gamma, v)) &\triangleq \mathrm{isSEQ}(\gamma, Q(i), v) * \mathrm{turnCtx}(\gamma, i) \\
\mathrm{turnCtx}(\gamma, i) &\triangleq \mathrm{turns}_e(\gamma, \mathrm{affectingOps}(\mathit{pushTicket}, q)) * \mathrm{turns}_d(\gamma, \mathrm{affectingOps}(\mathit{popTicket}, q)) \\
\mathrm{affectingOps}(\mathit{ops}, q) &\triangleq \lfloor \mathit{ops}/q \rfloor + (\text{if } (i < \mathit{ops} \bmod q) \text{ then } 1 \text{ else } 0) \\
Q(i)(j, v) &\triangleq \mathsf{listAt}^{\gamma_l}(jq + i, v)
\end{aligned}$$

Figure 3.6: Invariant for the MPMC queue

track of the physical state of the queues, ensures that the MPMC queue represents a logic-level list of values corresponding to the coarse-grained queue, manages the turns for all the SEQs, and handles the external linearization point.

The invariant is parameterized by the interpretation of the type of values stored in the queue (R), ghost names (γ_t, γ_m, γ_l), the size of the queue (q), the values for the MPMC queue (ℓ_{pop}, ℓ_{push}, ℓ_{arr}, *SEQs*), and the value for the coarse-grained queue (w).

We now cover each different annotated part of Figure 3.6 in turn.

Relation to the coarse-grained queue. The existentially quantified lists of values xs_{i} and xs_{s} represent the abstract state of the MPMC queue and the coarse-grained queue respectively. The state of the coarse-grained queue is tied to xs_{s} by $\mathrm{I}_{\mathrm{CG}}(w, xs_{\mathsf{s}})$ and xs_{i} is tied to the MPMC queue by the rest of the invariant. The separating conjunction over the two lists thus ensures that the abstract states of the two queues are always related at type R. For example, if we store integers in the queue, then the separating conjunction states that the xs_{s} and xs_{i} both contain the same integers.

GHOST-LIST-ALLOC
$$\Rrightarrow \exists \gamma_l.\ \mathsf{list}^{\gamma_l}([])$$

GHOST-LIST-APPEND
$$\frac{\mathsf{list}^{\gamma_l}(xs)}{\Rrightarrow \mathsf{list}^{\gamma_l}(xs +\!\!+ [x]) * \mathsf{listAt}^{\gamma_l}(|xs|, x)}$$

GHOST-LIST-AGREE
$$\frac{\mathsf{listAt}^{\gamma_l}(i, x) \qquad \mathsf{listAt}^{\gamma_l}(i, x')}{x = x'}$$

GHOST-LIST-LOOKUP
$$\frac{\mathsf{list}^{\gamma_l}(xs) \qquad xs_i = x}{\Rrightarrow \mathsf{list}^{\gamma_l}(xs) * \mathsf{listAt}^{\gamma_l}(i, x)}$$

(a) Rules for the ghost list.

TOKENS-ALLOC
$$\Rrightarrow \exists \gamma_t.\ \mathsf{tokensFrom}^{\gamma_t}(0)$$

TOKEN-EXCLUSIVE
$$\frac{\mathsf{token}^{\gamma_t}(n) \qquad \mathsf{token}^{\gamma_t}(n)}{\mathsf{False}}$$

TOKENS-TAKE
$$\frac{\mathsf{tokensFrom}^{\gamma_t}(i)}{\mathsf{tokensFrom}^{\gamma_t}(i+1) * \mathsf{token}^{\gamma_t}(i)}$$

(b) Rules for tokens.

IDENTIFIER-ALLOC
$$\Rrightarrow \exists \gamma_m.\ \mathsf{ids}^{\gamma_m}(0)$$

IDENTIFIER-DECIDE
$$\frac{\mathsf{ids}^{\gamma_m}(n)}{\Rrightarrow \mathsf{ids}^{\gamma_m}(n+1) * \mathsf{idsAt}^{\gamma_m}(n, id)}$$

IDENTIFIER-SKIP
$$\frac{\mathsf{ids}^{\gamma_m}(n)}{\mathsf{ids}^{\gamma_m}(n+1)}$$

IDENTIFIER-AGREE
$$\frac{\mathsf{idsAt}^{\gamma_m}(i, id) \qquad \mathsf{idsAt}^{\gamma_m}(i, id')}{id = id'}$$

(c) Rules for identifier registry.

Figure 3.7: Ghost state rules.

Physical state. The physical state of the queue is rather simple. The queue consists of three locations and the invariant contains points-to predicates for all three. As the pointer to the array never changes we represent it using the persistent points-to predicate $\hookrightarrow_*^{\square}$ [VB21].

Ghost list. We previously explained how the physical state of the queue reveals very little about the actual values stored in the queue. To connect the physical and abstract states, we use a *ghost list* m. It contains *all* values that have been enqueued, in particular, this includes both values that are no longer and not yet physically present in the queue. Thus, while the physical state does not change when enqueue executes its **FAA**, the ghost state does. And, since the linearization point of enqueue is when it increments *pushTicket*, the number of values that have been added to the queue is always exactly *pushTicket*. Hence, the ghost list is connected with the physical state in part from the requirement that its length is equal to the value of

pushTicket.

Ownership of a ghost list xs is denoted by a proposition $\mathsf{list}^{\gamma_l}(xs)$. Ghost list can grow over time, when the new values are enqueued at the end. This is in fact the *only* way in which the ghost list can change, and that means that once a value is part of the ghost list it says there. To that extent, we have a *persistent* predicate $\mathsf{listAt}^{\gamma_l}(i, x)$, which denotes the knowledge that the ith element of the list (corresponding to the the ith value added to the queue) is x. The ghost list satisfies a number of proof rules presented in Figure 3.7a; these rules are sufficient to carry out the proof.

In the invariant we can see the ownership of the ghost list ($\mathsf{list}^{\gamma_l}(m)$) of the size *pushTicket* ($|m| = \mathit{pushTicket}$). Moreover, if we remove the first *popTicket* elements from m, then the remaining list is exactly the abstract state of the queue ($\mathrm{drop}(\mathit{popTicket}, m) = xs_{\mathsf{i}}$). This makes sense since the ghost list contains all values that have been enqueued and we remove exactly those that have also been dequeued. Note that when $\mathit{pushTicket} \leq \mathit{popTicket}$, then the above implies that xs_{i} is empty.

Invariants for the Single-Element Queues. For each of the q single-element queues in the array the invariant needs to include the invariant for the SEQ and to manage its turns.

We need to instantiate the invariant for each SEQ with the predicate Q that holds for the values in it. Recall that Q is parameterized both by the value in the queue and its corresponding turn. We use this to define a Q that relates the value in the queue to the "right" value in the ghost list:

$$Q(i)(j, v) \triangleq \mathsf{listAt}^{\gamma_l}(jq + i, v),$$

were q is the capacity of the queue and $0 \leq i < q$ is the index of the particular SEQ. For the jth element v added to this SEQ we can then calculate the position of this element in the whole queue as $jq + i$, which we record using the ghost list.

In addition to picking the predicate Q, we must keep track of the turns for each SEQ. We must calculate these turns based on the current value of *popTicket* and *pushTicket*. The affectingOps function aids in this. Given the "global" count *ops* of an operation (dequeue or enqueue), it calculates how many times the SEQ in question was affected.

Handling of external linearization points. The part of the invariant for handling the external linearization point is rather intricate. For the ith pair of operations, either enqueue or dequeue arrives first. In the former case, the invariant should allow both enqueue and dequeue to carry out their own linearization point. In the latter case, the invariant must facilitate handling of the external linearization point.

To do this we must intuitively encode the following: when dequeue opens the invariant around its **FAA** it must transfer the requisite resources into the invariant that will allow another thread to carry out its linearization point. Then, when enqueue opens the invariant around its **FAA** it should be forced to carry out the corresponding dequeue's linearization point and transfer the result into the invariant.

Later, dequeue needs to open the invariant again, conclude that its linearization point has been carried out, and be able to transfer the resources for the executed linearization point out of the invariant.

Came-first token. To keep track of which operation came first we use *tokens*—custom ghost state theory similar to the one that we constructed for turns earlier. The rules for this ghost state are in Figure 3.7b. The ith dequeue or enqueue that comes first will be able to take the token $\mathsf{token}^{\gamma_t}(i)$. Hence, owning $\mathsf{token}^{\gamma_t}(i)$ proves that an operation came before its corresponding counterpart. The invariant owns all the tokens where neither operation has taken a ticket:

$$\mathsf{tokensFrom}^{\gamma_t}(\max(\mathit{popTicket}, \mathit{pushTicket})).$$

To see how this allows the operation that arrives first to take a ticket, note that when enqueue and dequeue open the invariant around their **FAA**, they will close the invariant by using $\mathit{pushTicket} + 1$ and $\mathit{popTicket} + 1$, respectively, for the existential variable that they introduced. If enqueue comes first then $\mathit{popTicket} \leq \mathit{pushTicket}$. Hence $\max(\mathit{popTicket}, \mathit{pushTicket})$ is equal to *pushTicket*, and only $\mathsf{tokensFrom}^{\gamma_t}(\mathit{pushTicket} + 1)$ is required for closing the invariant and one token can be kept by enqueue per the rule TOKENS-TAKE. On the other hand, if enqueue is last, then $\mathit{pushTicket} < \mathit{popTicket}$ and $\max(\mathit{popTicket}, \mathit{pushTicket}) = \max(\mathit{popTicket}, \mathit{pushTicket} + 1)$. Thus when closing the invariant, all the tokens are required and none can be kept. For dequeue the situation is symmetric. All in all, this means that this construction ensures that ith operation that comes first can take the ith token.

Identifier registry. Concretely, for enqueue to carry out its corresponding dequeue's linearization point means that it should step dequeue's specification forward. To this end, $\models - \precsim_{\mathit{id}} e : -$ represents that some thread, identified by *id*, needs to show that its implementation refines e. This resource is part of the extensions that we make to ReLoC which is explained in greater detail in Section 3.7 and the approach here is an instance of the general proof pattern identified in Section 3.7.1. For now, it suffices to know that the state of dequeue's specification is associated with an identifier, *id*, and that dequeue needs a way to ensure that enqueue steps precisely the specification with that identifier forward. To support this, the invariant contains a resource that lets the ith dequeue *register* which identifier it has. The rules for this construction are shown in Figure 3.7c. The resource $\mathsf{ids}^{\gamma_m}(n)$ represents that only the n first dequeue operations might have registered an identifier. The persistent resource $\mathsf{idsAt}^{\gamma_m}(i, \mathit{id})$ represents the knowledge that the ith dequeue has registered the identifier *id*.

Pending dequeues. When $\mathit{pushTicket} < \mathit{popTicket}$, there are $\mathit{popTicket} - \mathit{pushTicket}$ dequeue operations blocked, waiting for a value to read. These blocked dequeues are exactly those with external linearization points, and when an enqueue comes

along, it should carry out the corresponding dequeue's linearization point. To this end, enqueue needs some resources, which we store in the invariant:

$$\mathop{\Large *}_{i=\mathit{pushTicket}}^{\mathit{popTicket}-1} \exists \mathit{id}.\ \mathsf{idsAt}^{\gamma_m}(i, \mathit{id}) * (\models - \precsim_{\mathit{id}} \mathsf{dequeue}_{\mathrm{CG}}\ w : -)\,.$$

This reads: every ith dequeue operation (where $\mathit{pushTicket} \leq i < \mathit{popTicket}$), has stored some identifier in the identifier registry and we have the corresponding right refinement, which is ready to invoke dequeue on the coarse-grained queue.

Enqueue obligation. The final piece in the invariant is

$$\mathop{\Large *}_{i=0}^{\mathit{pushTicket}-1} \mathrm{enqueueObl}(\gamma_l, \gamma_t, \gamma_m, i).$$

Since enqueue increments *pushTicket* its proof must close the invariant with *pushTicket*+ 1 for the existential *pushTicket* and thus the big separating conjunction ranges over one additional conjunct. Hence, in the proof one must show

$$\mathrm{enqueueObl}(\gamma_l, \gamma_t, \gamma_m, \mathit{pushTicket})$$

and one should think of $\mathrm{enqueueObl}(\gamma_l, \gamma_t, \gamma_m, i)$ as something which enqueue is *obliged to produce* when it takes the ith ticket. Since the proposition enqueueObl is a disjunction, there are two ways for enqueue to meet this obligation. When enqueue comes first, the obligation is trivial: it can take the token $\mathrm{token}^{\gamma_t}(\mathit{pushTicket})$, and this is exactly the first disjunct. If, on the other hand, enqueue is last, then there is no way to show the first disjunct and the only option is to show the second disjunct, which involves carrying out the dequeue's linearization point.

3.7 Extending ReLoC with Support for External Linearization Points

As mentioned earlier, to show that an operation refines its specification with ReLoC, one symbolically executes the implementation up to its linearization point. At the linearization point, the specification is then symbolically executed to reflect the change in the state of the implementation at the linearization point. However, for an external linearization point this approach does not work as the linearization point does not happen during the symbolic execution of the operation; instead, it happens during the symbolic execution of some other operation. Intuitively, it is when we symbolically execute this second operation that we should symbolically execute the specification. This kind of reasoning is not supported by the current ReLoC rules.

To support such reasoning we extend ReLoC with additional rules, a selection of which is shown in Figure 3.8. We explain how these rules are used by using the external linearization point in the MPMC queue as an example.

$$\textsc{rel-split}\quad \frac{\forall id.(\models - \precsim_{id} e_2 : -) \mathrel{-\!\!*} \Delta \models e_1 \precsim_{id} - : \tau}{\Delta \models e_1 \precsim e_2 : \tau} \qquad \textsc{rel-combine}\quad \frac{\models - \precsim_{id} e_2 : - \qquad \Delta \models e_1 \precsim e_2 : \tau}{\Delta \models e_1 \precsim_{id} - : \tau}$$

$$\textsc{rel-load-l'}\quad \frac{\ell \hookrightarrow v \qquad (\ell \hookrightarrow v \mathrel{-\!\!*} \Delta \models K[\,v\,] \precsim_{id} - : \tau)}{\Delta \models K[\,!\ell\,] \precsim_{id} - : \tau}$$

$$\textsc{rel-right-load}\quad \frac{\ell \hookrightarrow_{\mathsf{s}} v \qquad \models - \precsim_{id} K[\,!\ell\,] : -}{\Rightarrow\!\!\Vdash (\ell \hookrightarrow_{\mathsf{s}} v) * (\models - \precsim_{id} K[\,v\,] : -)} \qquad \textsc{rel-dequeue-detached}\quad \frac{\mathrm{I}_{\mathrm{CG}}(w, v :: \vec{x}) \qquad (\models - \precsim_{id} \mathsf{dequeue}_{\mathrm{CG}}\ w : -)}{\Rightarrow\!\!\Vdash \mathrm{I}_{\mathrm{CG}}(w, \vec{x}) * (\models - \precsim_{id} v : -)}$$

Figure 3.8: Selected rules for external linearization points.

When we show the dequeue refinement, we symbolically execute the implementation until we reach the expression $\mathbf{FAA}(\mathit{popTicket}, 1)$. At this point, if $\mathit{pushTicket} \leq \mathit{popTicket}$ then the linearization point is external, and the specification should be symbolically executed during the corresponding enqueue operation. To this end, we apply the rule REL-SPLIT which *splits* a refinement judgment into a *left refinement* of the form $\models e_1 \precsim_{id} - : \tau$ and a *right refinement* of the form $\models - \precsim_{id} e_2 : -$. These represent the state of the implementation and the specification, respectively. When we split a refinement judgment, we naturally want to keep track of the fact that the two parts originate from the same refinement judgment. The split refinement judgments is therefore parameterized by an identifier *id* from an opaque set Id of identifiers. Since the right refinement $(\models - \precsim_{id} e_2 : -)$ appears on the left-hand side of a wand $\mathrel{-\!\!*}$ in REL-SPLIT we can assume it as a *resource*.[3] Hence, after applying REL-SPLIT we obtain the right refinement $\models - \precsim_{id} \mathsf{dequeue}_{\mathrm{CG}}\ w : -$ for some *id* as a proposition. We transfer this right refinement into the invariant, as described in the previous section.

Our goal is now a left refinement with the state of the implementation. To be able to symbolically execute the left refinement, we have generalized all the rules in ReLoC for symbolically executing the implementation in a refinement judgment such that they apply both in the presence and in the absence of a specification side. The rule REL-LOAD-L' show the generalized rule REL-LOAD-L specialized to a left refinement. We can hence continue symbolically executing the implementation up to the point where dequeue reads a value from its designated SEQ. Intuitively, by now an enqueue operation must have carried out the linearization point, *i.e.,* symbolically executed the right refinement that we placed inside the invariant (we explain how this is done below). We know this, as the enqueue obligation $\mathrm{enqueueObl}(i)$ corresponding to our dequeue operation must have been fulfilled in the invariant. And since we came first and thus were able to take the came-first token, we can conclude that

[3] This treatment of the right refinement stems from the "specifications-as-resources" approach of Turon et al. [TDB13] and is present in the model of ReLoC as well.

the obligation contains $\models - \precsim_{id} v_s : -$. The identifier registry ensures that the id of this right refinement matches the left refinement in our goal. We take the right refinement out of the invariant in exchange for our came-first token. To "re-insert" this right refinement into our goal we use the rule REL-COMBINE. This rule acts as a counterpart to REL-SPLIT and combines a right refinement in the context with a left refinement in the goal. After applying this rule our goal is again a standard refinement judgment, but, with a fully evaluated specification. The remaining part of proof can be completed using existing rules in ReLoC.

We now consider how the external linearization point is handled in the refinement proof of enqueue. We symbolically execute the implementation up to **FAA**(*pushTicket*, 1). If *pushTicket* $<$ *popTicket* then the corresponding dequeue's linearization point is external and we must step its specification forward. In the invariant this corresponds to producing a particular enqueue obligation enqueueObl(i). Since we do not have a came-first token for this obligation, we must produce the right refinement $\models - \precsim_{id} v_s : -$ for some *id* and v_s. We can do this by symbolically executing the right refinement $\models - \precsim_{id} \mathsf{dequeue}_{\mathsf{CG}}\ w : -$ present in the invariant by using a new set of rules that applies to a right refinement in one's context (REL-RIGHT-LOAD is one such rule). From these rules the required REL-DEQUEUE-DETACHED can be derived.

3.7.1 Proof Pattern for External Linearization Points

Summarizing, the generally applicable pattern for external linearization points is as follows. One must establish an invariant that allows transferring a right refinement between the operation with an external linearization point and the operation during which the external linearization point occurs. In the refinement proof of the operation with the external linearization point, one symbolically executes the implementation up to the point where another operation may carry out the linearization point. At this point, one applies REL-SPLIT and transfers the right refinement into the invariant. Then, one uses the generalized symbolic execution rules to step the implementation forward until the point where it is certain that the external linearization point has occurred. At that point, one extracts the advanced right refinement from the invariant and applies REL-COMBINE to merge it back into the left refinement. In the refinement of the operation during which the linearization point happens, one steps forward the implementation to the point where the external linearization point occurs, take a right refinement from the invariant, steps it forward using the symbolic execution rules for a right refinement, and puts it back into the invariant afterward.

This approach is general and in our Coq formalization we have applied it to two other examples of data structures with external linearization points: a version of the elimination-backoff stack from [HSY04], and the red flags versus blue flags example from [Tur+13a].

3.7.2 Changes to the ReLoC Model

We now describe how the left and right refinement judgments are defined and how the rules are encoded. The changes that we make to ReLoC rely on exposing and encapsulating a suitable amount of capabilities already present in the underlying model (described in [FKB20a]) and thus the soundness result of ReLoC is unaffected.

Recall, from [FKB20a], that the refinement judgment is defined[4] as:

$$\models e_1 \precsim e_2 : \tau \triangleq \forall j, K.\\ \{\mathsf{specCtx} * j \Rightarrow K[e_2]\}\, e_1\, \{v.\ \exists v'.\ j \Rightarrow K[v'] * [\![\tau]\!](v, v')\}$$

That is, it is a particular Hoare triple for the left-hand side expression e_1, specifications for which talk about the thread-pool resource $j \Rightarrow K[e']$ and an invariant specCtx (the latter can be ignored). These thread-pool resources are part of the ghost thread-pool: the key element in the definition of the model.

In order to obtain a right refinement, we package this thread-pool resource $j \Rightarrow K[e']$ together with the invariant specCtx. The identifier for such a refinement is then a pair of the thread id j and the evaluation context K. This hides all the unnecessary details:

$$\mathsf{Id} \triangleq \{j : nat, K : ctx\}$$
$$\models - \precsim_{id} e_2 : - \triangleq \mathsf{specCtx} * id.j \Rightarrow id.K[e_2]$$

Finally, the left refinement judgment is obtained by taking the definition of a normal refinement, and stripping away the information about the right refinement from the precondition in the Hoare triple:

$$\models e_1 \precsim_{id} - : \tau \triangleq \{\mathsf{True}\}\, e_1\, \{v.\exists v', id.j \Rightarrow id.K[v'] * [\![\tau]\!](v, v')\}$$

In the Coq formalization, we formalize a generalized definition that combines the left refinement $\models e_1 \precsim_{id} - : \tau$ and the regular refinement $\models e_1 \precsim e_2 : \tau$. This allowed us to make *tactics* that automatically apply the correct rule, depending on whether we are proving a left refinement or a regular one. Tactics allow the user to interactively carry out refinement proofs, without worrying too much about the low-level details of the rules. For example, the user can invoke a tactic `rel_load_l`, that applies either REL-LOAD-L or REL-LOAD-L', depending on what is applicable. The tactics automatically determine the evaluation context K and the resource $\ell \mapsto v$ (if available).

3.8 Discussion: Conclusion, Related and Future Work

We now discuss related and future work along two dimensions: (1) specification and verification of the MPMC queue, and (2) and the extension of ReLoC with support for reasoning about external linearization points.

[4] For reasons of clarity, the definitions given here are presented without masks and view-shifts; see [FKB20a] for details.

Wrt. (1), ours is the first formal specification and verification of the highly-efficient and practical MPMC queue algorithm used in Meta's Folly library. Thanks to our modular approach we also get specifications for its submodules. For example, our specification for the turn sequencer can also be used to verify other clients than the SEQ; indeed, in our Coq formalization we have used the turn sequencer to implement and verify a ticket lock.

Recently a similar bounded queue was considered by Mével and Jourdan [MJ21a]. Their motivation, approach, and challenges are different from ours. They specified the queue they considered in terms of *logically atomic triples*, while we prove contextual refinement. They verify the queue with respect to the weak memory model of multicore OCaml by using the Cosmo logic [MJP20] , while we assume sequential consistency (a simplification compared to the C++ memory model). Their challenges stem from the complexities of weak memory, but the queue operations they verify are comparatively simpler than ours and have only fixed linearization points.

Wrt. (2), we emphasize that our extensions to ReLoC are generally applicable and suitable to support mechanized verification of a wide range of fine-grained concurrent algorithms with external linearization points. Indeed in our Coq formalization we have applied our methodology to two other examples: a version of the elimination-backoff stack from [HSY04], and the red flags versus blue flags example from [Tur+13a].

The most closely related work not already discussed earlier in the paper is Liang and Feng's local rely/guarantee-style relational logic [LF13b], which can be used to show refinement for fine-grained concurrent algorithms with non-fixed linearization points, including algorithms with external linearization points. In contrast to Liang and Feng's logic, our extended version of ReLoC supports a more expressive programming language with higher-order functions (we use them to write out the constructors as closures encapsulating the internal state of the queue). Recently, a variant of Liang and Feng's logic has been formalized in Coq by Zou et. al. [Zou+19], for the purposes of verifying a concurrent file system with external linearization points. They extend the logic of Liang and Eng with abstract "helping" mechanism, which allows one thread to carry out linearization points of several other threads. It would be interesting to obtain the Coq formalization and investigate *a)* how a proof of the MPMC queue in that setting would compare with our proof in ReLoC; *b)* whether the mechanism of helpers can be implemented and applied in ReLoC.

Another relational program logic that was used for verifying algorithms with external linearization points is CaReSL [TDB13], which also supports a functional programming language with higher-order functions and higher-order state. As mentioned, our approach to handling external linearization points is closely inspired by the "specifications-as-resources" approach of CaReSL present in the model of ReLoC.

In addition to (relational) program logics, there are many alternative methods for verifying concurrent data structures with external linearization points, including generic methods like *interval reasoning* [DD13; DDH12], or data structure specific

methods like aspect-oriented proofs for concurrent queues [Cha+15]. We refer an interested reader to the survey article by Dongol and Derrick [DD15].

Other alternatives to contextual refinement include logically atomic Hoare triples [JP11; Jun+15a; RDG14] (which were used in the aforementioned work [MJ21a]) and HOCAP-style specifications [SBP13], which aim at internalizing the notion of atomicity. In particular, the Iris notion of logically atomic triples is a popular correctness criterion that can handle data structures with external linearization points [Jun+15a]. A logically atomic triple is a special kind of Hoare triple for a single program—unlike ReLoC's refinement judgment which relates an implementation to a specification. One strong point of logically atomic triples is that they are easy to use and build upon inside the Iris logic. On the other hand, they do not yield as strong results outside the logic as ReLoC's refinement judgment, which implies contextual refinement. Recently, logically atomic triples have been shown to imply linearizability[Bir+21], but only in a simpler first-order setting.

Contextual refinement is related to another popular correctness for concurrent algorithms: *linearizability* [HW90]. While there is an abundance of methods for verifying or checking linearizability, it has mainly been considered for first-order languages and with certain restrictions placed on how clients can interact with the concurrent algorithm.[5] To the best of our knowledge, linearizability has not even been properly defined for a programming language with features that we consider here (*e.g.*,higher-order functions, higher-order state, fork-based concurrency).

[5] In such a setting contextual refinement and linearizability are equivalent [Fil+10b].

Chapter 4

Spirea: A Mechanized Concurrent Separation Logic for Weak Persistent Memory

Abstract

Weak persistent memory (a.k.a. non-volatile memory) is an emerging technology that offers fast byte-addressable durable main memory. A wealth of algorithms and libraries has been developed to explore this exciting technology. As noted by others, this has led to a significant verification gap. Towards closing this gap, we present Spirea, the first concurrent separation logic for verification of programs under a weak persistent memory model. Spirea is based on the Iris and Perennial verification frameworks, and by combining features from these logics with novel techniques it supports high-level modular reasoning about crash-safe and thread-safe programs and libraries. Spirea is fully mechanized in the Coq proof assistant and allows for interactive development of proofs with the Iris Proof Mode. We use Spirea to verify several challenging examples with modular specifications. We show how our logic can verify thread-safety and crash-safety of non-blocking durable data structures with null-recovery, in particular the Treiber stack and the Michael-Scott queue adapted to persistent memory. This is the first time durable data structures have been verified with a program logic.

4.1 Introduction

In the traditional storage hierarchy programmers can choose between fast, but volatile, main memory and non-volatile, but slower, secondary storage. Persistent memory (a.k.a. non-volatile memory) is an exciting emerging technology that, uniquely, offers both fast random access at byte granularity and persistence of data in the absence of power and across system crashes. It thus shakes up the traditional storage hierarchy with a new abstraction: storage that is suitable both as main memory and as durable storage of data.

A wealth of algorithms, libraries, and tools have been developed for persistent memory, exploring the new potential. This includes durable data structures [Cai+21; Fri+18], memory allocators [Sch+15], garbage collectors [Cai+20], transactions [RCF21; VTS11], key-value stores [Che+20; Kai+19], and language-level support for persistent memory [Geo+20], just to mention a few. An important class of data structures that is new and unique to persistent memory is durable data-structures with *null-recovery* [IMS16]. These reside in persistent memory and are preserved across crashes with *no recovery being needed* after a crash to maintain their consistency.

Ensuring correctness when programming for persistent memory is, however, extremely challenging. Since data stored in persistent memory is expected to be permanent, programs for persistent memory must be *crash-safe*. Thus, programmers must ensure that if the system crashes (which can happen non-deterministically at any time, *e.g.*,due to power failure) then, after the crash, the content of the persistent memory should be in a consistent state from which recovery is possible.

Moreover, due to the volatile caches on contemporary CPUs, writes to persistent memory are buffered. They occur *asynchronously* and may reach persistent memory in a different order than the one in which they were carried out. This *persistent memory order* (or persist order) does not coincide with the *weak memory order*, the order in which the CPU guarantees that writes by one thread are made visible to other threads. Hence, a program can be correct for weak memory (by taking into account the weak memory order), but not correct for persistent memory (by failing to take the persistent memory order properly into account). To tame this non-determinism, modern instruction sets such as x86 and ARM offer various flush and fence instructions, which programmers can insert between writes to enforce a desired persist order. These instructions are expensive, though, and should only be used when necessary.

One solution to ensure correctness in the presence of these challenges is, of course, to formally verify programs for persistent memory using a program logic. However, as Raad et al. [RLV20] identified, there is a significant *verification gap*: The development of algorithms and libraries for persistent memory is far ahead of formal verification techniques for persistent memory. As a first step towards closing this gap two program logics have been developed: Persistent Owicki-Gries (POG) [RLV20] and Pierogi [Bil+22]. Both are adaptations of the Owicki-Gries proof system and for reasoning about programs under the machine-level x86-TSO memory model. However, since these logics are based on Owicki-Gries they only support a very simple first-order sequential programming language and do not include features such as separation, (user defined) ghost state, higher-order reasoning, and abstract specifications. This results in a lack of modularity that is evident, for instance, in [RLV20], where to verify an example using a lock, the lock and the client of the lock are verified together using a global invariant with knowledge about the internals of both. It is not possible to give the lock an abstract specification, verify it in isolation, and reuse the specification with multiple clients. In contrast, modern concurrent separation logics (CSLs), such as Iris [Jun+18a], scale to much richer programming languages and support the aforementioned features. We thus think that the next

step to closing the verification gap is to develop a CSL for persistent memory, and that is exactly what we do in this paper.

4.1.1 Challenges

Prior work has explored the application of CSL to weak memory and to persistency *individually*. The RSL and GPS logics has spawned a line of logics for weak (but not persistent) memory [DV16; Kai+17; TVD14; VN13]. The Perennial logic, which is a state-of-the-art CSL for reasoning about crash-safety, and its predecessor Crash Hoare Logic applies to programs that use durable secondary storage (but without any weak behaviors) [Cha22; Cha+19; Cha+21; Che+16]. These logics have been successful in their respective domains but no CSL has been developed for the *weak persistency* found in persistent memory. As persistent memory combines challenging aspects from both weak memory and persistency a natural approach is to learn from the above-mentioned logics and try to adapt their techniques into a logic for persistent memory. As it turns out, there are however serious obstacles to such an endeavor:

Non-deterministic crashes In a strong persistency model, such as the one considered for Crash Hoare Logic and Perennial, crashes are *deterministic*. This means that if a crash occurs at a given program point the state of the machine after the crash is uniquely determined by its state before the crash at that program point. The durable storage is completely unaffected by the crash whereas the content of volatile memory is entirely lost. At the program logic level this means that some logical resources are kept unchanged at a crash while others are discarded. Perennial includes a *post-crash* modality, $\langle \mathsf{PC} \rangle$, that carries out this transformation. All rules for their post-crash modality have the form $R \vdash \langle \mathsf{PC} \rangle\, R$, which means that the resource R is preseved during a crash. If a resource P is lost at a crash this is simply encoded by having no such rule for P.

For persistent memory the persistency model is weak due to the asynchronous nature of writes and fences. This means that the crash step is *non-deterministic*. As such, resources are not merely kept or lost at a crash; instead they are non-deterministically kept, discarded, or *changed*. Hence, the straightforward behavior of Perennial's post-crash modality is no longer sufficient and its model, which relies on changing ghost names for lost resources, is not applicable either! We thus introduce a more sophisticated post-crash modality and prove it sound using a more subtle model.

Sound invariants It is well-known that Iris-style invariants are unsound for weak memory. To overcome this, CSLs for weak memory have had to restrict invariants in various ways. One approach taken by GPS, iGPS and iRC11 is to associate invariants with specific locations and only allow access to their content when physically synchronizing with the location. We observe that in a persistent memory setting even these restricted invariants allows for resource transfer that

is unsound for persistent memory. In particular, in weak memory if a RMW (read-modify-write) operation is successful then the overwritten value can never be read again by another RMW operation. The weak memory invariants rely on this property for certain types of resource transfer. But, in persistent memory, a write made by an RMW operation might be lost at a crash, and the overwritten value will then be observable again after the crash.

Additionally, we want invariants that are strong enough to handle durable data structures with null-recovery. The obvious way to encode at the logic level that a data-structure is preserved across crashed is to say that its invariant (inside its representation predicate) is preserved under the post-crash modality. However, it is not clear how an Iris invariant can soundly interact with a post-crash modality. Indeed, in Perennial, which uses Iris invariants, one cannot use the post-crash modality to establish that an invariant holds after a crash. Instead, Perennial relies on recovery code to establish *new* invariants after a crash, but this approach does not work for null-recovery where there is no recovery code.

A somewhat subtle point is that the issues with reconciling Iris invariants and crashes also pose challenges regarding modeling of the logic. Prior Iris-based logics for weak memory use Iris invariants internally to model their more restrictive user-level invariants. But if invariants can not survive crashes, then they can not be used in the model either.

Persistent memory instructions Persistent memory models usually involve some combination of flushes and fences to restrict the persist order when necessary. These instructions are specific to persistent memory and are not addressed by prior separation logics. We consider a weak flush instruction that may be reordered with respect to other instructions up to a fence. As noted by Raad et al. [RLV20] such a flush instruction is difficult to reason about as its effect does not take place at the program point of the flush. As for fences we consider both asynchronous fences and synchronous fences.

4.1.2 Our Contributions

This paper contributes Spirea, the first CSL for weak persistency in general and persistent memory in particular. We use the *explicit epoch persistency* model by Izraelevitz et al. [IMS16].[1] This model is a slight generalization of the x86 and the ARM persistency models which can be efficiently implemented on both processors. As the model is slightly weaker than x86 and ARM, programs that are proven correct for this model are correct for both x86 and ARM. Similarly, reasoning principles that apply for this model are more general and are sound also for x86 and ARM. As such, the ideas in Spirea are generally applicable and can also be used, for instance, in logics specifically for x86 and ARM. In Section 4.2 we give an intuitive account of

[1] Not to be confused with the (implicit) epoch persistency model which cannot be efficiently implemented on x86 or ARM.

the persistency model as well as the consistency model and explain the verification challenges in more detail. Izraelevitz et al. [IMS16] define the explicit epoch persistency model in a declarative style, as a number of ordering constraints on abstract histories. Such a formulation is not well-suited for reasoning in a CSL, so we recast their model as a view-based small-step operational semantics (see Section 4.7.1) that can be used with the Perennial and Iris logical frameworks. As our focus in this paper is squarely on the logic we do not establish a formal correspondence between Izraelevitz et al.'s formulation and ours but instead leave this to future work.

Our logic improves the state-of-the-art both in terms the programming language features it supports, the expressivity and power of the logic, and in the scope of the case studies we have verified. Our programming language λ_{pmem} includes many features that are not supported by the Owicki-Gries based logics, most importantly: dynamic allocation of references, dynamic forking of threads, functions (including higher-order recursive functions and closures), and compound data types. As for the logic, Spirea is a higher-order separation logic and includes all the usual features in Iris based separation logics (except for those that are unsound in our setting). For reasoning about crashes Spirea contains features equivalent to those of Perennial. We cover this background in Section 4.4.

To tackle the above-mentioned challenges, Spirea includes the following key innovations:

1. A *resource changing posts crash modality* that can account for the non-deterministic changes in resources at crashes under weak persistency. Our post-crash modality supports rules of the form $R \vdash \langle \mathsf{PC} \rangle\, R'$, where R' reflects how R is non-deterministically affected by the crash. We make this possible by modelling our post-crash modality using an *exchange resource*. This can be seen as a generalization of the model of Perennial's post-crash modality: the Perennial model is the special case where the exchange resource is the empty resource.

2. *Crash-aware invariants*, which, in contrast to Iris-style and GPS-style invariants, are sound under weak persistency. Soundness of Spirea crash-aware invariants relies on having novel proof rules for transfer of resources in and out of invariants. Our Spirea invariants are *crash-aware*, meaning that they can be preserved under our post-crash modality and thus facilitate resource transfer between code executing before and after a crash. This is the first time a separation logic contains invariants that can be used to this end. We devise a novel model for our invariants that does *not* rely on Iris invariants.

3. An assortment of features to handle persistent memory instructions: *Post-fence modalities*, a *post-crash flush modality*, and *state lower-bounds* w.r.t. fences. These work in tandem to reason about weak flushes and synchronous and asynchronous fences.

We explain these in depth in Section 4.6 where we give a high level introduction to Spirea, explain its design, and present several examples.

$$
\begin{aligned}
v \in \textsc{Val} ::= {} & () \mid i \in \mathbb{Z} \mid \ell \in \mathit{Loc} \mid \mathsf{True} \mid \mathsf{False} \mid (v, v) \\
& \mid \mathbf{inj}_1\ v \mid \mathbf{inj}_2\ v \mid \mathbf{rec}\ f(x) = e \mid \cdots \\
e \in \textsc{Expr} ::= {} & x \mid v \mid \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e \mid (e, e) \mid \pi_1\, e \mid \pi_2\, e \mid \mathbf{inj}_1\ e \mid \mathbf{inj}_2\ e \mid e\ e \mid \cdots \\
& \mid \mathbf{match}\ e\ \mathbf{with}\ \mathbf{inj}_1\ x \Rightarrow e \mid \mathbf{inj}_2\ x \Rightarrow e \mid \mathbf{fork}\,\{e\} \\
& \mid \mathbf{ref}_a\ e \mid !_a\, e \mid e :=_a e \mid \mathbf{CAS}\ e\ e\ e \mid \mathbf{FAA}\ e\ e\ e \qquad \text{for } a \in \{\mathsf{na}, \mathsf{at}\} \\
& \mid \mathbf{flush}\ e \mid \mathbf{fence} \mid \mathbf{fence}_{\mathbf{sync}}
\end{aligned}
$$

Figure 4.1: The syntax of λ_{pmem}

Spirea and its high-level reasoning rule are modelled on top of a lower-level logic called BaseSpirea. This logic, in turn, is modelled using an instantiation of the Perennial program logic and using the Iris base logic. In Section 4.7 we state the soundness result in terms of the operational semantics. We also give an overview of the semantic model and the proof of soundness to the extent that space permits. For the full details regarding the model and the soundness proof we refer the reader to our mechanization.

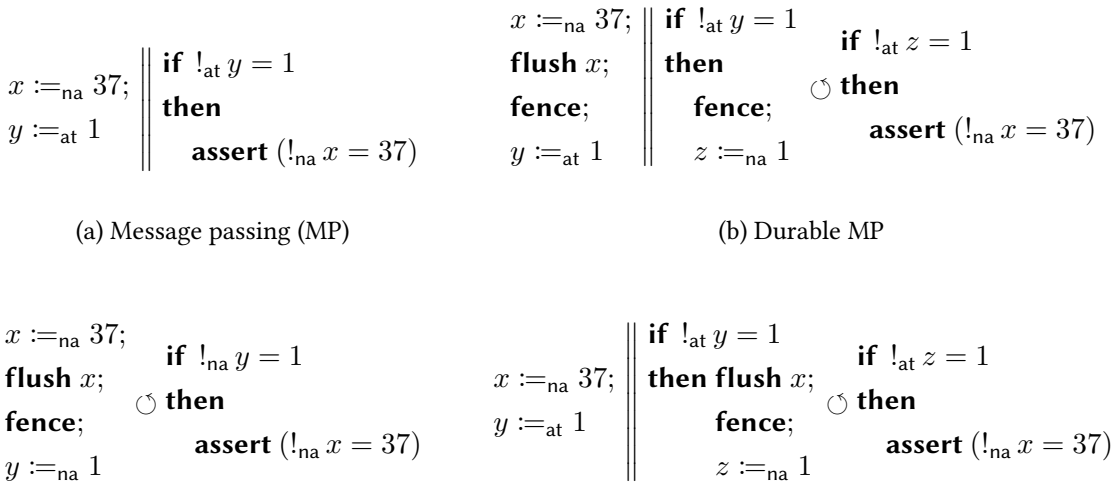
Spirea and all our results are fully mechanized in the Coq proof assistant. The mechanization allows for interactive development of proofs using the Iris proof mode. The development is available online at `https://github.com/logsem/spirea` and as an artifact [VB23b]. We have used the mechanization of our logic to formally verify a range of examples and case studies. We cover a number of these in Section 4.8. The case studies demonstrate how our logic is capable of verifying tricky synthetic examples, that it can give modular and compositional specifications to thread-safe and crash-safe libraries, and even verify entire durable data structures with null-recovery. For the latter we have verified crash-safety and thread-safety of both a durable version of the Treiber stack and the Michael-Scott queue. This is the first time durable data structures have been verified with a program logic.

In Section 4.9 we discuss related and future work.

4.2 Persistent Memory Verification Challenges

Before we can introduce Spirea we must first understand the kinds of programs that it aims to verify correctness of and the challenges involved in this. To this end we introduce our programming language λ_{pmem}. Its syntax is seen in Figure 4.1. We use highlighted text to indicate the parts of the language that are related to persistent memory only. Loosely speaking, if we erased those parts we would get a language for weak, but not persistent, memory.

λ_{pmem} is a lambda-calculus with standard features (recursive functions, booleans, products, sums, *etc.*), fork-based concurrency, references with dynamic allocation, and operations for weak persistent memory. The expression $\mathbf{fork}\,\{e\}$ spawns a new thread that evaluates e in parallel with existing threads. We use the notation $e_1 \parallel e_2$ for the parallel execution of e_1 and e_2, which is derivable from **fork**. We



(a) Message passing (MP)

(b) Durable MP

(c) Flush and fence

(d) Optimized durable MP

Figure 4.2: Examples of programs that use weak and persistent memory operations

define **assert** to be function that is unsafe (gets stuck) if its argument is not True. The language features a weak persistent memory model. The full formal operational semantics appears in Section 4.3. In this section we give an intuitive explanation of the memory model illustrated by the examples in Figure 4.2. But first we fix some terminology.

A *consistency model* specifies the semantics of shared memory by restricting the *weak memory order*, the order of memory operations across threads. A concurrent program that correctly accounts for interleavings and the weak memory order is *thread-safe*. A *persistency model* specifies the semantics of persistent memory by restricting the *persist order*, the order in which writes may reach the persistent memory[PCW14]. A program using durable storage that correctly accounts for crashes and the persist order is *crash-safe*. The mentioned orders are defined using the *program order*, the order in which memory operations are issued by the program.

4.2.1 Release-Acquire and Non-Atomic Consistency

We use a highly relaxed consistency model closely resembling the release-acquire and non-atomic fragment of C11.[2] The memory operations for allocations ($\mathbf{ref}_a$), writes ($:=_a$), and reads ($!_a$) are annotated with a *memory access mode* $a \in \{\mathsf{na}, \mathsf{at}\}$. Allocations are considered a form of writes in the memory model. The access modes na and at are *non-atomic* and *atomic* access, respectively.

[2]The largest deviation from C11 is that we make no attempt to rule out data races on non-atomics which is undefined behavior in C11. This can be done with a *race-detector* [Dan+20; Kai+17]—we avoid that here for simplicity.

Non-atomic access is to be used when there are no races on data. For instance, when a thread uses a location exclusively or when synchronization has been established through other means, *e.g.,*through a lock or atomic operations (explained below). Non-atomic writes ($:=_{\mathsf{na}}$) performed by one thread give no guarantees on the order in which other threads may see them. This implies that it would be unsafe to use a non-atomic write to y in the example in Figure 4.2a. The right thread might read 1 from y without also reading 37 from x.

To ensure a desired weak memory order across threads, atomic access must be used. An atomic write ($:=_{\mathsf{at}}$) is called a *release-write* and an atomic read ($!_{\mathsf{at}}$) is called an *acquire-read*. If an acquire-read reads a value written by a release-write we say that the acquire-read *synchronizes* with the release-write. In this case, the write is ordered before the read in the weak memory order. Furthermore, a release-write is ordered after all preceding (in program order) memory operations, and an acquire-read is ordered before all succeeding (in program order) reads and writes. Together, this means that when a thread, call it t_1, performs an acquire-read and synchronizes with a release-write of another thread, say t_2, then t_1 becomes "aware of" (or acquires) all the writes that t_2 was aware of at the time of writing. This is exemplified by the message passing (MP) example in Figure 4.2a where the use of atomic operations make the assertion safe. When the sender thread writes 1 to y it is aware of the write of 37 to x (since it wrote it itself, program order). Hence, if the receiving thread reads 1 from y it also becomes aware of the write to x, thus the following read of x is certain to yield 37, and the assert will succeed.

The read-modify-write (RMW) operations **CAS** (compare-and-set) and **FAA** (fetch-and-add) count as both an acquire-read and a release-write at the same time.

4.2.2 Explicit Epoch Persistency

We use the examples in Figures 4.2b to 4.2d to explain the memory model we use. The notation $e \circlearrowleft e_r$ denotes execution of e with e_r configured as recovery code.[3] We use the *explicit epoch persistency* model by Izraelevitz et al. [IMS16]. As they argue this persistency model is a slight generalization of the x86 and ARM machine level persistency models. The model includes three operations to manage the persist order: an explicit flush, **flush** (also called a write-back), an asynchronous fence, **fence**, and a synchronous fence, $\textbf{fence}_{\textbf{sync}}$. In the absence of these instructions, *no guarantees* are given on the persist order. For instance, it is not safe to run the left-hand side of Figure 4.2a with the recovery code in Figure 4.2b. As there are no flushes or fences, the two writes might persist in any order: after a crash the recovery code might see y being 1 and x still being 0, even though, during normal execution, this would never be observable due to the release-write.

To enforce a certain persist order one must *explicitly* flush writes and then end an *epoch* with a fence. An asynchronous fence ensures that all writes that have been flushed before the fence persist prior to any writes after the fence. The asynchronous

[3] Note, that this is *not* syntax in the programming language.

fence does *not* ensure that the flushed writes have actually been persisted; hence, if a crash happens after the fence, the writes flushed prior to it might still be lost. But, when a certain persist order has been established, recovery code can perform a kind of "backwards reasoning". For instance, in Figure 4.2c the flush and fence implies that the write to x persists before the write to y. Hence, the recovery code can read y, and then, if the read yielded 1, reason backwards through the persist order and conclude that it is now certain to read 37 from x. This makes the assertion in Figure 4.2c safe. A synchronous fence, is stronger, but also potentially slower, than an asynchronous fence. It additionally *blocks execution* until all flushed writes have actually reached persistent memory. This means that had Figure 4.2c used a synchronous fence, then the write to x would have been persisted with certainty after executing the program.

Flushes and fences interact with release-writes and acquires-reads as a way to "connect" the weak memory order and the persist order. If an acquire-read synchronizes with a release-write then anything flushed *and* fenced prior to the release-write is guaranteed to persist before anything following a fence after the acquire-read. In the durable MP example in Figure 4.2b this ensures that the write to z in the right thread must persist after the write to x in the left thread and hence that the assertion made at recovery is safe. Note that the fence after the acquire-read of y is necessary. When performing an acquire-read a thread immediately gains knowledge of the writes the releasing thread know about. But, only after a fence does it gain knowledge about flushed and fenced writes known to the releasing thread. Note also that flushes without fences provide no ordering guarantees with respect to atomic operations.

The optimized durable MP example in Figure 4.2d is similar to the durable MP example except that the left thread does not flush the write to x before sending it through y. Hence, when the right threads read 1 from y it is still certain to know about the write to x (as in Figure 4.2a), but it no longer receives knowledge about the write being flushed. Hence, the right thread must flush x. With this being done it is still the case that the write to z persists after the write to x. But, it is no longer the case that the write to x will persist before the write to y. This brings us to the crucial point regarding this example: reading 1 from y carries with it different information to a concurrent thread (which gains knowledge that x holds 37) than it does to recovery code (which gains nothing). In Figure 4.2b it would also have been safe for the recovery code to read y instead of z, but here this would not be safe. At the logic level, this means that the resources associated with the write to y in Figure 4.2d must change at a crash, but it need not change in Figure 4.2b.

4.3 Operational Semantics

This section defines the full formal operational semantics of λ_{pmem}. The semantics formalizes the consistency and persistency models described informally in prior section.

$$m \in \text{MEvent} ::= \mathsf{Al}_a(\ell, v) \mid \mathsf{R}_a(\ell, v) \mid \mathsf{W}_a(\ell, v) \mid \mathsf{RMW}(\ell, v_r, v_w) \mid \mathsf{RMW}_{\mathsf{fail}}(\ell, v) \mid \mathsf{FL}(\ell) \mid \mathsf{F} \mid \mathsf{FS}$$

$$\begin{array}{rlrl}
\mathcal{V}, \mathcal{S}, \mathcal{F}, \mathcal{P}, \mathcal{B} \in \textsc{View} &\triangleq \textsc{Loc} \xrightarrow{\text{fin}} \mathbb{N} & \sigma \in \textsc{Store} &\triangleq \textsc{Loc} \xrightarrow{\text{fin}} \textsc{History} \\
\langle \mathcal{S}, \mathcal{F}, \mathcal{B} \rangle, \mathcal{T} \in \textsc{ThreadView} &\triangleq \textsc{View}^3 & h \in \textsc{History} &\triangleq \mathbb{N} \xrightarrow{\text{fin}} \textsc{Message} \\
\langle \sigma, \mathcal{P} \rangle, M \in \textsc{MemConf} &\triangleq \textsc{Store} \times \textsc{View} & \langle e, \mathcal{T} \rangle, t \in \textsc{ThreadState} &\triangleq \textsc{Expr} \times \textsc{ThreadView} \\
\langle v, \mathcal{S}_m, \mathcal{F}_m, \mathcal{P}_m \rangle \in \textsc{Message} &\triangleq \textsc{Val} \times \textsc{View}^3 & \langle M, \vec{t} \rangle, \rho \in \textsc{MemConf} &\times \textsc{List}(\textsc{ThreadState})
\end{array}$$

Figure 4.3: Definitions of semantic objects used in the operational semantics

Note: This section and Section 4.5 where included as appendices in Vindum and Birkedal [VB23c]. As such they are not essential to understand the rest of this chapter. Readers who wish a quick path to the Spirea logic can skip both sections and readers who want a deeper understanding are invited to read them. Both sections are required to understand Chapter 6.

Memory Events

To define how expressions interact with the memory we use two labeled transition systems (LTSs), one for expressions and one for the memory. This approach neatly keeps the memory model considerations separate from the rest of the language semantics. The labels for the LTSs are *memory events*, defined in Figure 4.3, describing how expressions can interact with the memory.

4.3.1 Expression LTS

The LTS for expressions has the form $e \xrightarrow{m} e'; \vec{e}$, meaning that the expression e can step to e' with the label $m \in \text{MEvent} \cup \{\epsilon\}$ while forking the sequence of threads $\vec{e}$. The label ϵ is used for expressions that do not interact with the memory (pure reductions, *etc.*). A selection of expression transitions is seen in Figure 4.4. First is the step for application (just to show that standard reductions work as expected), then the one for **fork** (the only rule where the sequence of forked threads is not the empty sequence ε), and then transitions that interact with the memory (*i.e.,* where $m \neq \epsilon$). Note how these transitions make it clear how the memory events correspond to operations in the language, for instance, the expression $\mathbf{ref}_a\ v$ emits an event for allocation of the form $\mathsf{Al}_a(\ell, v)$, reading a value with $!_a\ \ell$ emits an event for reading $\mathsf{R}_a(\ell, v)$, and so on.

4.3.2 Memory LTS

We now wish to define an LTS for the memory. To this end, we need some semantic objects, defined in Figure 4.3, which we now explain.

In a strong sequentially consistent memory threads always read the last write to a location, and hence the store (*i.e.,* the memory) can be modeled simply as a finite map from locations to values. In a weak persistent memory model, on the other hand, threads and recovery code may read out-of-date values. Therefore, the *store* is a finite map from locations to *histories*. Each history contains all writes to a location as a finite map from timestamps (natural numbers) to *messages*. Every message corresponds to a write to the location and contains the written value and other data explained below. The timestamps correspond to the order of the writes.

Closely related to the definition of the store is the notion of a *view*: a finite map from locations to timestamps. Views are used to represent subsets of messages in the store. For a store σ, a view $\mathcal{V}$ intuitively represents all messages of the form $\sigma(\ell)(t)$ for $t \leq \mathcal{V}(\ell)$. We sometimes talk of the messages "in" a view to mean this set of messages. Views naturally form a semi-lattice where the least element $\bot$ is the empty partial function, where $\mathcal{V}_1 \sqsubseteq \mathcal{V}_2 \triangleq \forall \ell \in \mathsf{dom}(\mathcal{V}_1).\, \mathcal{V}_1(\ell) \leq \mathcal{V}_2(\ell)$, and where the least upper bound is given by $(\mathcal{V}_1 \sqcup \mathcal{V}_2)(\ell) \triangleq \max(\mathcal{V}_1(\ell), \mathcal{V}_2(\ell))$.

A *memory configuration* (MemConf) contains the entire state of the memory. It is a pair of a store and a view: $\langle \sigma, \mathcal{P} \rangle$. We refer to $\mathcal{P}$ as the *persist view*; it represents the messages in σ that are certain to have been persisted.

A *thread view* is a triple of views: $\langle \mathcal{S}, \mathcal{F}, \mathcal{B} \rangle$. These views are a thread's *store view*, *flush view*, and *buffer view*. The store view $\mathcal{S}$ (flush view $\mathcal{F}$, respectively) is used to encode the weak memory order (persistent memory order). Messages in $\mathcal{S}$ are those that the thread knows of and that future memory operations will be ordered after. Messages in $\mathcal{F}$ are those that the thread knows have been flushed and fenced, meaning that will persist before any future memory operations by the thread. The buffer view $\mathcal{B}$ represents the messages that the thread has flushed.

A *message* is a tuple of the form: $\langle v, \mathcal{S}_m, \mathcal{F}_m, \mathcal{P}_m \rangle$. As mentioned, a message corresponds to a write and v is the value written. For an atomic write $\mathcal{S}_m$ and $\mathcal{F}_m$ is the writing thread's store view and flush view at the time of the write. An atomic read will acquire these views when reading the message. The $\mathcal{P}_m$ view enforces the persist order—a write can have persisted only if all messages in $\mathcal{P}_m$ have also persisted. In the operational semantics, the persist view of a message is only used in the reduction rule for a crash, corresponding to the fact that the persist order only affects crashes. For a message m, we write $m.v$, $m.\mathcal{S}$, *etc.* for its components.

The LTS for the memory has the form $\langle \sigma, \mathcal{P} \rangle; \langle \mathcal{S}, \mathcal{F}, \mathcal{B} \rangle \xrightarrow{m} \langle \sigma', \mathcal{P}' \rangle; \langle \mathcal{S}', \mathcal{F}', \mathcal{B}' \rangle$. As the outcome of a memory operation depends on the views of the thread making the operation, the LTS is parameterized both by a memory configuration and by a thread view: The transition rules appear in Figure 4.4; to keep the presentation concise we make use of the notation

$$\lfloor \mathcal{V} \rfloor_{\mathsf{at}} \triangleq \mathcal{V} \qquad \lfloor \mathcal{V} \rfloor_{\mathsf{na}} \triangleq \bot \qquad \mathcal{V}_0(\ell) = \begin{cases} \mathcal{V}(\ell) & \text{if } \ell \in \mathsf{dom}(\mathcal{V}) \\ 0 & \text{otherwise} \end{cases}$$

The $\lfloor \mathcal{V} \rfloor_a$ notation captures the effect of the access mode in several of the rules. For instance, the only difference between a write ($:=_{\mathsf{na}}$) and a release-write ($:=_{\mathsf{at}}$) is in

which views are stored in the written message. We use $\mathcal{V}_0$ simply to be able to write $\mathcal{V}_0(\ell)$ even if ℓ is not certain to be in the domain of $\mathcal{V}$.

We now comment on the transition rules. The rule for allocation (with label $\mathsf{Al}_a(\ell, v)$) extends the store with a fresh location that contains a history with a single message at timestamp 0. In this rule and in the rule for writing the store view and flush view of the message is $\bot$ if the access mode is na. This is because non-atomic operations are not for synchronization between threads and therefore no views should be exchanged when performing them. In contrast, when the access mode is at then the thread's store view and flush view are included in the message. For a read with access mode at, rule (with label $\mathsf{R}_a(\ell, v)$) then merges the store view from the read message into the thread's store view. This ensures that the acquire-read actually acquires information from the thread whose write it is reading. The flush view from the message is only added to the thread's buffer view. The buffer view is never transferred between threads, it is only used within a thread to keep account of information that will be acquired at the next fence. Indeed, the buffer view is moved into a thread's flush view by the two fence rules (with labels F and FS).

Due to the condition $\mathcal{S}_0(\ell) \leq t$ in rule for reading, a read may non-deterministically read any message for ℓ with a timestamp greater than the thread's timestamp in its store view for ℓ.

Note that the rule for writing ensures that the thread's flush view $\mathcal{F}$ is transferred irrespectively of access mode; this is to ensure that the persist order is recorded (it is used when we account for crashes, in Section 4.3.4).

The rules for flushes and fences are rather straightforward. Flushing a location moves the thread's timestamp for the location in its store view into its buffer view. The two rules for fences, move a thread's buffer view into its flush view. The rule for the synchronous fence additionally moves the buffer into the memory configuration's persist view as well.

4.3.3 Machine Reductions

We define a head reduction $\rightarrow_{\mathsf{h}}$ for a memory and a thread state (a pair of an expression and a thread view) by combining the LTSs for the memory and for expressions. It is given by the two rules seen in Figure 4.4. The first rule is for expression steps that interact with the memory and the second for those that do not.

As is standard we use evaluation contexts K to lift the head reduction to a per-thread reduction $\rightarrow_{\mathsf{t}}$ which again is lifted to a threadpool reduction $\rightarrow_{\mathsf{tp}}$ that non-deterministically picks a thread from the threadpool to reduce. They are each given by a single rule seen in Figure 4.4. Here K is an evaluation context; the definition of evaluation contexts for λ_{pmem} is entirely standard, capturing a call-by-value left-to-right evaluation order, and has thus been omitted.

Expression LTS

$$(\textbf{rec}\ f(x) = e)\ v \xrightarrow{\varepsilon} e[\textbf{rec}\ f(x) = e, v/f, x]; \varepsilon \qquad \textbf{fork}\ \{e\} \xrightarrow{\varepsilon} (); e$$

$$\textbf{ref}_a\ v \xrightarrow{\mathsf{Al}_a(\ell,v)} \ell; \varepsilon \qquad !_a\ \ell \xrightarrow{\mathsf{R}_a(\ell,v)} v; \varepsilon \qquad \ell :=_a v \xrightarrow{\mathsf{W}_a(\ell,v)} (); \varepsilon \qquad \textbf{flush}\ \ell \xrightarrow{\mathsf{FL}(\ell)} (); \varepsilon$$

$$\textbf{fence} \xrightarrow{\mathsf{F}} (); \varepsilon \qquad \textbf{fence}_{\textbf{sync}} \xrightarrow{\mathsf{FS}} (); \varepsilon \qquad \textbf{CAS}\ \ell\ v_1\ v_2 \xrightarrow{\mathsf{RMW}(\ell,v_1,v_2)} \mathsf{True}; \varepsilon$$

$$\frac{v_1 \neq v_l}{\textbf{CAS}\ \ell\ v_1\ v_2 \xrightarrow{\mathsf{RMW}_{\mathsf{fail}}(\ell,v_l)} \mathsf{False}; \varepsilon}$$

Memory Model LTS

$$\frac{\ell \notin \mathsf{dom}(\sigma) \qquad h = \{0 \mapsto \langle v, \lfloor \mathcal{S} \rfloor_a, \lfloor \mathcal{F} \rfloor_a, \mathcal{F} \rangle\}}{\langle \sigma, \mathcal{P} \rangle; \langle \mathcal{S}, \mathcal{F}, \mathcal{B} \rangle \xrightarrow{\mathsf{Al}_a(\ell,v)} \langle \sigma[\ell \mapsto h], \mathcal{P} \rangle; \langle \mathcal{S}, \mathcal{F}, \mathcal{B} \rangle}$$

$$\frac{t = \mathcal{S}_0(\ell) \qquad \mathcal{B}' = \mathcal{B}[\ell \mapsto t]}{\langle \sigma, \mathcal{P} \rangle; \langle \mathcal{S}, \mathcal{F}, \mathcal{B} \rangle \xrightarrow{\mathsf{FL}(\ell)} \langle \sigma, \mathcal{P} \rangle; \langle \mathcal{S}, \mathcal{F}, \mathcal{B}' \rangle}$$

$$\frac{\mathcal{S}_0(\ell) \leq t \qquad \sigma(\ell)(t) = \langle v, \mathcal{S}_m, \mathcal{F}_m, \_ \rangle}{\langle \sigma, \mathcal{P} \rangle; \langle \mathcal{S}, \mathcal{F}, \mathcal{B} \rangle \xrightarrow{\mathsf{R}_a(\ell,v)} \langle \sigma, \mathcal{P} \rangle; \langle \mathcal{S} \sqcup \lfloor \mathcal{S}_m \rfloor_a, \mathcal{F}, \mathcal{B} \sqcup \lfloor \mathcal{F}_m \rfloor_a \rangle}$$

$$\frac{\mathcal{S}_0(\ell) < t \qquad \sigma(\ell) = h \qquad t \notin \mathsf{dom}(h) \qquad h' = h[t \mapsto \langle v, \lfloor \mathcal{S} \rfloor_a, \lfloor \mathcal{F} \rfloor_a, \mathcal{F} \rangle]}{\langle \sigma, \mathcal{P} \rangle; \langle \mathcal{S}, \mathcal{F}, \mathcal{B} \rangle \xrightarrow{\mathsf{W}_a(\ell,v)} \langle \sigma[\ell \mapsto h'], \mathcal{P} \rangle; \langle \mathcal{S}[\ell \mapsto t], \mathcal{F}, \mathcal{B} \rangle}$$

$$\frac{\begin{array}{c}\mathcal{S}_0(\ell) \leq t \qquad t+1 \notin \mathsf{dom}(h) \qquad \sigma(\ell) = h \qquad h(t) = \langle v_m, \mathcal{S}_m, \mathcal{F}_m, \_ \rangle \\ \mathcal{S}' = (\mathcal{S} \sqcup \mathcal{S}_m)[\ell \mapsto t+1] \qquad h' = h[t+1 \mapsto \langle v, \mathcal{S}', \mathcal{F} \sqcup \mathcal{F}_m, \mathcal{F} \sqcup \mathcal{F}_m \rangle]\end{array}}{\langle \sigma, \mathcal{P} \rangle; \langle \mathcal{S}, \mathcal{F}, \mathcal{B} \rangle \xrightarrow{\mathsf{RMW}(\ell,v,v')} \langle \sigma[\ell \mapsto h'], \mathcal{P} \rangle; \langle \mathcal{S}', \mathcal{F}, \mathcal{B} \sqcup \mathcal{F}_m \rangle}$$

$$\frac{\mathcal{S}_0(\ell) \leq t \qquad t+1 \notin \mathsf{dom}(h) \qquad \sigma(\ell) = h \qquad h(t) = \langle v_m, \mathcal{S}_m, \mathcal{F}_m, \_ \rangle}{\langle \sigma, \mathcal{P} \rangle; \langle \mathcal{S}, \mathcal{F}, \mathcal{B} \rangle \xrightarrow{\mathsf{RMW}_{\mathsf{fail}}(\ell,v)} \langle \sigma, \mathcal{P} \rangle; \langle \mathcal{S} \sqcup \mathcal{S}_m, \mathcal{F}, \mathcal{B} \sqcup \mathcal{F}_m \rangle}$$

$$\langle \sigma, \mathcal{P} \rangle; \langle \mathcal{S}, \mathcal{F}, \mathcal{B} \rangle \xrightarrow{\mathsf{F}} \langle \sigma, \mathcal{P} \rangle; \langle \mathcal{S}, \mathcal{F} \sqcup \mathcal{B}, \mathcal{B} \rangle$$

$$\langle \sigma, \mathcal{P} \rangle; \langle \mathcal{S}, \mathcal{F}, \mathcal{B} \rangle \xrightarrow{\mathsf{FS}} \langle \sigma; \mathcal{P} \sqcup \mathcal{B} \rangle; \langle \mathcal{S}, \mathcal{F} \sqcup \mathcal{B}, \mathcal{B} \rangle$$

Head Reduction

$$\frac{M; \mathcal{T} \xrightarrow{m} M'; \mathcal{T}' \qquad e \xrightarrow{m} e'; e_1 \ldots e_n}{M; \langle e, \mathcal{T} \rangle \rightarrow_{\mathsf{h}} M'; \langle e', \mathcal{T}' \rangle; \langle e_1, \mathcal{T}' \rangle \ldots \langle e_n, \mathcal{T}' \rangle}$$

$$\frac{e \xrightarrow{\varepsilon} e'; e_1 \ldots e_n}{M; \langle e, \mathcal{T} \rangle \rightarrow_{\mathsf{h}} M; \langle e', \mathcal{T} \rangle; \langle e_1, \mathcal{T} \rangle \ldots \langle e_n, \mathcal{T} \rangle}$$

Thread-local and threadpool reduction

$$\frac{M; \langle e, \mathcal{T} \rangle \rightarrow_{\mathsf{h}} M'; \langle e', \mathcal{T}' \rangle; \vec{t}}{M; \langle K[e], \mathcal{T} \rangle \rightarrow_{\mathsf{t}} M'; \langle K[e'], \mathcal{T}' \rangle; \vec{t}} \qquad \frac{M; t \rightarrow_{\mathsf{t}} M'; t'; \vec{t}}{\langle M; \vec{t_l} t \vec{t_r} \rangle \rightarrow_{\mathsf{tp}} \langle M'; \vec{t_l} t' \vec{t_r} \vec{t} \rangle}$$

Figure 4.4: Expression and Memory Model LTS transitions

4.3.4 Accounting for Crashes

The state of the memory after a crash is determined by a *crash view* $\mathcal{C}$. The crash view represents all the messages that persisted before the crash. It has to be a *consistent cut* in the sense that no message can have persisted without all of the messages in its persist view also having persisted:

Definition 4.3.1. *A view $\mathcal{C}$ is a* consistent cut *of a store σ, written consistent$(\sigma, \mathcal{C})$, iff for every $\mathcal{C}(\ell) = t$ there exists a history h such that $\sigma(\ell) = h$ and $t \in \mathsf{dom}(h)$. Furthermore, for all $t' \in \mathsf{dom}(h)$ where $t' \leq t$ there exists a message m such that $h(t') = m$ and $m.\mathcal{P} \sqsubseteq \mathcal{C}$.*

With this in hand, we can define the *crash step* reduction $\xrightarrow{\lightning}$. This reduction goes from memory configurations to memory configurations and describes what can happen to the memory at a crash. It is generated by a single rule:

M-CRASH

$$\frac{\begin{array}{cc}\mathcal{P} \sqsubseteq \mathcal{C} & \mathsf{consistent}(\sigma, \mathcal{C}) \\ \mathsf{dom}(\sigma') = \mathsf{dom}(\mathcal{C}) & \forall \ell \in \mathsf{dom}(\mathcal{C}).\, \sigma'(\ell) = \{0 \mapsto \langle \sigma(\ell)(\mathcal{C}(\ell)).v, \bot, \bot, \bot\rangle\}\end{array}}{\langle \sigma, \mathcal{P}\rangle \xrightarrow{\lightning} \langle \sigma', \mathrm{viewToZero}(\mathcal{C})\rangle}$$

The first two assumptions ensure that $\mathcal{C}$ is a consistent cut and that it includes all the definitely persisted messages in $\mathcal{P}$. The next line serves to constrict the new store σ', created by picking out a message for each recovered location. The new persist view $\mathrm{viewToZero}(\mathcal{C})$ is the view such that $\mathrm{viewToZero}(\mathcal{C})(\ell) = 0$ for all $\ell \in \mathsf{dom}(\mathcal{C})$ and which is undefined for all $\ell \notin \mathsf{dom}(\mathcal{C})$. That is, the crash view but with 0 at every entry.

Finally, we define a *recoverable execution relation* $\Rightarrow_{\mathsf{r}}$, which expresses what it means to execute a program together with a recovery program e_r that is run after each crash. The relation has the form $e_r; \rho \Rightarrow_{\mathsf{r}} \rho'; s$ where e_r is the recovery expression, ρ and ρ' are machine configurations, and $s \in \{\mathsf{NotCrashed}, \mathsf{Crashed}\}$ is a crash-status indicating whether the execution has been crash free or has crashed along the way.

REXEC-NORMAL

$$\frac{\rho \rightarrow^*_{\mathsf{tp}} \rho'}{e_r; \rho \Rightarrow_{\mathsf{r}} \rho'; \mathsf{NoCrash}}$$

REXEC-CRASHED

$$\frac{\rho \rightarrow^*_{\mathsf{tp}} \langle M, \vec{t}\rangle \qquad M \xrightarrow{\lightning} M' \qquad e_r; \langle M', [\langle e_r, \langle \bot, \bot, \bot\rangle\rangle]\rangle \Rightarrow_{\mathsf{r}} \rho'; s}{e_r; \rho \Rightarrow_{\mathsf{r}} \rho'; \mathsf{Crashed}}$$

Note that in contrast to the other relations we have defined above this is a *big step* relation. The first rule says that a machine can execute (without crashes) per the threadpool reduction with the label NoCrash. The second rule says that a machine may execute normally for some number of steps (the first assumption), then let the

$$\textsc{Htc-atomic}\quad \frac{\text{atomic}(e) \qquad P \twoheadrightarrow Q_c}{\{P\}\, e\, \{Q \wedge Q_c\} \vdash \{P\}\, e\, \{Q\}\{Q_c\}}$$

$$\textsc{Htr-idempotence}\quad \frac{\{P\}\, e\, \{Q\}\{Q_r\} \qquad Q_r \twoheadrightarrow \langle \mathsf{PC} \rangle\, R \qquad \{R\}\, e_r\, \{Q_r\}\{Q_r\}}{\{P\}\, e \circlearrowleft e_r\, \{Q\}\{Q_r\}}$$

Figure 4.5: Key rules for quadruples in Perennial

memory take a crash step (the second assumption), and then keep executing with the memory after the crash and a single thread executing the recovery expression. Note that at a crash all the running threads $\vec{t}$ are discarded.

4.4 Background: Crash Reasoning Features In Perennial

Perennial extends Hoare logic with a *crash Hoare quadruple* of the form $\{P\}\, e\, \{Q\}\{Q_c\}$. Here P and Q are standard pre- and postconditions. The fourth component Q_c is a *crash condition* that must hold during every step of execution of e. Since Q_c holds at every step, if a crash occurs at some point, then Q_c will necessarily hold at that point. Hence, the crash-condition is a property that recovery code can rely on after a crash.

In addition to standard language independent structural rules (a frame rule, a bind rule, *etc.*), the key rule for deriving a crash Hoare quadruple is Htc-atomic seen in Figure 4.5. The rule states that to prove a crash Hoare quadruple for an *atomic* expression e, it suffices to prove that the pre-condition implies Q_c and an ordinary Hoare triple for e with Q_c added to the postcondition. Since e is atomic and can take only a single step, it suffices to show the crash condition before and after this single step. Note the use of the standard (non-separating) conjunction $\wedge$. This makes it possible to use all the resources one has at hand to show both Q and Q_c. This is a crucial aspect of crash conditions: they can be established without losing the resources necessary to show them.[4] The use of $\wedge$ is sound since, when the program runs, it will *either* take a normal step of execution (in which case the proof of Q is needed) *or* crash (in which case the proof of Q_c is needed). Since both cannot happen at the same time, it is not necessary to show the two conjuncts for disjoint resources. The Htc-atomic rule is important since it, in combination with the structural rules, allows us to show a crash Hoare quadruple by showing a normal Hoare triple at each step. This explains why we show rules for normal Hoare triples later in Section 4.6.

To show crash-safety Perennial offers *recovery Hoare quadruples* of the form $\{P\}\, e \circlearrowleft e_r\, \{Q\}\{Q_r\}$. The intuitive reading is: given that P holds initially, it is

[4] This is in contrast to normal Iris invariants, where one has to sacrifice ownership of the resources necessary to show the invariant.

safe to execute e with the recovery program e_r. If e terminates in a value v without crashing then $Q(v)$ holds. If, on the other hand, one or more crashes occur during execution (of e and e_r) and e_r terminates in a value v, then $Q_r(v)$ holds.

Per the idempotence rule Htr-idempotence one can show a recovery Hoare quadruple for a program e and recovery program e_r by showing a crash Hoare quadruple for e and one for e_r. In both cases the crash condition is Q_r, such that e_r can rely on this resource; not directly though, as the crash itself might change Q_r, hence the inclusion of the post-crash modality. Since e_r itself maintains the crash condition Q_r, any number of crashes during e_r are still safe.

In summary, the proof burden for proving crash-safety is to pick a crash condition and apply Htr-idempotence. Then one verifies two crash Hoare quadruples. The verification of these is similar to using normal Hoare triples except that the crash condition must be shown at every step.[5]

4.5 BaseSpirea – The Low-Level Logic

In this section we present BaseSpirea, a program logic for λ_{pmem}. The logic is built on top of Iris and Perennial by instantiating the Perennial program logic framework. Before we proceed, we briefly explain the relationship between Iris and Perennial and what such an instantiation entails.

4.5.1 Instantiating Perennial

Iris includes both a base logic and a program logic framework. The base logic contains the fundamental features of Iris, such as the separation logic connectives and ghost state, but not program verification capabilities. Perennial, in turn, builds a program logic framework on top of the Iris base logic. Unlike the program logic in Iris, Perennial's is able to reason about crashes and to verify crash-safety, through a combination of powerful features made for this purpose. Here "framework" describes the fact that one can instantiate the program logic with any suitable programming language. We instantiate the framework with λ_{pmem}, which, by no coincidence, is one such suitable language. By doing this instantiation one gets the basic building blocks of a program logic "for free". These building blocks include the definition of Hoare triples (and related notions explained in the next section) and language independent structural rules for working with them. The instantiator (*i.e.,* us) then defines language specific assertions and proof rules that must be proven sound. This is done in tandem with a so-called *state interpretation* that is picked as part of the instantiation. The state interpretation's purpose is to link the physical state (*i.e.,* the state in the operational semantics) with logical ghost state. At this level, our state interpretation is fairly standard and we do not give the details here—the interested reader can find them in our Coq mechanization. One noteworthy aspect of our state interpretation is that it is parameterized over an "extra" resource. In BaseSpirea this

[5] Showing the crash condition is usually trivial and can be automated with a Coq tactic.

extra resource is simply **true**, but when building Spirea on top of BaseSpirea, we use it to inject additional resources into the state interpretation.

We next describe the most important features of the Perennial program logic that we inherit (Section 4.5.2); the language specific assertions BaseSpirea adds (Section 4.5.3); some of the program rules that BaseSpirea includes (Section 4.5.4); and finally we give the adequacy result of the logic (Section 4.5.5).

4.5.2 The Perennial Program Logic

In this section we explain the most important features from Perennial that are also present in BaseSpirea. Note that in this subsection we use e to denote expressions from the point of view of Perennial. When instantiated with λ_{pmem}, such an expression is in fact a thread state (defined in Section 4.3).

In addition to the well-known Hoare triple, Perennial includes a *crash Hoare triple*[6] of the form $\{P\}\, e\, \{Q\}\{Q_c\}$. Here P and Q are standard pre- and postconditions and the fourth component Q_c is a *crash condition* that must hold during every step of execution of e. Since Q_c holds at every step, if a crash occurs at some point, then Q_c will necessarily hold at that point. Hence, the crash-condition is a property that recovery code can rely on after a crash.

In addition to standard language independent structural rules (a frame rule, a bind rule, *etc.*), the key rule for deriving a crash Hoare triple is HTC-ATOMIC seen in Figure 4.7

The rule states that to prove a crash Hoare triple for an atomic expression e, it suffices to prove that the pre-condition implies Q_c and an ordinary Hoare triple for e holds with Q_c added to the postcondition. Since e is atomic and can take only a single step, it suffices to show the crash condition before and after this single step. Note the use of the standard (non-separating) conjunction $\wedge$. This makes it possible to use all the resources one has at hand to show both Q and Q_c. This is a crucial aspect of crash conditions: they can be established without losing the resources necessary to show them.[7] The use of $\wedge$ is sound since when the program runs it will *either* take a normal step of execution (in which case the proof of Q is needed) *or* crash (in which case the proof of Q_c is needed). Since both cannot happen at the same time, it is not necessary to show the two conjuncts for disjoint resources. The HTC-ATOMIC rule is important since it, in combination with the structural rules, allows us to show a crash Hoare triple by showing a normal Hoare triple. This explains why we show rules for normal Hoare triples later on in this section.

To reason about the combination of a program e and its associated recovery program e_r, Perennial offers a *recovery Hoare triple*[8] of the form $\{P\}\, e \circlearrowleft e_r\, \{Q\}\{Q_r\}$. The intuitive reading is: given that P holds initially, it is safe to execute e with the

[6] Really, it is a quadruple, but we stick with the Hoare triple terminology

[7] This is in contrast to normal Iris invariants, where one has to sacrifice ownership of the resources necessary to show the invariant.

[8] Again, we stick with the Hoare triple terminology, even if more than three components are involved.

Assertion	Description
$\ell \hookrightarrow_{\mathsf{h}} h$	Location ℓ points to the history h.
valid($\mathcal{S}$)	View $\mathcal{S}$ is valid.
persisted($\mathcal{P}$)	View $\mathcal{P}$ has been persisted.
$\mathrm{crashedAt}(\mathcal{C})$	View $\mathcal{C}$ was recovered at the last crash.
$\langle \mathsf{PC} \rangle\, P$	Proposition P holds after a crash.

Figure 4.6: Overview over assertions in BaseSpirea

recovery program e_r. If e terminates in a value v without crashing then $Q(v)$ holds. If, on the other hand, one or more crashes occur during execution (of e and e_r) then, if e_r terminates in a value v, then $Q_r(v)$ holds. Beware that the Q_r plays a different role from the Q_c used in a crash Hoare triple.

A *post-crash modality* $\langle \mathsf{PC} \rangle$ internalizes in the logic how resources are affected by a crash. The assertion $\langle \mathsf{PC} \rangle\, P$ means that P holds after a crash and a rule of the form $P \vdash \langle \mathsf{PC} \rangle\, Q$ means that if one has P then one has Q after a crash.

Since the post-crash modality depends intrinsically on the semantics of the specific programming language one reasons about, we do not get the post-crash modality by instantiating Perennial; we have to define it ourselves. The *idea* of a post-crash modality, however, is from Perennial. That being said, ours is more complicated than earlier ones used with Perennial due to the more intricate semantics for crashes in λ_{pmem} (for more details, see the discussion of related work in Section 4.9).

Per the idempotence rule HTR-IDEMPOTENCE one can show a recovery Hoare triple for a program e and recovery program e_r by showing a crash Hoare triple for e and one for e_r. In both cases the crash condition is Q_r, such that e_r can rely on this resource; not directly though, as the crash itself might change Q_r, hence the inclusion of the post-crash modality. Since e_r itself maintains the crash condition Q_r, any number of crashes during e_r are still safe.

The Perennial program logic contains other features. For instance, *crash borrows* that make it possible to transfer and split crash conditions between threads. But what we have explained thus far suffices for this paper, so we proceed to explain the assertions specific to BaseSpirea.

4.5.3 Assertions in BaseSpirea

Figure 4.6 provides an overview over the assertions in BaseSpirea.

Since the store in our operational semantics contains not just single values, but entire histories, the points-to predicate in BaseSpirea $\ell \hookrightarrow_{\mathsf{h}} h$ naturally associates a location with a history h. Except for this, it is similar to the normal separation logic points-to predicate.

The assertion valid($\mathcal{S}$) states that a view $\mathcal{S}$ is valid. This means that if $\mathcal{S}(\ell) = t$ then the history for ℓ in the physical store actually contains a message with at least

HTC-ATOMIC
$$\frac{\mathsf{atomic}(e) \qquad P \twoheadrightarrow Q_c}{\{P\}\, e\, \{Q \wedge Q_c\} \vdash \{P\}\, e\, \{Q\}\{Q_c\}}$$

HTR-IDEMPOTENCE
$$\frac{\{P\}\, e\, \{Q\}\{Q_r\} \qquad Q_r \twoheadrightarrow \langle \mathsf{PC} \rangle\, R \qquad \{R\}\, e_r\, \{Q_r\}\{Q_r\}}{\{P\}\, e \circlearrowleft e_r\, \{Q\}\{Q_r\}\,.}$$

CRASHED-AT-AGREE
$$\mathrm{crashedAt}(\mathcal{C}) * \mathrm{crashedAt}(\mathcal{C}') \vdash \mathcal{C} = \mathcal{C}'$$

PERSISTED-SEP
$$\mathrm{persisted}\,\mathcal{P}_1 * \mathrm{persisted}\,\mathcal{P}_2 \dashv\vdash \mathrm{persisted}(\mathcal{P}_1 \sqcup \mathcal{P}_2)$$

PC-PERSISTED
$$\mathrm{persisted}(\mathcal{P}) \vdash \langle \mathsf{PC} \rangle\, \mathrm{persisted}(\mathrm{viewToZero}(\mathcal{P})) * \exists \mathcal{C} \sqsupseteq \mathcal{P}.\ \mathrm{crashedAt}(\mathcal{C})$$

PC-POINTS-TO
$$\ell \hookrightarrow_{\mathsf{h}} h \vdash \langle \mathsf{PC} \rangle\, \exists \mathcal{C}.\, \mathrm{crashedAt}(\mathcal{C}) * \left(\ell \notin \mathsf{dom}(\mathcal{C}) \vee \left(\exists t, m. \begin{array}{l} h(t) = m * \mathcal{C}(\ell) = t * m.\mathcal{P} \sqsubseteq \mathcal{C} * \\ \ell \hookrightarrow_h \{0 \mapsto \langle m.v, \bot, \bot, \bot \rangle\} \end{array} \right) \right)$$

Figure 4.7: Selected rules for assertions in BaseSpirea

the timestamp t. Knowing this is necessary to conclude, for instance, that performing a read with the store view $\mathcal{S}$ is safe.

The assertion $\mathrm{persisted}(\mathcal{P})$ means that the view $\mathcal{P}$ is included in the persist view in the physical state. It does not entail any ownership (in the separation logic sense) of the physical persist view. Since it only expresses a lower bound and since the physical persist view only grows during normal execution it is persistent.[9]

The assertion $\mathrm{crashedAt}(\mathcal{C})$ means that at *the last crash* the consistent cut that the machine crashed at was $\mathcal{C}$. The assertion $\mathrm{crashedAt}(\mathcal{C})$ is persistent and has agreement (CRASHED-AT-AGREE).

We now need to describe how the assertions interact with the post-crash modality. There are no rules for valid or $\mathrm{crashedAt}$. These resources do not imply any non-trivial resources after a crash, *i.e.,* the assertions $\mathrm{valid}(\mathcal{S})$ and $\mathrm{crashedAt}(\mathcal{C})$ are lost under the post-crash modality.

For $\mathrm{persisted}(\mathcal{P})$ we have the rule PC-PERSISTED. It says that given $\mathrm{persisted}(\mathcal{P})$, then after a crash $\mathrm{persisted}$ holds for the same view but with zero at every entry. Furthermore, there exists some view $\mathcal{C}$ such that $\mathcal{C} \sqsupseteq \mathcal{P}$ and $\mathrm{crashedAt}(\mathcal{C})$ holds. This rule is sound because $\mathrm{persisted}(\mathcal{P})$ is a lower bound on the persist view in the physical state and a crash view has to include the persist view (recall the rule M-CRASH in the operational semantics).

The rule PC-POINTS-TO for the points-to predicate is a bit more involved. After a crash, we again have $\mathrm{crashedAt}(\mathcal{C})$ for some $\mathcal{C}$ and two distinct cases: the location was either lost or recovered at the crash. In the first case, the location must not be present in the crash view, $\ell \notin \mathsf{dom}(\mathcal{C})$, and we have, of course, lost the points-to predicate. In the latter case, $\mathcal{C}(\ell) = t$ for some t, the message $h(t)$ was recovered, and we now have a points-to predicate for the recovered message. Furthermore, the view $\mathcal{P}$ in the message must be included in $\mathcal{C}$. This internalizes the fact that $\mathcal{C}$

[9] Not to be confused with persistent memory, in Iris a persistent proposition is one that does not entail exclusive ownership but only represents duplicable knowledge. $\Box P$ means that P always holds and a proposition P is persistent if $P \vdash \Box P$.

HT-ALLOC
$\{\mathrm{valid}(\mathcal{S})\}\ \mathbf{ref}_a\ v; \langle \mathcal{S}, \mathcal{F}, \mathcal{B}\rangle\ \{\ell; \langle \mathcal{S}', \mathcal{F}', \mathcal{B}'\rangle.\ \ell \hookrightarrow_{\mathsf{h}} \{0 \mapsto \langle v, \lfloor \mathcal{S}\rfloor_a, \lfloor \mathcal{F}\rfloor_a, \mathcal{F}\rangle\} * \mathcal{S} = \mathcal{S}' * \mathcal{F} = \mathcal{F}' * \mathcal{B} = \mathcal{B}'\}$

HT-READ
$$\left\{\begin{array}{l}\mathrm{valid}(\mathcal{S}) * \\ \ell \hookrightarrow_{\mathsf{h}} h\end{array}\right\}\ !_a\,\ell; \langle \mathcal{S}, \mathcal{F}, \mathcal{B}\rangle\ \left\{\begin{array}{l} v; \langle \mathcal{S}', \mathcal{F}', \mathcal{B}'\rangle.\ \exists t.\, \mathcal{S}_0(\ell) \le t * h(t) = \langle v_m, \mathcal{S}_m, \mathcal{P}_m, \_\rangle * v = v_m * \\ \quad \mathcal{S}' = \mathcal{S} \sqcup \lfloor \mathcal{S}_m\rfloor_a * \mathcal{F}' = \mathcal{F} * \mathcal{B} = \mathcal{B}' \sqcup \lfloor \mathcal{P}_m\rfloor_a * \mathrm{valid}(\mathcal{S}') * \ell \hookrightarrow_{\mathsf{h}} h \end{array}\right\}$$

HT-STORE
$$\left\{\begin{array}{l}\mathrm{valid}(\mathcal{S}) * \\ \ell \hookrightarrow_{\mathsf{h}} h\end{array}\right\}\ \ell :=_a v; \langle \mathcal{S}, \mathcal{F}, \mathcal{B}\rangle\ \left\{\begin{array}{l} w; \langle \mathcal{S}', \mathcal{F}', \mathcal{B}'\rangle.\ \exists t.\, \mathcal{S}_0(\ell) < t * t \notin \mathrm{dom}(h) * w = () * \mathcal{S}' = \mathcal{S}[\ell \mapsto t] * \\ \quad \mathcal{F}' = \mathcal{F} * \mathcal{B}' = \mathcal{B} * \mathrm{valid}(\mathcal{S}') * \ell \hookrightarrow_{\mathsf{h}} h[t \mapsto \langle v, \lfloor \mathcal{S}'\rfloor_a, \lfloor \mathcal{F}\rfloor_a, \mathcal{F}\rangle] \end{array}\right\}$$

HT-FLUSH
$\{\ell \hookrightarrow_{\mathsf{h}} h\}\ \mathbf{flush}\ \ell; \langle \mathcal{S}, \mathcal{F}, \mathcal{B}\rangle\ \{w; \langle \mathcal{S}', \mathcal{F}', \mathcal{B}'\rangle.\ \mathcal{S}' = \mathcal{S} * \mathcal{F}' = \mathcal{F} * \mathcal{B}' = \mathcal{B}[\ell \mapsto \mathcal{S}_0(t)] * \ell \hookrightarrow_{\mathsf{h}} h\}$

HT-FENCE
$\{\mathbf{true}\}\ \mathbf{fence}; \langle \mathcal{S}, \mathcal{F}, \mathcal{B}\rangle\ \{w; \langle \mathcal{S}', \mathcal{F}', \mathcal{B}'\rangle.\ \mathcal{S}' = \mathcal{S} * \mathcal{F}' = \mathcal{F} \sqcup \mathcal{B} * \mathcal{B}' = \mathcal{B}\}$

HT-FENCE-SYNC
$\{\mathbf{true}\}\ \mathbf{fence}_{\mathbf{sync}}; \langle \mathcal{S}, \mathcal{F}, \mathcal{B}\rangle\ \{w; \langle \mathcal{S}', \mathcal{F}', \mathcal{B}'\rangle.\ \mathcal{S}' = \mathcal{S} * \mathcal{F}' = \mathcal{F} \sqcup \mathcal{B} * \mathcal{B}' = \mathcal{B} * \mathrm{persisted}(\mathcal{B})\}$

Figure 4.8: Selected rules for Hoare triples in BaseSpirea

must be a consistent cut and hence respect $\mathcal{P}$ in the message. This is what makes it possible to do the kind of "backwards reasoning" for recovery code that we discussed earlier.

Let us see how the post-crash rules for $\mathrm{persisted}(\mathcal{P})$ and $\ell \hookrightarrow_{\mathsf{h}} h$ work together if we have both before a crash. Since $\mathrm{crashedAt}$ has agreement (CRASHED-AT-AGREE) the two crash views gained by PC-PERSISTED and PC-POINTS-TO must be equal. Hence, if one knows that $\ell \in \mathrm{dom}(\mathcal{P})$ then one can rule out the first case in the disjunction in PC-POINTS-TO and obtain a points-to predicate after the crash.

4.5.4 BaseSpirea Program Logic

BaseSpirea includes proof rules for all programming language constructs of λ_{pmem}. The rules for the memory-related operations are shown in Figure 4.8; we only include these rules as the proof rules for the remaining part of λ_{pmem} are as in standard Iris, see, *e.g.*,[Jun+18a] (but, we hasten to point out, we have also proven those standard rules sound!).

The memory-related rules very closely reflect the underlying operational semantics and thus we do not explain them in great detail, but only make a few general observations. Since the state of a thread in our operational semantics is described not only by an expression but also by a thread view, the "program" in our Hoare-triple is a thread state and not just an expression. All the rules that involve reading and writing to a location include $\mathrm{valid}(\mathcal{S})$ in their precondition and $\mathrm{valid}(\mathcal{S}')$ in their postcondition. The knowledge of validity is necessary to conclude that reads do not get stuck, as mentioned above.

The rules for flushing and fences are very simple. The rule HT-FLUSH requires a points-to predicate only to ensure that the flushed location actually exists in the store. The only difference between HT-FENCE and HT-FENCE-SYNC is that the later includes $\mathrm{persisted}(\mathcal{B})$ in the postcondition. The rule for a synchronous fence is the only way to get the $\mathrm{persisted}(P)$ assertion.

4.5.5 Soundness

The soundness theorem for BaseSpirea states that a recovery Hoare triple for a program proven inside the logic implies a safety result about the program with respect to the operational semantics—independently of the logic. Since this result is the same in the soundness theorem for both BaseSpirea and Spirea we define it separately, such that we can reuse it in both theorems.

Definition 4.5.1. *For expressions e and e_r, memory configuration M, and meta-level predicates on values Φ and Φ_r, we say that* $\mathrm{safe}(e, e_r, M, \Phi, \Phi_r)$ *holds if, for any recoverable execution*

$$e_r; \langle M, [\langle e, \langle \bot, \bot, \bot \rangle \rangle] \rangle \Rightarrow_r \langle M, \vec{t} \rangle; s$$

it is the case that: (1) For every thread $\langle e, \mathcal{T} \rangle \in \vec{t}$, if e is not a value then the thread is not stuck. (2) For $\langle e', \mathcal{T} \rangle = (\vec{t})_1$, if e' is a value v (i.e., the initial expression e terminated) then $\Phi(v)$ holds if $s =$ NotCrashed and $\Phi_r(v)$ holds if $s =$ Crashed.

With this safety definition we state the soundness theorem.

Theorem 4.5.2 (soundness)**.** *Let e, e_r, $\langle \sigma, \mathcal{P} \rangle$, Φ, and Φ_r be as in Definition 4.5.1. If the following recovery Hoare triple is provable in BaseSpirea*

$$\left\{ \mathrm{valid}(\mathcal{P}) * \mathop{\Large *}_{\ell \in \mathsf{dom}(\sigma)} \ell \hookrightarrow_h \sigma(\ell) \right\} \langle e, \mathcal{P} \rangle \circlearrowleft \langle e_r, \bot, \bot, \bot \rangle \, \{\Phi\}\{\Phi_r\}$$

then $\mathrm{safe}(e, e_r, \langle \sigma, \mathcal{P} \rangle, \Phi, \Phi_r)$ *holds.*

Note that the theorem applies to a store and persist view that is not necessarily empty to begin with. Importantly, this makes it possible to apply it to programs that assume already existing and persisted locations.

4.6 Spirea

Spirea is a CSL based on the Iris separation logic framework. As such it contains all the standard connectives from Iris-based separation logics such as the separating conjunction, ghost state, higher-order quantifiers, *etc.* For reasoning about programs it offers Hoare triples, recovery Hoare quadruples, and crash Hoare quadruples. The latter two support the same rules as they do in Perennial. In this section we explain the novel aspects of Spirea. Throughout the section we cover the verification of the two examples from Figure 4.2a and Figure 4.2c; proof outlines are shown in Figure 4.13 and Figure 4.14. Figure 4.9 provide an overview of the assertions that are introduced over the course of this section.

Assertion	Description
$\langle \mathsf{obj} \rangle\, P$	P holds independently of any views.
$\langle \mathsf{NF} \rangle\, P$	P holds without making any assertions on what is flushed and what is in the buffer.
$\langle \mathsf{NB} \rangle\, P$	P holds without making any assertions on the buffer.
$\boxed{\ell}\boxed{\pi}$	The location ℓ is associated with the invariant π.
$\ell \hookrightarrow_{\mathsf{na}} \vec{\sigma}$	The states $\vec{\sigma}$ have been written to ℓ in the given order and only the states in $\vec{\sigma}$ are possible after a crash.
$\ell \hookrightarrow_{\mathsf{at}} \vec{\sigma}$	The states $\vec{\sigma}$ have been written to ℓ in the given order.
$\ell \succsim_{\mathsf{p}} \sigma$	ℓ is know to be persisted in σ by the current thread.
$\ell \succsim_{\mathsf{f}} \sigma$	ℓ is known to be flushed in σ by the current thread.
$\ell \succsim_{\mathsf{s}} \sigma$	ℓ is know to be stored in σ by the current thread.
$\langle \mathsf{ifRec} \rangle_{\ell}\, P$	The proposition P holds if ℓ was persisted before the last crash.
$\langle \mathsf{PC} \rangle\, P$	P holds after a crash.
$\langle \mathsf{PCF} \rangle\, P$	P holds after a crash *if* the writes flushed by the current thread reaches persistent memory before the crash.
$\mathrm{crashedIn}(\ell, \sigma)$	At the last crash the location was recovered in the state σ before the crash.
$\langle \mathsf{PF} \rangle\, P$	P holds after the next asynchronous fence.
$\langle \mathsf{PF_S} \rangle\, P$	P holds after the next synchronous fence.

Figure 4.9: Overview over assertions in Spirea

$$\textsc{mod-sep}\quad \langle \mathsf{M} \rangle\, P * \langle \mathsf{M} \rangle\, Q \vdash \langle \mathsf{M} \rangle (P * Q) \qquad \textsc{mod-mono}\quad \frac{P \vdash Q}{\langle \mathsf{M} \rangle\, P \vdash \langle \mathsf{M} \rangle\, Q} \qquad \textsc{mod-intro}\quad P \vdash \langle \mathsf{M} \rangle\, P$$

$$\textsc{mod-idemp}\quad \langle \mathsf{M} \rangle \langle \mathsf{M} \rangle\, P \dashv\vdash \langle \mathsf{M} \rangle\, P \qquad \textsc{mod-elim}\quad \langle \mathsf{M} \rangle\, P \vdash P$$

Figure 4.10: General rules for modalities

Knowledge vs. resources In Iris a persistent proposition is one that does not entail ownership but only represents duplicable knowledge. $\Box P$ means that P always holds, and a proposition P is persistent if $P \vdash \Box P$. To avoid confusion with the different notions of "persistent" we use the word "knowledge" to mean persistent propositions. For example, $n = 37$ is knowledge and $\ell \hookrightarrow 37$ is not.

Conventions for modalities As we will see, Spirea contains a healthy number of modalities. In order to avoid having to introduce a plethora of symbols, we denote

LB-KNOWLEDGE
$$\frac{l \in \{\mathsf{p}, \mathsf{f}, \mathsf{s}\}}{\ell \succsim_{\mathsf{l}} \sigma \vdash \Box\, \ell \succsim_{\mathsf{l}} \sigma}$$

LB-PERSISTENT-FLUSH-STORE
$$\ell \succsim_{\mathsf{p}} \sigma \vdash \ell \succsim_{\mathsf{f}} \sigma \vdash \ell \succsim_{\mathsf{s}} \sigma$$

OBJ-NOFLUSH-NOBUFFER
$$\langle \mathsf{obj} \rangle\, P \vdash \langle \mathsf{NF} \rangle\, P \vdash \langle \mathsf{NB} \rangle\, P$$

MAPSTO-STORE-LB
$$\ell \hookrightarrow_a \vec{\sigma}\sigma \vdash \ell \succsim_{\mathsf{s}} \sigma$$

MAPSTO-LB-PERS
$$\frac{\sigma_2 \not\sqsubseteq \sigma_1 \qquad \ell \succsim_{\mathsf{p}} \sigma_2 \qquad \ell \hookrightarrow_{\mathsf{na}} \sigma_1 \vec{\sigma}}{\ell \hookrightarrow_{\mathsf{na}} \vec{\sigma}}$$

MAPSTO-NA-STORE-LB
$$\frac{\ell \succsim_{\mathsf{s}} \sigma_1 \qquad \ell \hookrightarrow_{\mathsf{na}} \vec{\sigma}\sigma_2}{\sigma_1 \sqsubseteq \sigma_2}$$

POST-FENCE-NO-FLUSH
$$\langle \mathsf{PF} \rangle\, \langle \mathsf{NF} \rangle\, P \vdash P$$

PFS-PF
$$\langle \mathsf{PF} \rangle\, P \vdash \langle \mathsf{PF_S} \rangle\, P$$

REC-IN-IF-REC
$$\mathrm{crashedIn}(\ell, \sigma) * \langle \mathsf{ifRec} \rangle_\ell\, P \vdash P$$

Rules for the post-crash modality

PC-NA-MAPSTO
$$\ell \hookrightarrow_{\mathsf{na}} \sigma_1\sigma_2 \cdots \sigma_n \vdash \langle \mathsf{PC} \rangle\, \langle \mathsf{ifRec} \rangle_\ell\, \exists i \leq n.\, \ell \hookrightarrow_{\mathsf{na}} \psi(\sigma_1)\psi(\sigma_2) \cdots \psi(\sigma_i) * \mathrm{crashedIn}(\ell, \sigma_i)$$

PC-AT-MAPSTO
$$\ell \hookrightarrow_{\mathsf{at}} \sigma \vdash \langle \mathsf{PC} \rangle\, \langle \mathsf{ifRec} \rangle_\ell\, \exists \sigma_r.\, \ell \hookrightarrow_{\mathsf{at}} \psi(\sigma_r) * \mathrm{crashedIn}(\ell, \sigma_r)$$

PC-INVARIANT
$$\boxed{\ell \mid \pi} \vdash \langle \mathsf{PC} \rangle\, \langle \mathsf{ifRec} \rangle_\ell\, \boxed{\ell \mid \pi}$$

PC-PCF
$$\langle \mathsf{PC} \rangle\, P \vdash \langle \mathsf{PCF} \rangle\, P$$

PC-PERSIST-LB
$$\ell \succsim_{\mathsf{p}} \sigma \vdash \langle \mathsf{PC} \rangle\, \ell \succsim_{\mathsf{p}} \psi(\sigma) * \exists \sigma_r \sqsupseteq \sigma.\, \mathrm{crashedIn}(\ell, \sigma_r)$$

PCF-FLUSH-LB
$$\ell \succsim_{\mathsf{f}} \sigma \vdash \langle \mathsf{PCF} \rangle\, \ell \succsim_{\mathsf{p}} \psi(\sigma) * \exists \sigma_r \sqsupseteq \sigma.\, \mathrm{crashedIn}(\ell, \sigma_r)$$

REC-IN-AGREE
$$\mathrm{crashedIn}(\ell, \sigma) * \mathrm{crashedIn}(\ell, \sigma') \vdash \sigma = \sigma'$$

Figure 4.11: Selected rules for assertions and modalities in the logic

modalities (except already well-known ones) as $\langle M \rangle$ where M is a mnemonic for the modality. All of our modalities satisfy basic structural rules such as MOD-SEP and MOD-MONO seen in Figure 4.10. Additionally, some modalities are monadic (they satisfy MOD-INTRO, *etc.*) or comonadic (they satisfy MOD-ELIM, *etc.*).

Crash-Aware Invariants As mentioned, one of the key innovations in Spirea is *crash-aware invariants* (or just invariants for short when it is clear from the context that we are not talking about Iris invariants). We start things off with the definition. The definition uses concepts in Spirea that we have yet to see, but these can be

HT-FLUSH
$$\{\ell \succsim_{\mathsf{s}} \sigma\}\ \mathbf{flush}\ \ell\ \{\langle \mathsf{PF}\rangle(\ell \succsim_{\mathsf{f}} \sigma) * \langle \mathsf{PF_S}\rangle(\ell \succsim_{\mathsf{p}} \sigma)\}$$

HT-FENCE-SYNC
$$\{\langle \mathsf{PF_S}\rangle\ P\}\ \mathbf{fence_{sync}}\ \{P\}$$

HT-FENCE
$$\{\langle \mathsf{PF}\rangle\ P\}\ \mathbf{fence}\ \{P\}$$

HT-NA-ALLOC
$$\{\phi(\sigma, v)\}\ \mathbf{ref}_{\mathsf{na}}\ v\ \{\ell.\ \boxed{\ell}\boxed{\pi} * \ell \hookrightarrow_{\mathsf{na}} \sigma\}$$

HT-AT-ALLOC
$$\{\phi(\sigma, v)\}\ \mathbf{ref}_{\mathsf{at}}\ v\ \{\ell.\ \boxed{\ell}\boxed{\pi} * \ell \hookrightarrow_{\mathsf{at}} \sigma\}$$

HT-NA-READ
$$\left\{\begin{array}{l}\boxed{\ell}\boxed{\pi} * \ell \hookrightarrow_{\mathsf{na}} \vec{\sigma}\sigma * \\ (\langle \mathsf{obj}\rangle\ \forall v.\ \phi(\sigma, v) \mathrel{-\!\!*} Q(v) * \phi(\sigma, v))\end{array}\right\}\ !_{\mathsf{na}}\ \ell\ \{w.\ \ell \hookrightarrow_{\mathsf{na}} \vec{\sigma}\sigma * Q(w)\}$$

HT-NA-WRITE
$$\{\boxed{\ell}\boxed{\pi} * \ell \hookrightarrow_{\mathsf{na}} \vec{\sigma}\sigma * \phi(\sigma_t, v_t) * \sigma \sqsubseteq \sigma_t\}\ \ell :=_{\mathsf{na}} v_t\ \{\ell \hookrightarrow_{\mathsf{na}} \vec{\sigma}\sigma\sigma_t\}$$

HT-AT-READ
$$\left\{\begin{array}{l}\boxed{\ell}\boxed{\pi} * \ell \hookrightarrow_{\mathsf{at}} \sigma * \\ \langle \mathsf{obj}\rangle\ \forall \sigma_r \sqsupseteq \sigma, v_r.\ \phi(\sigma_r, v_r) \mathrel{-\!\!*} Q(\sigma_r, v_r) * \phi(\sigma_r, v_r)\end{array}\right\}$$
$$!_{\mathsf{at}}\ \ell$$
$$\{v.\ \exists \sigma_r \sqsupseteq \sigma.\ \ell \hookrightarrow_{\mathsf{at}} \sigma_r * \langle \mathsf{PF}\rangle\ Q(\sigma_r, v)\}$$

HT-AT-WRITE
$$\left\{\begin{array}{l}\boxed{\ell}\boxed{\pi} * \ell \hookrightarrow_{\mathsf{at}} \sigma * \phi(\sigma_t, v_t) * \sigma \sqsubseteq \sigma_t * \\ \forall \sigma_c \sqsupseteq \sigma, v, v_c.\ \phi(\sigma, v) \mathrel{-\!\!*} \phi(\sigma_t, v_t) \mathrel{-\!\!*} \phi(\sigma_c, v_c) \mathrel{-\!\!*} \sigma_c \sqsubseteq \sigma_t \sqsubseteq \sigma_c\end{array}\right\}$$
$$\ell :=_{\mathsf{at}} v_t$$
$$\{\ell \hookrightarrow_{\mathsf{at}} \sigma_t\}$$

Figure 4.12: Selected program rules for memory operations

$$
\left\{\begin{array}{l}\boxed{x}\boxed{\pi_x} * \boxed{y}\boxed{\pi_{y,mp}} * \\ x \hookrightarrow_{\mathsf{na}} [\bot] * y \hookrightarrow_{\mathsf{at}} \bot * \mathrm{tok}_1\end{array}\right\}
$$

$$
\begin{array}{l||l}
\left\{\begin{array}{l}x \hookrightarrow_{\mathsf{na}} [\bot] * \\ y \hookrightarrow_{\mathsf{at}} \bot\end{array}\right\} & \{y \hookrightarrow_{\mathsf{at}} \bot * \mathrm{tok}_1\} \\
x :=_{\mathsf{na}} 37; & \textbf{if } !_{\mathsf{at}}\, y = 1 \\
\{x \hookrightarrow_{\mathsf{na}} [\bot, \top]\} & \textbf{then} \\
y :=_{\mathsf{at}} 1 & \quad \left\{\begin{array}{l}y \hookrightarrow_{\mathsf{at}} \top * \\ x \hookrightarrow_{\mathsf{na}} [\bot, \top]\end{array}\right\} \\
\{y \hookrightarrow_{\mathsf{at}} \top\} & \quad \textbf{assert } !_{\mathsf{na}}\, x = 37 \\
 & \quad \{\textbf{true}\}
\end{array}
$$

$$
\{\textbf{true}\}
$$

Figure 4.13: Proof outline for the message passing example.

$$
\left\{\begin{array}{l}\boxed{x}\boxed{\pi_x} * \boxed{y}\boxed{\pi_{y,ff}} * \\ x \hookrightarrow_{\mathsf{na}} [\bot] * y \hookrightarrow_{\mathsf{na}} [\bot] \\ y \succsim_{\mathsf{p}} \sigma_y * y \hookrightarrow_{\mathsf{na}} [\sigma_y]\end{array}\right\}
$$

$$
\begin{array}{l}
x :=_{\mathsf{na}} 37; \\
\{x \hookrightarrow_{\mathsf{na}} [\bot, \top] * x \succsim_{\mathsf{s}} \top\} \\
\textbf{flush } x; \\
\{\langle \mathsf{PF} \rangle\, x \succsim_{\mathsf{f}} \top\} \\
\textbf{fence}; \\
\{x \succsim_{\mathsf{f}} \top * y \hookrightarrow [\bot]\} \\
y :=_{\mathsf{na}} 1 \\
\{y \hookrightarrow_{\mathsf{na}} [\bot, \top]\}
\end{array}
\quad \circlearrowleft \quad
\left\{\begin{array}{l}\exists \sigma_x, \sigma_y. \\ \quad \boxed{x}\boxed{\pi_x} * \boxed{y}\boxed{\pi_{y,ff}} * \\ \quad x \succsim_{\mathsf{p}} \sigma_x * x \hookrightarrow_{\mathsf{na}} [\sigma_x] * \\ \quad y \succsim_{\mathsf{p}} \sigma_y * y \hookrightarrow_{\mathsf{na}} [\sigma_y]\end{array}\right\}
$$

$$
\begin{array}{l}
\textbf{if } !_{\mathsf{at}}\, y = 1 \\
\textbf{then} \\
\quad \{\sigma_y = \top * x \succsim_{\mathsf{f}} \top\} \\
\quad \{x \hookrightarrow_{\mathsf{na}} [\top]\} \\
\quad \textbf{assert } !_{\mathsf{na}}\, x = 37 \\
\quad \{\mathsf{True}\}
\end{array}
$$

Figure 4.14: Proof outline for the asynchronous fence example

ignored for now. We will refer back to, and provide explanations of, the definition throughout the section.

Definition 4.6.1. *A* crash-aware invariant π *consists of: a set of states* Σ*, a preorder* $\sqsubseteq$ *on* Σ*, a* write assertion $\phi : \Sigma \times \mathit{Val} \to \mathit{dProp}$ *(dProp is the type of propositions in Spirea), and a* state-change *function* $\psi : \Sigma \to \Sigma$ *that is monotone w.r.t.* $\sqsubseteq$*. The data must satisfy the following two conditions:*

(1) $\forall \sigma \in \Sigma, v \in \mathit{Val}.\ \phi(\sigma, v) \vdash \langle \mathsf{NB} \rangle\, \phi(\sigma, v)$

(2) $\forall \sigma \in \Sigma, v \in \mathit{Val}.\ \phi(\sigma, v) \vdash \langle \mathsf{PCF} \rangle\, \phi(\psi(\sigma), v)$.

For an invariant π we refer to its components, say ϕ, with $\pi.\phi$, but more often we just write ϕ when it is clear from context which invariant the component is from.

In the logic every location ℓ is associated with a specific invariant π throughout its lifetime. This invariant is chosen dynamically when the location is allocated by using the rules Ht-na-alloc and Ht-at-alloc that appear in Figure 4.12. In these rules the *invariant assertion* $\boxed{\ell \mid \pi}$ appears in the postcondition. It denotes the knowledge that ℓ is associated with π. On the first line of the proof outlines (Figure 4.13 and Figure 4.14) we see invariant assertions for both x and y. For such preexisting locations invariants can be picked at the beginning of the proof (we will see the details in Section 4.7.2). The invariant assertions hold throughout the proofs, but to avoid clutter in the outlines we do not repeat unchanged resources.

Invariant States Consider a thread reading y in parallel with the sending thread in Figure 4.13. Such a thread can observe the initial value of 0, the final value of 1, and once it sees the latter it never sees the former again. We can represent the situation with a state transition system (STS): y can be in one of the two states $\bot$ and $\top$ (corresponding to 0 and 1 respectively) and it can transition from $\bot$ to $\top$—we say that $\top$ is a greater state and write $\bot \sqsubseteq \top$. A key insight going back to GPS is that the above can be put to good use in a logic by letting each location be governed by an STS as part of its invariant. This is the purpose of Σ and $\sqsubseteq$ in Definition 4.6.1, they represent an STS that the location must evolve through. In the examples, we use the described STS with two states for x and y as both locations are written exactly once. When writing to a location a state $\sigma \in \Sigma$ must be picked such that the states *grow monotonically* with each write. For a single location the memory model ensures all threads observe writes to it in the same order, and the invariant rules ensure that this order corresponds to an increasing order of states. Furthermore, while the weak memory order and the persist order do not agree in general they do coincide for a single location. We hence observe that we can soundly adopt the use of STSs for persistent memory such that they represent both the weak memory order (as in GPS) as well as the persist order.

Write Assertions A release-write can transfer resources from one thread to another, as in Figure 4.13 where the write to y carries with it the right to access

x. The *write assertion* in invariants describe such resources. A write assertion, $\phi : \Sigma \times \text{Val} \to \mathsf{dProp}$, is parameterized over the invariant's states and values. The idea is that for *every* write to the location governed by the invariant, say with value v and state σ, the assertion $\phi(\sigma, v)$ holds.

As a simple example, in both Figure 4.13 and Figure 4.14 we pick the following write assertion for x:

$$\phi_x(\sigma, v) \triangleq (\sigma = \bot * v = 0) \lor (\sigma = \top * v = 37). \tag{4.1}$$

The write assertion gives *meaning* to the states by establishing a correspondence between them and specific values. Having the state determine the value in this way is a common pattern. Since x is not used for resource transfer this suffices for its write assertion. When we verify the message passing example below we see an example where resource transfer is needed.

Points-To Predicates Points-to predicates in Spirea have the form $\ell \hookrightarrow_a \vec{\sigma}$. Here $\vec{\sigma}$ is a *sequence* of states that has been written to ℓ.[10] When a is na (respectively at) we say that the points-to predicate is non-atomic (atomic) and the location can then only be accessed using the na (at) access mode. On the first line in Figure 4.13 we use a non-atomic points-to predicate for x and an atomic one for y.

The non-atomic points-to predicate entails exclusive ownership over ℓ and supports fractional permissions, denoted $\ell \hookrightarrow^q_{\mathsf{na}} \vec{\sigma}$ for a fraction $q \in (0; 1]$. (As usual, we often omit the fraction q if it is 1.) Hence, in Figure 4.13 we need to transfer ownership over the points-to predicate for x from the left thread to the right thread. The sequence $\vec{\sigma}$ contains (at least) all writes that can ever be read again, both before and after a crash. It may be surprising that we use a sequence of states for non-atomics as prior logics for weak memory have been able to establish "normal" points-to predicates for non-atomics that associate a location with a single value, thereby completely hiding the weak semantics. However, this is not possible in the persistent setting, where the asynchronicity of writes and the fact that crashes are ever-present (in contrast to data-races that can be avoided) means that at least some old states must be remembered. For instance, in Figure 4.14, at the end of executing the right thread we have the resource $x \hookrightarrow_{\mathsf{na}} [\bot, \top]$. This preserves the precise information that after a crash x can have the value 0 or 37.

The atomic points-to predicate does not entail ownership and is knowledge. Hence, several threads can access atomic locations in parallel. This is needed for y in Figure 4.13 where both threads own $y \hookrightarrow_{\mathsf{at}} \bot$ initially. Since several threads can write to an atomic location without any synchronization the sequence of states $\vec{\sigma}$ is only *partial*. Other threads may have performed writes that the current thread is not aware of and that are thus not in $\vec{\sigma}$. Hence, for the atomic points-to predicate states can freely be dropped and, in practice, it often suffices to remember only the

[10] By convention, we name sequences with arrows $\vec{\sigma}$ and use juxtaposition for concatenation. For instance, $\vec{\sigma}\sigma$ is a sequence starting with $\vec{\sigma}$ and ending with σ. For a concrete sequence we sometimes use list notation, as in $[\sigma_1, \sigma_2, \sigma_3]$.

latest write. Therefore, and as the rules that take advantage of the entire sequence of states in the atomic case are fairly involved, in the remainder of the paper we only use the atomic points-to predicate with a single state and present specialized proof rules for this simpler case.

Message Passing Example The message passing example contains reads and writes of all kinds. This makes it a great example to explain the read and write rules in Spirea and to see how invariants facilitate resource transfer between threads. We start with the write assertion for y:

$$\phi_{y,mp}(\bot, v) \triangleq v = 0 * \mathsf{tok}_0$$
$$\phi_{y,mp}(\top, v) \triangleq v = 1 * (x \hookrightarrow_{\mathsf{na}} [\bot, \top] \lor \mathsf{tok}_1)$$

The equalities on v should be clear. The two tokens, tok_0 and tok_1, are *exclusive*: Only one of each exist and hence $\mathsf{tok}_n * \mathsf{tok}_n$ is a contradiction (*i.e.,* it implies false). This is a standard construction using Iris ghost state. The purpose of these tokens and the disjunction is best explained in the proof.

Notice how we split the initial resources from the first to the second line in Figure 4.13. The left thread gets the non-atomic points-to predicate for x and the right thread gets the token tok_1. The rest is knowledge, so both threads get a copy. We now cover the two writes and the two reads.

Non-atomic write ($x :=_{na} 37$). The rule HT-NA-WRITE states that to write v to a non-atomic location one must pick a target state σ_t. We choose $\top$. The precondition requires an invariant assertion, a points-to predicate, that the write assertion holds, and that the new state preserves the order of the states. All of these are trivial: we have an invariant assertion, a points-to predicate ending in the state $\bot$, $\phi_x(\top, 37)$ is immediate from the definition in Equation (4.1), and $\bot \sqsubseteq \top$ per definition. In the postcondition we receive an updated points-to predicate with the newly written state appended at the end. Non-atomic writes are usually this trivial, as precise information about them is known.

Atomic write ($y :=_{at} 1$). The first line of the precondition of HT-AT-WRITE is similar to what we just saw for non-atomics. We pick the state $\top$ for the write and show the write assertion by choosing the left side of the disjunction and using our points-predicate for x. That is, we transfer ownership over x into the invariant. The conjunct on the second line of the precondition of HT-AT-WRITE serves to maintain the monotone order of writes. Since atomic locations can be shared, we need to account for potential racy writes to the location. The universally quantified σ_c represents such a write and the obligation is to show that it and the written state σ_t can transition between each other, $\sigma_c \sqsubseteq \sigma_t$ and $\sigma_t \sqsubseteq \sigma_c$. This ensures that they are equivalent w.r.t. the preorder and that the order of the states is preserved no matter which of the two racy writes end up first in the memory order. To show this obligation the writer can assume the assertion of both the original state σ, the concurrent state σ_c, and the written state σ_t. If we look at the whole program we are verifying it is clear that there are no concurrent writes to y. But, as we are verifying

the left thread modularly in isolation, we must be able to draw this conclusion based solely on the invariant. To this end, we assume some concurrent write σ_c and must show $\sigma_c \sqsubseteq \top \sqsubseteq \sigma_c$. If $\sigma_c = \top$ the conclusion is trivial. If the $\sigma_c = \bot$ the conclusion is impossible. Fortunately, in this case we have the invariant for $\bot$ *twice*, hence we have the token tok_0 twice, which is a contradiction. Intuitively, the token tok_0 represents the right to write $\bot$ to x, and since only one token exists, this state can only ever be written once.

Atomic read ($?_{at}\, y$). Now in the right thread we, apply HT-AT-READ. At the present time we can ignore the $\langle \mathsf{obj} \rangle$ and $\langle \mathsf{PF} \rangle$ in the rule. We have the invariant and the points-to predicate required in the precondition. The last conjunct lets us open the invariant, access its content, and potentially transfer resources in and out of the invariant. The resource Q represents the resources that we want to *transfer out* of the invariant. We use

$$Q(\bot, v) \triangleq v = 0 \qquad\qquad Q(\top, v) \triangleq v = 1 * x \hookrightarrow_{\mathsf{na}} [\bot, \top].$$

Hence, if we read 1 we transfer the points-to predicate for x out. We need to show the wand in HT-AT-READ. For some read state σ_r and value v_r the reader receives the invariant $\phi(\sigma_r, v_r)$ (the antecedent of the wand). We now have access to the content of the invariant, but, since the invariant also appears in the consequent the access is temporary—we say that we have to *close* the invariant. If $\sigma_r = \bot$ then $Q(\bot, v_r)$ is plain knowledge and showing it and the invariant is trivial. If $\sigma_r = \top$ then we use $x \hookrightarrow_{\mathsf{na}} [\bot, \top]$ to show $Q(\top, v_r)$. However, now we can not use this points-to predicate to close the invariant. Fortunately, the invariant contains a disjunction and we can show the right disjunct using the tok_1 that the right thread owns. That is, we transfer tok_1 *in to* the invariant in order to transfer $x \hookrightarrow_{\mathsf{na}} [\bot, \top]$ *out of* the invariant. This sort of reasoning is well-known to readers familiar with Iris invariants, but it is in fact significantly stronger than the read rule in GPS and iGPS. In these logics, a read can only transfer *knowledge* out of the invariant—transferring ownership over resources is not possible! Returning to the proof, having shown the preconditions for the read, we now get Q in the postcondition. The case where we read 0 is trivial, so we consider the case where we read 1 and enter the branch. In this case we have the points-to predicate for x after the read, as shown in the proof outline. All that remains is to show that the read of x yields 37.

Non-atomic read ($?_{na}\, x$). We apply HT-NA-READ which is much like the read rule for atomics, which we just went through. The notable difference is that for a non-atomic is it certain that the last state in the points-to predicate (σ in the rule) is read. Hence, the rule does not quantify over some read state. When applying the rule we pick $Q(v) \triangleq v = 37$, which is easy to show when opening the invariant, and which gives us what we need.

RMW Operations We have now seen the rules for reading and writing. Spirea also contains rules for the RMW operations **CAS** and **FAA**. We do not include these rules for space reasons and since they are rather complex. Since RMW operations

are simultaneously both a read and a write, our rules for these essentially combine the read and the write rule. The rules require that the write assertion is shown for the read value (like HT-AT-READ) *and* the written value (like HT-AT-WRITE). This is in contrast to other CSLs for weak memory, where the equivalent notion to our write assertion would not have to be shown for the read value. This makes resource transfer through RMW operations more restricted, but ensures that invariants are sound. In Section 4.8.3 we show how to combine Spirea with BaseSpirea for examples where the CAS rule is not strong enough, in Section 4.8.4 we see an example where the CAS rule is sufficient, and we discuss the limitation further in Section 4.9.

Flushes and Fences To verify programs using flushes and fences we need assertions that capture the knowledge gained by these operations. Consider the pre-crash code in Figure 4.14. Just after writing to x the thread merely knows that the write with state $\top$ exists (which implies that a successive read reads this or a more recent state). Knowledge of this form is captured by the *store lower bound* assertion $\ell \succsim_{\mathsf{s}} \sigma$. The program then flushes x and carries out an asynchronous fence. After this the thread knows that the write will persist before any succeeding writes. This form of knowledge is represented by the *flush lower bound* $\ell \succsim_{\mathsf{f}} \sigma$. Suppose the program had instead carried out a *synchronous* fence. The thread would then know that the write had been saved to persistent memory. The *persist lower bound* $\ell \succsim_{\mathsf{p}} \sigma$ represents this knowledge.

These assertions are lower bounds, in the sense that $\ell \succsim_{l} \sigma$ implies knowledge of a write in at least state σ but not that this is necessarily the most recent state. This, together with the fact that states grow monotonically, makes the assertions knowledge (LB-KNOWLEDGE). The three lower bound relations are ordered as shown in LB-PERSISTENT-FLUSH-STORE since a state is written before it is flushed, and since a synchronous fence is strictly stronger than an asynchronous fence.

Following the above, the effect of flushing a location ℓ and a fence is then that the most recent write σ known to the flushing thread advances from $\ell \succsim_{\mathsf{s}} \sigma$ to $\ell \succsim_{\mathsf{f}} \sigma$ (in the case of an asynchronous fence) or to $\ell \succsim_{\mathsf{p}} \ell$ (in the case of a synchronous). The rules for flush and fence should achieve this while taking the following three things into account: (1) **flush** and **fence** are two separate operations and the fence may not necessarily immediately follow the flush. (2) A fence can apply to arbitrarily many preceding flushes. (3) A fence is not only used in combination with a flush. As in Figure 4.2b it is also used in combination with an acquire-read to acquire persist information from the release-write. We want our program rules to support all these usage patterns. To this end Spirea includes two *fence modalities*: $\langle \mathsf{PF} \rangle$ and $\langle \mathsf{PF}_\mathsf{S} \rangle$. The assertions $\langle \mathsf{PF} \rangle\, P$ and $\langle \mathsf{PF}_\mathsf{S} \rangle\, P$ mean that P holds after the next asynchronous fence and synchronous fence, respectively.

In Figure 4.14 we apply HT-FLUSH at the **flush** operation. The precondition takes a store lower bound that we can extract from $x \hookrightarrow_{\mathsf{na}} [\bot, \top]$ using MAPSTO-STORE-LB. The postcondition contains both a flush lower bound under $\langle \mathsf{PF} \rangle$ and a persist lower bound under $\langle \mathsf{PF}_\mathsf{S} \rangle$ such that the **flush** can later be matched with both types of

fences. In our case we only need the flush lower bound. At **fence** we use HT-FENCE. This rule (and HT-FENCE-SYNC) exactly matches the intuition of the fence modalities. If P holds under a fence modality, then executing a fence eliminates the modality. In our case this means that we have the flush lower bound after the fence. Note, that since the fence modalities are modalities and have a separation rule (as MOD-SEP) the result from several flushes can be combined and extracted with a single fence. In the rule HT-AT-READ the extracted resource Q is under a fence modality which enforces that a fence be used when necessary. As such, using modalities for fences neatly achieves the requirements stated above.

To conclude the proof of the pre-crash program in Figure 4.14 we define the write assertion for y

$$\phi_{y,\mathit{ff}}(\sigma, v) \triangleq (\sigma = \bot * v = 0) \vee (\sigma = \top * v = 1 * x \succsim_{\mathsf{f}} \top). \tag{4.2}$$

The assertion contains a flush lower bound for x when y has the state $\top$. To prove this at the write to y we use the flush lower bound gained from the flush and the fence. In the next section we see how this is used to verify the recovery code.

Non-Deterministic Post-Crash Modality To verify the entire flush and fence example, including the recovery code, we apply HTR-IDEMPOTENCE where we must pick a crash condition Q_c. The R in the rule is the precondition for the recovery code in Figure 4.14. As a crash condition we pick $\langle \mathsf{PC} \rangle\, R$. Using the post-crash modality directly in the crash condition like this is common in Spirea as it turns out to be the most convenient approach in practice. Proving the wand for R in HTR-IDEMPOTENCE becomes trivial, and the proof effort is concentrated on showing the crash condition at every step. In order to do this, we need to understand how our post-crash modality works. The rules for it appear in the lower half of Figure 4.11.

Consider how an invariant $\boxed{\ell \mid \pi}$ should change at a crash. As we have mentioned, our invariants are crash-aware, and we want them to survive crashes. At the same time our programming language supports allocation, and since allocations might not persist before a crash, locations can be entirely lost at crashes. If a location is not lost after a crash, we say that it was *recovered* after the crash, and only in this case would it make sense still to have an invariant assertion for it. Such a situation is common, and we capture it by an *if-recovered* modality: the assertion $\langle \mathsf{ifRec} \rangle_\ell\, P$ mean that if the location ℓ was recovered at the last crash, then P (which would typically mention ℓ) holds. The rule PC-INVARIANT is now clear: it preserves invariants for locations as long as they are recovered.

The if-recovered modality captures some of the non-determinism at a crash. Additional non-determinism is present in the rule PC-NA-MAPSTO for non-atomic points-to predicates. Here the non-determinism is represented by the existential quantifier. The rule states that, for some i, only the first i states of the points-to predicate exist after the crash (ignore the ψ in the rule for now, it is explained later in the section). For state σ_i, the rule contains the assertion $\mathsf{crashedIn}(\ell, \sigma_i)$. The meaning of this assertion is that σ_i is the most recent recovered state for ℓ, which is

exactly how it is used in the rule. Only one such state exists so two such assertion must agree on the state REC-IN-AGREE. The crashedIn(ℓ, σ_i) assertion also implies that ℓ was in fact recovered and it can thus be used to eliminate the if-recovered modality as seen in REC-IN-IF-REC.

The only way to know with certainty that a location will be recovered is through a persistent lower bound $\ell \succsim_{\mathsf{p}} \sigma$. Per PC-PERSIST-LB a persistent lower bound is preserved across a crash (again, ignore ψ) and the most recent recovered state σ_r has to be at least σ. In contrast, a store lower bound clearly offers no knowledge after a crash as it only deals with the weak memory order. But what about a flush lower bound? A flush lower bound (and the **flush** and **fence** it represents) provides no knowledge of the state of the persistent memory, and as such it too has no meaningful interaction with the post-crash modality. Its effect is more subtle and only restricts the order of persists, as in the flush and fence example where the write to x persists before the write to y. To tease out this effect in the logic we introduce a *post-crash-flush* modality: $\langle \mathsf{PCF} \rangle\, P$ means that P holds after a crash *if* we are in the fortunate scenario where everything flushed and fenced actually reached persistent memory before the crash. In this case, a flush lower bound is just as good as a persist lower bound, and PCF-FLUSH-LB results in the same resources under the post-crash-flush modality as we saw in PC-PERSIST-LB. The post-crash-flush modality is weaker than the post-crash modality (PC-PCF) so the rules for the post-crash modality also applies to it.

The single place where we use the post-crash-flush modality is in the second condition for write assertions in the definition of invariants (Definition 4.6.1). This condition is necessary to make it possible to transfer invariants across a crash, *i.e.,* it is used to prove soundness of PC-INVARIANT. During this proof the write assertion ϕ must be established for the recovered state σ. Since σ was recovered, it must have persisted before the crash, and thus anything flushed and fenced prior to σ (that ϕ might know about) is also guaranteed to have persisted. As such, using the post-crash-flush modality in the condition is sufficiently strong, and allows us to use PCF-FLUSH-LB to show that ϕ holds for the recovered state. We note that, in our example, it is easy to show (using PCF-FLUSH-LB) that the second condition in Definition 4.6.1 does indeed hold for $\phi_{y,\mathit{ff}}$.

By using the rules for the post-crash modality it is now quite trivial to show the crash condition at every program point in the pre-crash code. And with the resources after the crash established, proving the recovery code is also straightforward. If reading 1 from y the recovery code learns that $\sigma_y = \top$ and acquires the resource $x \succsim_{\mathsf{f}} \top$ from the invariant. The flush lower bound can be weakened to $x \succsim_{\mathsf{s}} \top$ per LB-PERSISTENT-FLUSH-STORE, and combined with $x \hookrightarrow_{\mathsf{na}} [\sigma_x]$ the rule MAPSTO-NA-STORE-LB implies that $\top \sqsubseteq \sigma_x$, which in turn means that $\sigma_x = \top$. With that established reading x is sure to result in 37 just as what we saw in the message passing example.

Subjectivity We now take a step back and consider an issue that we have so far swept under the rug. Propositions in Spirea can be *subjective*. That is, describe facts

that are true from one thread's perspective, but that are not necessarily true from the point of view of other threads. For instance, after the left thread in Figure 4.14 has flushed x it knows $\langle \mathsf{PF} \rangle\, x \succsim_{\mathsf{f}} \top$. But, as a flush by one thread provides no orderings across threads, it would be unsound to transfer this resource to another thread. We thus need to make certain restrictions on resource transfer. We accomplish this with three comonadic modalities. The *no-buffer modality*, $\langle \mathsf{NB} \rangle\, P$, means that P does not contain any of the post-fence modalities.[11] The first condition in Definition 4.6.1 uses this modality to ensures that the described unsound transfer is not possible. Write assertions that invole $\langle \mathsf{PF} \rangle$ or $\langle \mathsf{PF_S} \rangle$ do not pass this requirement. The *no-flush modality*, $\langle \mathsf{NF} \rangle\, P$, adds the requirement that P does not contain knowledge of flushes $\ell \succsim_{\mathsf{f}} \sigma$. Assertions of the form $\langle \mathsf{NF} \rangle\, P$ are of interest as they can safely be extracted from the post-fence modality per POST-FENCE-NO-FLUSH. This is what allowed us to ignore the $\langle \mathsf{PF} \rangle$ modality when we applied HT-AT-READ in Figure 4.13 as the Q we picked did not use flush lower bounds. Finally, the *objectively modality*, $\langle \mathsf{obj} \rangle\, P$, means that P holds at *all* points of view of the memory and thus that it is always sound to transfer P between threads. Examples are $\ell \succsim_{\mathsf{p}} \sigma$ and $\boxed{\ell}\boxed{\pi}$. One use of this modality is in HT-AT-READ where it ensures that the reading thread can not transfer subjective resources to other reading threads.

State-Change Function The final component of invariants that we still have not seen is *state-change functions*. To understand the need for these, consider how we would verify the optimized message passing example in Figure 4.2d. Similar to the verification in Figure 4.14, the write to z need to carry with it the knowledge $x \succsim_{\mathsf{f}} \top$. In order for the left thread to have this knowledge it must acquire $x \succsim_{\mathsf{s}} \top$ when reading y. As such, the write assertion for y must have the form $\phi_y(\top, v) \triangleq v = 1 * x \succsim_{\mathsf{s}} \top$. However, as there are no fences between the writes to x and y, if the *recovery code* were to read y it would be unsound for it to gain the knowledge $x \succsim_{\mathsf{s}} \top$. In other words, the write to y serves to transfer a resource to concurrently running threads that should not be available to recovery code. To capture this, a monotone *state-change function* ψ can change the state of a write after a crash. The idea is that if a write corresponds to the state σ before a crash, it then corresponds to $\psi(\sigma)$ after the crash. This is evident by looking at the crash related rules in Figure 4.11 where states under the post-crash modality always have ψ applied to them. In examples where the above issue does not arise, the state-change function can simply be the identity function, and then the ψs can be ignored as we have done so far.

In order to verify the optimized message passing example we can extend the set of states for y with an additional state σ_{pc} that is below the two other states. The state-change function transitions every write into this state at a crash: $\psi(\sigma) \triangleq \sigma_{pc}$. The write assertion for this state is simply $\phi_y(\sigma_{pc}, v) \triangleq v = 0 \lor v = 1$. This ensures that if the recovery code were to read y it would gain no information whatsoever while still allowing for the desired resource transfer to work.

[11] The name refers to the fact that flushes use a buffer in the operational semantics.

Summary We have now completed our tour of Spirea. We hope it has become clear that it supports thread-local modular reasoning by extending ideas from separation logic, in particular ownership and resource transfer, with a range of modalities, which allow us to capture the subtle conditions under which resource transfer is sound.

4.7 Soundness

In this section we present an overview over the operational semantics of λ_{pmem}, state the soundness theorem of Spirea, and give an overview of the model, including some of the details. Readers who are more interested in seeing Spirea applied to examples can proceed to our case studies in Section 4.8.

4.7.1 Operational Semantics

For readers that skipped Section 4.3 we mention a few details of the operational semantics that are necessary to understand in order to explain the soundness theorem. The semantics of λ_{pmem} is a small-step interleaving operational semantics. Like prior such semantics for weak memory, it is based on *views*. For instance, Bila et al. created a view-based operational semantics for the x86 and ARM persistency models [Bil+22].

The small-step semantics is lifted to a big-step *recoverable execution relation* of the form $e_r; \rho \Rightarrow_{\mathsf{r}} \rho'; s$. Here, e_r is the recovery expression to execute after a crash, ρ and ρ' are machine configurations, and $s \in \{\mathsf{NotCrashed}, \mathsf{Crashed}\}$ is a crash-status. A machine configuration contains the state of entire machine, in particular the memory and all threads. The meaning of the relation is then: a machine in state ρ can execute to state ρ' with zero or more crashes along the way where e_r is executed after every crash. The crash-status indicates whether the execution has been crash free or not. If $s = \mathsf{NotCrashed}$ the execution was crash free and otherwise if $s = \mathsf{Crashed}$ then one or more crashed occurred. As we see below the soundness theorem is stated in terms of the recoverable execution relation.

4.7.2 Soundness

The soundness theorem uses the same safety definition, Definition 4.5.1, that we used for BaseSpirea.

Theorem 4.7.1 (soundness)**.** *Given expressions e and e_r, meta-level predicates on values Φ and Φ_r, a finite set of location L, and for each $\ell \in L$: an access mode a_ℓ, an invariant π_ℓ, a state $\sigma_\ell \in \pi_\ell.\phi$ (i.e., an element of the state of the invariant π_ℓ). Let R be the resource*

$$\ast_{\ell \in \mathsf{dom}(h)} \boxed{\ell}\boxed{\pi_\ell} * \ell \succsim_p \sigma_\ell * \ell \hookrightarrow_{a_\ell} \sigma_\ell.$$

If $R \twoheadrightarrow \ast_{\ell \in \mathsf{dom}(h)} \pi_\ell.\phi(\sigma_\ell, v_\ell)$ *and the recovery Hoare triple* $\{R\}\, e \circlearrowleft e_r\, \{\Phi\}\{\Phi_r\}$ *are provable in Spirea then* $\mathrm{safe}(e, e_r, \langle h, \mathcal{P}\rangle, \Phi, \Phi_r)$ *holds where* $h(\ell) = \langle v_\ell, \bot, \bot, \bot\rangle$ *and where* $\mathcal{P}(\ell) = 0$ *for all* $\ell \in L$.

This theorem applies to a memory that is not necessarily empty to begin with. When applying the soundness theorem one then gets to pick, for each location, its access mode, invariant, initial state, *etc.* The resource R then contains the resources for all locations. It must then be shown that the invariants hold for the initial states, and to do this one can use R. This is such that the initial invariants can use resources (persistent lower bounds, points-to predicates, *etc.*) for other locations.

4.7.3 Model

We give a brief overview of the model of Spirea and highlight some of the underlying key ideas.

Overall Structure Spirea is modeled atop a lower-level logic that we call BaseSpirea. BaseSpirea is constructed as an instantiation of Perennial's program logic framework based on the Iris base logic. This framework gives BaseSpirea basic definitions of the three Hoare triples/quadruples. Based on these we define various assertions to represent the physical state, define a post-crash modality, and prove program rules. However, these program proof rules directly expose the intricacies of the operational semantics, such as views, timestamps, and histories, and thus, while perfectly capable of verifying programs, BaseSpirea is quite tedious to use. We explain BaseSpirea in more detail in Section 4.5. To provide the more abstract reasoning rules of Spirea, we use BaseSpirea to model Spirea. It is at this level that we add crash-aware invariants, the facilities for handling persistent memory instructions without explicit mention of views, and a post-crash modality that works for the higher-level assertions.

Crash-Aware Invariants As mentioned in the introduction, a key challenge w.r.t. the model of Spirea's crash-aware invariants is that it is not clear how Iris invariants can be reconciled with crashes. We therefore take a different approach to invariants than other Iris-based logics for weak memory in that we do not model our crash-aware invariants using Iris invariants. Instead our model includes the resources for invariants inside the *state interpretation*. The state interpretation is a resource that is threaded through Hoare triples/quadruples in the program logic. With this approach the content of invariants is only available in the context of a Hoare triple/quadruple (as opposed to Iris invariants that can be accessed independently of a program). However, this is the case already in prior logics for weak memory, as accessing invariants in a weak memory model needs physical synchronization. The benefit of our approach is that when a crash occurs (more precisely, when proving soundness of Htr-idempotence), the resources belonging to all invariants are found inside the state interpretation, and can then be systematically updated to account for the crash.

Post-Crash Modality We explain our post-crash modality with a simplified sketch of its model that highlights the key ideas.

$$[\![\langle \mathsf{PC} \rangle\, P]\!] \triangleq \lambda \mathcal{T}, \vec{\gamma}_{old}. \forall \vec{\gamma}_{new}. R(\vec{\gamma}_{old}, \vec{\gamma}_{new}) \mathrel{-\!\!*} R(\vec{\gamma}_{old}, \vec{\gamma}_{new}) * [\![P]\!](\langle \bot, \bot, \bot \rangle, \vec{\gamma}_{new})$$

The semantic domain of propositions in Spirea is monotone predicates over thread views and a record of ghost names (denoted $\vec{\gamma}$). This explains why the model of the modality is a function taking two such arguments. Since resources are changed by a crash, new ghost resources along with new ghost names are introduced after a crash. The universal quantifier is over any such new record of new ghost names. However, the new resources are, to some extent, related to the old resources. The relationship is represented by the *exchange resource* R, which makes it possible to exchange old resources (valid before the crash) into new resources (valid after the crash). This works through rules of the form $P_{old} * R(\vec{\gamma}_{old}, \vec{\gamma}_{new}) \mathrel{-\!\!*} P_{new} * R(\vec{\gamma}_{old}, \vec{\gamma}_{new})$. Here P_{old} could be a points-to predicate before the crash and P_{new} would then be an updated points-to predicate corresponding to the physical state after the crash. When proving soundness of a rule such as PC-NA-MAPSTO we then use the exchange resource to acquire the updated points-to predicate. Note that as R appears in the conclusion, it can perform these exchanges without being consumed itself. This is necessary to prove rules such as MOD-SEP for the post-crash modality. The definition of R is rather extensive as it must allow for resource exchanges for *all* the various resources used in the model. Establishing R is done in the soundness proof of HTR-IDEMPOTENCE. This rule is given an assumption involving a post-crash modality, and to extract the resource under it, R must be procured.

4.8 Case Studies

In order to demonstrate the usefulness of our logic we have used it to verify several case studies.

4.8.1 Durable Message Passing

We have seen how to verify the message passing example Figure 4.2a and the asynchronous fence example in Figure 4.2c. We now show how to verify the durable message passing example from Figure 4.2b. We include recovery code, the safety of which depends on the property that the example satisfies. The recovery code is seen in the proof outline in Figure 4.15.

For all locations we pick the set of abstract states $\{0, 1\}$ and we choose the ψ in the invariant to be the identify function. The invariants are:

$$\begin{aligned}
\phi_x(n, v) &\triangleq n = v \\
\phi_y(n, v) &\triangleq (n = v = 0 * \mathsf{tok}) \vee (n = v = 1 * x \succsim_{\mathsf{f}} 1) \\
\phi_z(n, v) &\triangleq (n = v = 0) \vee (n = v = 1 * x \succsim_{\mathsf{f}} 1)
\end{aligned}$$

$\{x \hookrightarrow_{\mathsf{na}} [0] * y \hookrightarrow_{\mathsf{at}} 0 * z \hookrightarrow_{\mathsf{na}} 0 * x \succsim_{\mathsf{p}} 0 * z \succsim_{\mathsf{p}} 0\}$

$\{x \hookrightarrow_{\mathsf{na}} [0] * y \hookrightarrow_{\mathsf{at}} 0\}$
$x :=_{\mathsf{na}} 1;$
$\{x \hookrightarrow_{\mathsf{na}} [0,1]\}$
flush x;
$\{x \hookrightarrow_{\mathsf{na}} [0,1] * \langle \mathsf{PF} \rangle\, x \succsim_{\mathsf{f}} 1\}$
fence;
$\{x \hookrightarrow_{\mathsf{na}} [0,1] * x \succsim_{\mathsf{f}} 1\}$
$y :=_{\mathsf{at}} 1$
$\{x \hookrightarrow_{\mathsf{na}} [0,1] * y \hookrightarrow_{\mathsf{at}} 1\}$

$\{y \hookrightarrow_{\mathsf{at}} 0 * z \hookrightarrow_{\mathsf{na}} 0\}$
if $!_{\mathsf{at}}\, y = 1$
then
  $\{z \hookrightarrow_{\mathsf{na}} 0 * \langle \mathsf{PF} \rangle\, x \succsim_{\mathsf{f}} 1\}$
  fence;
  $\{z \hookrightarrow_{\mathsf{na}} 0 * x \succsim_{\mathsf{f}} 1\}$
  $z :=_{\mathsf{na}} 1$
  $\{\mathsf{True}\}$
else
  $\{\mathsf{True}\}\ ()\ \{\mathsf{True}\}$

$\{x \hookrightarrow_{\mathsf{at}} n * z \hookrightarrow_{\mathsf{na}} m\}$
if $!_{\mathsf{na}}\, z = 1$
then
  $\{x \hookrightarrow_{\mathsf{at}} n * z \hookrightarrow_{\mathsf{na}} 1 * x \succsim_{\mathsf{f}} 1\}$
  $\{x \hookrightarrow_{\mathsf{at}} 1 * z \hookrightarrow_{\mathsf{na}} 1\}$
  assert $!_{\mathsf{na}}\, x = 1$
  $\{\mathsf{True}\}$
else
  $\{\mathsf{True}\}\ ()\ \{\mathsf{True}\}$

Figure 4.15: Proof outline for the durable message passing example. The code for normal execution is at the top and the recovery code at the bottom.

Here “tok" is an exclusive token implemented using standard Iris ghost state. By exclusive we mean that “tok $*$ tok” is a contradiction. We choose the following crash condition:

$$\langle \mathsf{PC} \rangle\, \exists n, m.\, x \succsim_{\mathsf{p}} n * x \hookrightarrow_{\mathsf{na}} n * z \succsim_{\mathsf{p}} m * z \hookrightarrow_{\mathsf{na}} m$$

The crash condition does not mention y as the recovery code does not use y. The crash condition is easy to show at every step. Since the example involves two threads we need to mention that, using features derived from Perennial, it is possible to split a crash condition between threads where each thread is responsible for maintaining its part. This means that we can use standard concurrent-separation-logic-style thread-local reasoning and prove each of the two threads separately.

The key idea of the proof that we want to highlight is at the write $y :=_{\mathsf{at}} 1$ in the left thread. Here we use HT-AT-WRITE where σ is 0 and σ_t is 1. Showing the invariant is trivial, the tricky part is the last conjunct in the precondition of HT-AT-WRITE, namely that the written state fits in the ordered history of states. To do this we assume some σ_c that is either 0 or 1 (here we crucially use that the abstract state contains only 0 and 1). The latter case is trivial, so suppose $\sigma_c = 0$. In the first case we have the invariant for 0 twice. Since $\phi(0, v)$ contains an exclusive token this is a contradiction. The use of the exclusive token for the state 0 in ϕ_y ensures that the abstract history can only contain 0 once. Hence, when writing the state 1 we can rule out the case where another thread writes 0 that ends up succeeding our write of 1 (which would violate the order of the abstract history). Notice how the argument is modular, we do not assume any knowledge of any other threads, only that the invariant is respected.

We remark that Bila et al. [Bil+22] verified a variant of the durable message passing example where the flush and fence for x is moved from the left to the right thread as in Figure 4.2d. We have verified this variant as well in Coq and in this example the ψ function in the invariant for y is *not* the identity function as previously mentioned. Since the message is sent before anything is flushed the information in the message is lost at a crash, and hence the state needs to change at a crash. For the full details of this example see our Coq mechanization.

4.8.2 Read-Optimized Reference

To show how Spirea supports modular specifications, we give in Figure 4.16 a specification of a library implementing what we call *read-optimized references*. This module implements an interface that appears to clients as a single reference that can be read and writen. The implementation however optimizes the performance of reads. It does this by storing the content of the reference redundantly both in a "volatile" location (one can imagine it being stored in faster volatile memory) and in a persistent location (in the slightly slower persistent memory). When a client writes to the read-optimized reference the value it is saved to both locations, but when reading only the volatile reference is consulted for improved performance.

In the specification, an abstract (existentially quantified) predicate $\mathsf{isRR}(vr, v)$ is used to abstract over (hide from clients) the concrete data representation used by the library implementation; intuitively, it means that the value vr is a read optimized value with value v. After a crash, the volatile location might be lost and hence the reference needs to be recovered before it can be used after a crash. The abstract predicate $\mathsf{recRR}(vr, v)$ intuitively means that vr needs recovery. Just like in the verification of the flush and fence example we choose a crash condition that directly contains the post-crash modality. This simplifies the specification, in particular, in the crash condition for `write`. During the execution of `write`, after updating the volatile location but before updating the persistent location, the read-optimized reference is in an inconsistent state where it satisfies neither isRR for the old value nor the new value. Instead of trying to express this intermediate state we give the

$$
\begin{array}{l}
\texttt{init} \triangleq \lambda \mathsf{v}. \\
\quad \textbf{let}\ \mathsf{per} = \textbf{ref}_{\mathsf{na}}\ \mathsf{v}\ \textbf{in} \\
\quad \textbf{flush}\ \mathsf{per}; \textbf{fence}_{\textbf{sync}}; \\
\quad \textbf{let}\ \mathsf{vol} = \textbf{ref}_{\mathsf{na}}\ \mathsf{v}\ \textbf{in} \\
\quad (\mathsf{per}, \mathsf{vol})
\end{array}
\qquad
\begin{array}{l}
\texttt{read} \triangleq \lambda \mathsf{vr}.\ !_{\mathsf{na}}(\pi_2\ \mathsf{vr}) \\
\texttt{write} \triangleq \lambda \mathsf{vr}, \mathsf{v}. \\
\quad (\pi_1\ \mathsf{vr}) :=_{\mathsf{na}} \mathsf{v}; \\
\quad \textbf{flush}\ (\pi_1\ \mathsf{vr}); \textbf{fence}_{\textbf{sync}}; \\
\quad (\pi_2\ \mathsf{vr}) :=_{\mathsf{na}} \mathsf{v}
\end{array}
\qquad
\begin{array}{l}
\texttt{recover} \triangleq \lambda \mathsf{vr}. \\
\quad \textbf{let}\ \mathsf{per} = \pi_1\ \mathsf{vr}\ \textbf{in} \\
\quad \textbf{let}\ \mathsf{vol} = \textbf{ref}_{\mathsf{na}}\ (!_{\mathsf{na}}\ \mathsf{per})\ \textbf{in} \\
\quad (\mathsf{per}, \mathsf{vol})
\end{array}
$$

$$\{\textbf{true}\}\ \texttt{init}\ v\ \{vr.\ \mathsf{isRR}(vr, v)\}\{\textbf{true}\}$$

$$\{\mathsf{isRR}(vr, v)\}\ \texttt{read}\ vr\ \{w.\ v = w * \mathsf{isRR}(vr, v)\}\{\langle \mathsf{PC} \rangle\ \mathsf{recRR}(vr, v)\}$$

$$\{\mathsf{isRR}(vr, v)\}\ \texttt{write}\ vr\ w\ \{u.\ \mathsf{isRR}(vr, w)\}\{\langle \mathsf{PC} \rangle\ \exists u \in \{v, w\}.\ \mathsf{recRR}(vr, u)\}$$

$$\{\mathsf{recRR}(vr, v)\}\ \texttt{recover}\ vr\ \{vr'.\ \mathsf{isRR}(vr', v)\}\{\langle \mathsf{PC} \rangle\ \mathsf{recRR}(vr, v)\}$$

$$\mathsf{isRR}(vr, v) \vdash \langle \mathsf{PC} \rangle\ \mathsf{recRR}(vr, v)$$

Figure 4.16: Implementation and specification of the read-optimized reference

client what they actually need: the information that after a crash the read-optimized reference is recoverable in either the old or the new state.

Our Coq mechanization contains the full proof of the specification.

4.8.3 Atomic Persists

Raad et al. [RLV20] used the POG logic to verify an example where one thread writes to two locations, flushes and fences the writes, and transfers the information to a second thread through a spin lock. They call this example the *atomic persists* example. The program implemented in λ_{pmem} can be seen in the proof outline in Figure 4.17. The two threads use a shared lock that they both attempt to acquire. When the left thread acquires the lock it writes **true** to the locations x and y. It then flushes both locations and carries out a fence before it releases the lock. When the right thread acquires the lock it reads x and if it reads **true** it writes **true** to z.

Due to the limitations of the **CAS** rule in Spirea we can not verify the spin lock in Spirea. Instead, we verify the spin lock in BaseSpirea but give it a specification inside Spirea. We give the lock a crash-aware lock specification, similar to the one found in Perennial [Cha22, Chapter 3]. With the lock verified in BaseSpirea we can then verify the rest of the example purely in Spirea. This demonstrates both how to use BaseSpirea in combination with Spirea and modularity. In the proof given by Raad et al. [RLV20] the lock and the clients are verified together using one global invariant that contains knowledge about the locations used both internally in the lock and in the two clients. Hence, if the lock implementation is changed, the entire proof is affected. In our proof the lock is given a modular specification and a change in the lock implementation will only affect this proof and not the verification of the clients.

$$\left\{\mathsf{isLock}(\mathsf{lk}, P_{lk}, P_{c,lk}) * z \hookrightarrow_{\mathsf{na}} [\mathsf{False}] * z \succsim_{\mathsf{p}} \mathsf{False}\right\}$$

$\{\mathsf{True}\}$
`acquire` lk
$\left\{\begin{array}{l} x \succsim_{\mathsf{p}} \mathsf{False} * x \succsim_{\mathsf{f}} b * x \hookrightarrow_{\mathsf{na}} \vec{\sigma} +\!\!+ [b] * \\ y \succsim_{\mathsf{p}} \mathsf{False} * y \succsim_{\mathsf{f}} b * y \hookrightarrow_{\mathsf{na}} \vec{\sigma} +\!\!+ [b] \end{array}\right\}$
$x :=_{\mathsf{na}} \mathbf{true}$;
$\{x \hookrightarrow_{\mathsf{na}} \sigma +\!\!+ [b, \mathbf{true}]\}$
$y :=_{\mathsf{na}} \mathbf{true}$;
$\{y \hookrightarrow_{\mathsf{na}} \sigma +\!\!+ [b, \mathbf{true}]\}$
flush x;
$\{\langle \mathsf{PF} \rangle\, x \succsim_{\mathsf{f}} \mathbf{true}\}$
flush y;
$\{\langle \mathsf{PF} \rangle (x \succsim_{\mathsf{f}} \mathbf{true} * y \succsim_{\mathsf{f}} \mathbf{true})\}$
fence;
$\{x \succsim_{\mathsf{f}} \mathbf{true} * y \succsim_{\mathsf{f}} \mathbf{true}\}$
`release` lk
$\{\mathsf{True}\}$

$\{z \hookrightarrow_{\mathsf{na}} [\mathsf{False}]\}$
`acquire` lk
$\left\{\begin{array}{l} x \succsim_{\mathsf{p}} \mathsf{False} * x \succsim_{\mathsf{f}} b * x \hookrightarrow_{\mathsf{na}} \vec{\sigma} +\!\!+ [b] * \\ y \succsim_{\mathsf{p}} \mathsf{False} * y \succsim_{\mathsf{f}} b * y \hookrightarrow_{\mathsf{na}} \vec{\sigma} +\!\!+ [b] \end{array}\right\}$
if $!_{\mathsf{na}}\, x = \mathbf{true}$
then
  $\{b = \mathbf{true}\}$
  $\left\{\begin{array}{l} x \succsim_{\mathsf{f}} \mathbf{true} * x \hookrightarrow_{\mathsf{na}} \vec{\sigma} +\!\!+ [\mathbf{true}] * \\ y \succsim_{\mathsf{f}} \mathbf{true} * y \hookrightarrow_{\mathsf{na}} \vec{\sigma} +\!\!+ [\mathbf{true}] \end{array}\right\}$
  $z :=_{\mathsf{na}} 1$
  $\{\mathsf{True}\}$
else $()$
$\{\mathsf{True}\}$
`release` lk
$\{\mathsf{True}\}$

$\left\{\begin{array}{l} x \succsim_{\mathsf{p}} b_x * x \hookrightarrow_{\mathsf{na}} \vec{\sigma} +\!\!+ [b_x] * \\ y \succsim_{\mathsf{p}} b_y * y \hookrightarrow_{\mathsf{na}} \vec{\sigma} +\!\!+ [b_y] \\ z \succsim_{\mathsf{p}} b_z * z \hookrightarrow_{\mathsf{na}} \vec{\sigma} +\!\!+ [b_z] \end{array}\right\}$
if $!_{\mathsf{na}}\, z = \mathbf{true}$
then
  $\{x \succsim_{\mathsf{f}} \mathbf{true} * y \succsim_{\mathsf{f}} \mathbf{true}\}$
  $\left\{\begin{array}{l} x \hookrightarrow_{\mathsf{na}} \vec{\sigma} +\!\!+ [\mathbf{true}] * \\ y \hookrightarrow_{\mathsf{na}} \vec{\sigma} +\!\!+ [\mathbf{true}] \end{array}\right\}$
  assert $!_{\mathsf{na}}\, x = \mathbf{true}$
  assert $!_{\mathsf{na}}\, y = \mathbf{true}$
  $\{\mathsf{True}\}$
else
  $\{\mathsf{True}\}\ ()\ \{\mathsf{True}\}$

Figure 4.17: Proof outline for the atomic persists example. At the top is the pre-crash code and below the recovery code

We now explain the proof of the atomic persist example in more detail. The proof also appears in our Coq mechanization. The property that we want to show is that the write to z must be ordered after the two writes to x and y. In other words if the right thread reads **true** from x it must also be the case that it would read **true** from y if it where to do so. So, due to the use of the lock the two separate writes performed by the left thread appear as one atomic write to the right thread, *i.e.,* the right thread either sees none of the writes or all of the writes.

For the locations x and y we use a simple invariant with an abstract state of booleans $\{\textbf{true}, \mathsf{False}\}$ and $\phi_b(\sigma, b) \triangleq \sigma = b$ as the invariant.

To verify the example we prove a crash-aware lock specification for a (volatile) lock using BaseSpirea. The specification that we show for the lock is identical to the crash-aware lock specification proposed by Chajed [Cha22, Chapter 3] (note that while the specification is the same our proof is different as it has to account for weak persistent memory whereas Perennial's lock assumes sequentially consistent memory). Since our specification is identical to their we do not repeat it here. The key point that is relevant to our present goal is that the assertion for the lock has the form $\mathsf{isLock}(v, P, P_c)$ and means that the lock protects both a resource P, as the standard CSL lock specification, and a *crash-resource* P_c, which essentially is a crash condition that the lock guarantees to preserve.

We want the lock to own the points-to predicates for x and y. Furthermore, the resource should state that x and y have the same last state and that that state has been flushed. Finally, they should be persisted in at least the initial state.

$$\begin{aligned} P_{lock} \triangleq \exists \vec{\sigma}, b.\, x \succsim_{\mathsf{p}} \mathsf{False} * x \succsim_{\mathsf{f}} b * x \hookrightarrow_{\mathsf{na}} \vec{\sigma} +\!\!+ [b] * \\ y \succsim_{\mathsf{p}} \mathsf{False} * y \succsim_{\mathsf{f}} b * y \hookrightarrow_{\mathsf{na}} \vec{\sigma} +\!\!+ [b] \end{aligned}$$

For the lock's crash-resource we only need that the locations have been persisted in some state and then the points-to predicates ending in that state.

$$\begin{aligned} P_{c,lock} \triangleq\ & \langle \mathsf{PC} \rangle\, \exists \vec{\sigma}_x, \vec{\sigma}_y, b_y, b_x. \\ & x \succsim_{\mathsf{p}} b_x * x \hookrightarrow_{\mathsf{na}} \vec{\sigma} +\!\!+ [b_x] * \\ & y \succsim_{\mathsf{p}} b_y * y \hookrightarrow_{\mathsf{na}} \vec{\sigma} +\!\!+ [b_y] \end{aligned}$$

For the location z we use the abstract state of booleans and the invariant:

$$\begin{aligned} \phi_z(\mathsf{False}, v) &\triangleq v = \mathsf{False} \\ \phi_z(\textbf{true}, v) &\triangleq v = \textbf{true} * x \succsim_{\mathsf{f}} \textbf{true} * y \succsim_{\mathsf{f}} \textbf{true} \end{aligned}$$

The key aspect here is that when z has the value **true** then the invariant contains the fact that x and y has been flushed in the state **true**.

In addition to the crash resource for the lock we also need a crash condition that ensures that z is available after a crash:

$$P_c \triangleq \langle \mathsf{PC} \rangle\, \exists \vec{\sigma}, b.\, z \succsim_{\mathsf{p}} b * z \hookrightarrow_{\mathsf{na}} \vec{\sigma} +\!\!+ [b]$$

```
makeStack ≜ λ_.
  let node = ref_na nil in
  flush node;
  fence;
  ref_at node
sync ≜ λtoHead.
  flush toHead;
  fence_sync;
nil ≜ inj_1 ()
cons v toNext ≜ inj_2 (v, toNext)
```

```
pop ≜ rec loop toHead =
  let head = !_at toHead in
  fence;
  match !_na head with
    inj_1 _ ⇒ inj_1 ()
    inj_2 pair ⇒
      let next = !_na(π_2 pair) in
      if CAS toHead head next
      then inj_2 (π_1 pair)
      else loop toHead
```

```
push ≜ λtoHead, val.
  let toNext = ref_na () in
  let newNode =
    ref_na (cons val toNext) in
  flush newNode;
  (rec loop () =
    let head = !_at toHead in
    toNext :=_na head;
    flush toNext; fence;
    if CAS toHead head newNode
    then () else loop ()) ()
```

Figure 4.18: Implementation of the durable Treiber stack

The entire crash condition for the two threads is then: $P_{c,lock} * P_c$. When the lock is acquired threads are required to maintain $P_{c,lock}$ and the P_c part we let the right thread maintain.

With this setup in place the proof outline appears in Figure 4.17. Note that to keep the outline simple we do not repeat resources that are unchanged in between lines in the program.

4.8.4 Durable Data-Structures With Null-Recovery

Concurrent *non-blocking* data structures have the property that they can be made durable and crash-safe by appropriately inserting flushes and fences [Fri+20; IMS16]. They furthermore enjoy *null-recovery*. As mentioned, this is the property that no recovery code is needed after a crash to restore the consistency of the data structure.

$$\{\mathbf{true}\}\,\texttt{makeStack}\,()\,\{\ell.\,\mathsf{isStack}(\ell,\phi)\} \qquad \{\mathsf{isStack}(\ell,\phi) * \phi(w)\}\,\texttt{push}\,\ell\,w\,\{\mathbf{true}\}$$

$$\{\mathsf{isStack}(\ell,\phi)\}\,\texttt{pop}\,\ell\,\{v.\,v = \mathbf{inj}_1\,() \vee \exists x.\,v = \mathbf{inj}_2\,x * \phi(x)\}$$

$$\{\mathsf{isStack}(\ell,\phi)\}\,\texttt{sync}\,\ell\,\{\mathsf{synced}(\ell)\} \qquad \mathsf{isStack}(\ell,\phi) \mathrel{-\!\!*} \langle\mathsf{PCF}\rangle\,\mathsf{isStack}(\ell,\phi)$$

$$\mathsf{isStack}(\ell,\phi) * \mathsf{synced}(\ell) \mathrel{-\!\!*} \langle\mathsf{PC}\rangle\,\mathsf{isStack}(\ell,\phi)$$

Figure 4.19: Specification of the durable Treiber stack

Data structures with this property are *by construction* always in a consistent state—even after a crash. This makes them particularly well suited in a persistent setting and easier to use as clients of such data structures do not need to carry out recovery procedures (in contrast to, for instance, the read optimized reference). One would therefore hope to be able to derive similarly easy to use CSL specifications for such data structures. In this section we show how this is the possible in Spirea and explain how to specify and verify safety (including thread-safety and crash-safety) of non-blocking data structures with null-recovery. In our Coq mechanization we have verified durable implementations of both the Treiber stack and the Michael-Scott queue. These case studies show that our crash-aware invariants are sufficiently expressive to capture representation predicates for durable concurrent data structures and capable of handling null-recovery.

For space reasons we cover only the Treiber stack in this section. We focus on the resulting specification and sketch the proof. The full verification of both examples appears in our mechanization.

Implementation

The Treiber stack consists of a pointer to a linked list where, for thread-safety, the pointer is updated with **CAS**. The implementation of the stack appears in Figure 4.18. We use pointers to sums to represent nodes in the linked list: $\mathbf{inj}_1\ ()$ represents a nil-node and $\mathbf{inj}_2\ (v, \ell)$ represents a cons-node with value v and with ℓ pointing to the succeeding node. In order to make the stack crash-safe we have inserted flushes and fences appropriately.

Our implementation is *buffered durable linearizable*, which means that it never waits (with $\mathbf{fence}_{\mathbf{sync}}$) for an operation to reach persistent memory, but only ensures (with **fence**) that operations persist in the order in which they linearize. This improves performance but means that at a crash some returned operations might be lost. As is common for such data structures we include a `sync` operation that explicitly makes sure that the stack is persisted by using $\mathbf{fence}_{\mathbf{sync}}$.

Specification

The specification (in Figure 4.19) enforces that a predicate $\phi : \textsc{Val} \to \mathsf{dProp}$ holds for each item in the stack. The specifications make use of an abstract (existentially quantified) representation predicate isStack, which is persistent, in the Iris sense, and hence duplicable, so that several threads can access the stack concurrently. Since isStack is persistent it does not need to appear in crash-conditions and hence we can use normal Hoare triples instead of crash Hoare triples. As such, the non-highlighted part forms a completely typical per-item CSL specification for a concurrent stack. This is exactly what we want, as it implies that a client can use the durable stack as they would a normal stack. Note that our specification does not imply linearizability or the LIFO property of the stack, but it does imply thread-safety and crash-safety.

The three highlighted rules are specific for persistent memory. The first of these shows that by running `sync` ℓ one gets the resource $\mathsf{synced}(\ell)$ which is evidence that the stack has been persisted. The two last rules concern the interaction between isStack and the post-crash modalities. The first rule states that if ℓ is a stack before a crash then after a crash it is still a stack, but only under the $\langle \mathsf{PCF} \rangle$ modality since the stack is buffered. The second rule applies if the stack is certain to have been persisted, as witnessed by synced; in this case the stack is preserved under the $\langle \mathsf{PC} \rangle$ modality.

The last two rules capture not only crash-safety but also the null-recovery property of the stack. They imply that with no recovery code needed, the isStack representation predicate can be reclaimed after a crash, and thus that a client can safely keep using the stack after a crash.

For the specification to be sound in our weak persistent memory setting, the per-item predicate ϕ must satisfy that for all $v \in \textsc{Val}$ it is the case that (1) $\phi(v) \vdash \langle \mathsf{NB} \rangle\, \phi(v)$, (2) $\phi(v) \vdash \langle \mathsf{PCF} \rangle\, \phi(v)$, and (3) $\phi(v) \vdash \Box\, \phi(v)$. The first two requirements are necessary to make ϕ safe to transfer between threads and across crashes. The third requirement expresses that ϕ must be persistent (in the Iris sense). This is required for a subtle reason: Since the stack is buffered, operations might return before they persist. Therefore, a value v can be popped from the stack (at which point the client is given $\phi(v)$), and then a crash can happen before the changes by the pop persist. Then, after the crash, v is still present in the stack, and thus it can be popped again (at which point the client is given $\phi(v)$ once more). In summary, due to crashes, the same value can be popped several times and hence the resource must be duplicable, *i.e.,* persistent. This requirement holds, for instance, for simple properties such as v being an even number and for assertions about atomic locations. Had the implementation been non-buffered, *i.e.,* implemented using the synchronous fence, then this requirement could be removed.

Proof (sketch)

The proof proceeds by defining the predicates synced and isStack and then verifying that the specifications hold. The definition of synced expresses that ℓ has been

$$\mathsf{synced}(\ell) \triangleq \ell \succsim_{\mathsf{p}} \star$$

$$\mathsf{isStack}(\ell, \phi) \triangleq \boxed{\ell \mid \pi_{stack}(\phi)} * \ell \hookrightarrow_{\mathsf{at}} \star$$

$$\phi_{stack}(\phi)(\_, v) \triangleq \exists \ell_h, xs \in \textsc{List}(\textsc{Val}).\, v = \ell_h * \mathop{\Large *}_{x \in xs} \phi(x) * \mathsf{isNode}(\ell, xs)$$

$$\mathsf{isNode}(\ell_{node}, \quad []) \triangleq \exists q.\, \boxed{\ell_{node} \mid \mathbf{inj}_1\ ()} * \ell_{node} \hookrightarrow^{q}_{\mathsf{na}} \star * \ell_{node} \succsim_{\mathsf{f}} \star$$

$$\mathsf{isNode}(\ell_{node}, x :: xs) \triangleq \exists \ell_{toNext}, \ell_{next}, q_1, q_2, \vec{\sigma}, i.\, \boxed{\ell_{node} \mid \mathbf{inj}_2\ (x, \ell_{toNext})} * \ell_{node} \hookrightarrow^{q_1}_{\mathsf{na}} \star * \ell_{node} \succsim_{\mathsf{f}} \star * \boxed{\ell_{toNext} \mid \pi_{toNext}} * \ell_{toNext} \hookrightarrow^{q_2}_{\mathsf{na}} \vec{\sigma}(i, \ell_{next}) * \ell_{toNext} \succsim_{\mathsf{f}} (i, \ell_{next}) * \mathsf{isNode}(\ell_{next}, xs)$$

Figure 4.20: Invariants and definitions used in the proof of durable concurrent stack

persisted. For isStack we use three invariants. In all three the ψ function is the identity function. Two of the invariants use the abstract state set $1 = \{\star\}$. Elements of this abstract state carry no information, but lower bounds are still meaningful, *e.g.*,$\ell \succsim_{\mathsf{f}} \star$ means that location ℓ has certainly been flushed.

For a node the pointer to the sum never changes. For these locations we use the *constant invariant*. Given a value v the constant invariant $\pi_{const}(v)$ has the abstract state 1 and the invariant $\phi_{const}(\_, v') \triangleq v = v'$. We use the notation $\boxed{\ell \mid v}$ for $\boxed{\ell \mid \pi_{const}(v)}$.

The pointer from a cons-node to its successor potentially changes many times in `push` if the **CAS** in `push` fails. For this location we use the invariant π_{toNext}. Its abstract state is $\mathbb{N} \times \textsc{Val}$ ordered by the natural numbers in the first component. The invariant is $\phi_{toNext} = \lambda(n, v), v'.\, v = v'$.

For the stack itself (the pointer to the head of the linked list) we use the invariant $\pi_{stack}(\phi)$. Its abstract state is 1 and the invariant $\phi_{stack}(\phi)$ appears in Figure 4.20. It states that there exists a logic-level list xs, all of whose elements satisfy ϕ, and it uses isNode to recursively express that the structure of the linked list corresponds to xs.

With these definitions and invariants in place the proof that the code satisfies the specification is fairly straightforward; see our Coq mechanization for the details.

We finally remark that a similar (non-persistent, non-weak memory) concurrent stack can be verified in standard Iris [BB20]. The Iris proof uses an Iris invariant to define the isStack representation predicate.

4.9 Related and Future Work

We now discuss aspects of related work that have not already been treated in the paper.

Logics for Persistent Memory To our knowledge, there are only two prior program logics for persistent memory, namely Persistent Owicky-Gries (POG) [RLV20]

and Pierogi [Bil+22]. Both POG and Pierogi focus on the persistent memory model of the x86 architecture [RLV20], which is stronger, both in terms of weak and persistent memory, than our memory model, which does not include details specific for any one architecture; instead it is a slight generalization of the persistent memory models in x86 and ARM. The programming languages covered by POG and Pierogi are much simpler than ours, λ_{pmem}; the languages in *op. cit.* support only a static number of threads running sequential commands, and a static number of memory locations. In contrast, λ_{pmem} includes more high-level features such as higher-order functions and dynamic allocation of threads and locations.

Both POG and Pierogi are Owicki-Gries-style program logics. POG makes use of rely-guarantee style reasoning to support composition of threads that do not interfere, whereas Pierogi does not support thread-local reasoning. In contrast, Spirea is a separation logic and hence it supports frame rules and thread-local reasoning. Moreover, since Spirea is built on top of Iris, it includes advanced features such as user-defineable ghost state and higher-order quantification, which are not present in POG or Pierogi but which are important for modular specification and verification of libraries, such as the stack case study we considered in Section 4.8.4. From Perennial we gain the ability to reason about durable resources in a convenient fashion using normal separation logic ownership.

In contrast to POG but similarly to Pierogi, our Spirea logic is mechanized in a proof assistant. Pierogi has been mechanized in Isabelle/HOL and its authors report that the Sledgehammer tool can be used to search automatically for program proof rules to apply. In contrast, we make use of the Iris Proof Mode [KTB17c] to support interactive development of program proofs in the Coq style, which works well for our higher-order logic and larger examples.

Similarly to Pierogi, Spirea supports reasoning directly about optimized flushes (write-backs) (**flush**) and the use of fences. In contrast, POG only supports reasoning about a stronger operation that combines the write back and the fence. To handle other programs they instead offer a translation that in some cases can translate a program with the weaker, and more tricky to reason about, instructions into equivalent programs. This translation only works for programs that use these instructions in a certain pattern, and programs that do not adhere to this pattern can not be reasoned about using their logic. Since we handle these operations directly we can verify such programs.

Finally, POG and Pierogi have, to the best of our knowledge, only been applied to reason about very small programs consisting of only a few lines, whereas we have used Spirea to verify larger programs, in particular entire data structures. Additionally we have shown how to give such data structures modular specifications as extensions of traditional CSL specifications.

Separation Logic for Weak Memory GPS [TVD14] is a program logic for the release-acquire and non-atomic fragment of the C11 weak memory model. The logic introduced *protocols* to reason about atomic location, the inspiration for our

crash-aware invariants. GPS does not use protocols for non-atomic locations, but instead a standard points-to predicate. As mentioned, this approach is not sufficient in a persistent setting. The CAS rule in GPS does not require that (what we call) the invariant for the read value is preserved. When reading v_1 and simultaneously writing v_2 with a CAS, the CAS rules in GPS allows one to use the invariant for v_1 to show the invariant for v_2 and keep any additional resources *without* reestablishing the invariant for v_1. This is sound because the C11 semantics ensures that no CAS operation will ever read v_1 again. While this is also the case in our semantics, after a crash, the write for v_1 might have been persisted while the write for v_2 has not been persisted. Then another CAS operation might read v_1 again. Hence, in the presence of crashes the GPS CAS rule is unsound. Our rule for CAS requires that the invariant still holds for the read value, ensuring that the invariant always holds for all writes. This is sound even with crashes but is significantly more limiting than the GPS CAS rule. Essentially, the GPS CAS rule is sound for transferring resources between concurrently running CAS-operations, but not across crashes. Our CAS rule is sound for transferring resources across crashes, but only in a limited way between concurrent CAS'es. Creating a CSL rule that is simultaneously sound for both is very challenging and something that we would like to explore in future work. The CAS rule in BaseSpirea does not suffer from this limitation, and as demonstrated in Section 4.8.3, it can be used together with Spirea for cases where a stronger CAS rule would otherwise be needed.

The read rule for atomic locations in GPS does not make it possible to transfer exclusive resources out of the invariant for the value read. Our read rule makes it possible to extract exclusive resources as long as the invariant still holds (for instance by transferring other resources into the invariant). We make use of this capability to verify the message passing examples. In GPS an additional feature, *escrows*, is needed to verify the message passing examples.

Our use of modalities to reason about fences is inspired by Fenced Separation Logic (FSL), a program logic that supports reasoning about the release and acquire memory fences in the C11 memory model [DV16]. FSL includes two fence modalities to describe resources that have been prepared for release or acquire by a release or acquire fence. The release and acquire fences in C11 serve a different purpose than those in λ_{pmem} and the modalities in FLS are correspondingly different as well.

Recently, in the context of weak memory we have seen logics that support specifications that go beyond safety. Compass [Dan+22] and Cosmos [MJ21b] are both capable of showing stronger correctness results by using *logically atomic triples* as specifications. In contrast, our specification for the durable stack only implies safety. We think it would be interesting to investigate how ideas from these logics apply in our setting and we believe that a stronger CAS rule (per the discussion above) is necessary to achieve this.

Separation Logics for Durable Storage Crash Hoare Logic [Che+16] and the more advanced Perennial [Cha22; Cha+19; Cha+21] are separation logics capable

of verifying crash-safety. In contrast to our work, Crash Hoare Logic and prior work using Perennial has only considered sequentially consistent memory and synchronously persisting writes without any weak behavior. When writes persist synchronously/atomically the content of durable storage is always in a single certain state. Therefore, rules for the post-crash modality include no non-determinism and are simpler "either/or" rules where some (volatile) resources are entirely lost at a crash and other (non-volatile) resources are preserved unchanged after a crash. In our setting, since the crash step is non-deterministic, the rules for the post-crash modality are significantly more involved. Consider for instance a rule such as PC-NA-MAPSTO which illustrates that the post-crash modality both introduces non-determinism (the quantified i), potentially takes resources away (represented both by $\langle \mathsf{ifRec} \rangle$ and the lost states), and potentially adds new resources (the crashedIn(ℓ, σ_i)).

Persistency Models While our focus in this paper is on the logic, we remark on related work on persistency models. As mentioned, persistency models of the x86 and ARM architecture have been formalized [KL21; Raa+20; RWV19]. In parallel with our work, new variants of these that, like our semantics, are based on views have been presented [Cho+21]. It would be interesting to formally verify a correspondence between the explicit epoch persistency model and the x86 and ARM persistency models. We believe that our operational semantics could be used for this purpose. It would also be worthwhile to show an equivalence between our operational model and a model in a declarative or axiomatic style.

Chapter 5

The Nextgen Modality: A Modality for Non-Frame-Preserving Updates in Separation Logic

Abstract

Separation logic is about resources and the way in which resources can soundly change and be updated is fundamental. The way in which resources can change has typically been restricted to certain local or frame-preserving updates. However, recently we have seen separation logics where the restriction to frame-preserving updates seems to be a hindrance towards achieving the ideal program reasoning rules. In this paper we propose a novel *nextgen* modality that enables reasoning across *generations* where each generational change can update resources in ways that are non-local and non-frame-preserving. We implement the idea as an extension to the Iris base logic, which enriches Iris with an entirely new capability: the ability to make non-frame-preserving updates to ghost state. We show that two existing Iris modalities are special cases of the nextgen modality and our "extension" can thus also be seen as a generalization and simplification of the Iris base logic. To explore and demonstrate the utility of the nextgen modality we use it to construct a separation logic for a programming language with stack allocation and with a return operation that clears entire stack frames. The nextgen modality is used to great effect in the reasoning rule for return, where a modular and practical reasoning rule is otherwise out of reach. This is the first separation logic for a high-level programming language with stack allocation. We sketch ideas for future work in other domains where we think the nextgen modality can be useful.

5.1 Introduction

Separation logic is a logic for reasoning about ownership over resources. A crucial aspect of separation logic is the ability to perform what we call updates to resources that are *frame-preserving*. As a quintessential example of a frame-preserving update, consider the separation logic proof rule for assignment to a reference:

$$\{\ell \hookrightarrow v\}\, \ell \leftarrow w\, \{\ell \hookrightarrow w\}.$$

Here, the difference between the points-to assertion in the pre- and postcondition means that the heap resource is *updated*. The update is sound because the points-to assertion implies *exclusive* ownership over the location ℓ. This ensures that changing this part of the heap resource is guaranteed to not interfere with any other assertions. Any other assertion, or frame P, that might exist and that is valid in combination with $\ell \hookrightarrow v$ (meaning that $P * \ell \hookrightarrow v$ is not false) is also valid in combination with $\ell \hookrightarrow w$. This "locality" of the update is required in order for the assignment rule to combine soundly with the *frame rule* for Hoare triples. If we combine the two rules we get

$$\frac{\{\ell \hookrightarrow v\}\, \ell \leftarrow w\, \{\ell \hookrightarrow w\}}{\{\ell \hookrightarrow v * P\}\, \ell \leftarrow w\, \{\ell \hookrightarrow w * P\}}$$

which is only sound due to the property just described.

In the separation logic Iris [Jun+18a; Jun+15b], which we use as our vehicle in this paper, updates to resources that satisfy this property of being sound in combination with the frame rule are called *frame-preserving updates*. Such updates are local in the sense that the changes they allow one to perform are closely related to the resources and knowledge one owns locally. This makes them well-suited for reasoning about programming language features that make similarly local changes to the physical state. For instance, writing to a reference, as above, which effects a small and precisely delineated fragment of the physical state. Iris limits resource updates to those that are frame-preserving in this way. But updates that preserve the frame in this manner are not a natural fit for program execution steps that are less localized and that make sweeping changes to larger parts of the physical state. Examples of such non-local execution steps that can not easily be expressed as frame-preserving updates include:

Crashes in a setting with durable storage If one wishes to verify crash-safety in a setting with durable storage, how the state of a machine changes at a crash can be represented as a *crash-step* in the operational semantics. At such a step many locations might be lost. For instance, *all* locations residing in volatile memory are lost. This makes the crash step non-local. Recently, we have seen several separation logics for reasoning about crashes and durable storage [Cha+17; Cha+19; Cha+21; VB23a]. They have all had to work around the absence of non-frame-preserving updates.

Garbage collection step In a language with garbage collection, for which the operational semantics explicitly models the action of the garbage collector, a step corresponding to garbage collection would reclaim a potentially large number of locations in memory. If one wishes to reason about the garbage collection step, without having gathered all the resources for all the locations that the garbage collector might need to collect, then the update to the logical resources is not frame-preserving. Indeed, in recent work on separation logics in the presence of garbage collection, this has been worked around by having the logics maintain a global account of all the resources that could be reclaimed by the garbage collector, for instance in the form of an explicit correspondence between physical addresses and logical addresses [Gué+23; MP22; MCP23]

Returning from a function in language with stack allocation In a language with a stack and values allocated on the stack, returning from a function invalidates the locations in the entire stack frame corresponding to the function call. This example is explained in greater detail later, as we use it as a case-study in this paper.

One can also imagine cases where the desire to make non-frame-preserving updates arises without stemming from the operational semantics:

Temporary read-only permissions One might wish to, at the level of the logic, make a location read-only temporarily in order to obtain a freely shareable read-only points-to predicate. At some determined point these read-only points-to predicates should disappear and the original read-write points-to predicate should be reoptained. A separation logic with this capability was constructed in [CP17]. Their separation logic was not based on Iris, and the changes to resources that they use are not frame-preserving. The read-only points-to predicate is, naturally, incompatible with the read-write points-to predicate, and since the read-only points-to predicate is freely shareable it might always be contained in some frame, and hence an update that restores the read-write points-to predicate is not frame-preserving. This means that it is not clear how to support Charguéraud and Pottier's reasoning in Iris.

Time based resource revocation In distributed systems a notion of *time* is often used to control revocation of resources. For instance, acquiring a distributed lock might only grant the lock under a certain timeout expressed in terms of wall-clock time. Once this duration of time has passed, the right to use the lock is revoked and the lock should re-obtain the right to release to another peer. However, the lock cannot simply update its internal resources to obtain this right as that would not preserve the frame of the peer whose time expired. The recent Iris based separation logic Grove [Sha+23] allows for reasoning about time in a distributed setting using novel *time-bounded invariants*. Perhaps a mechanism that allows non-frame-preserving updates, justified by changes in time, can also be applied to tackle some of the problems in this space.

We hasten to emphasize that for the above examples it is of course not the case that creating a logic for a particular language or verifying programs with a particular characteristic is impossible without the ability to make non-frame-preserving updates. By using sophisticated resources, advanced invariants, or straight-up workarounds, it is often possible to do without the ability to make non-frame-preserving updates. Rather, it is the case that certain *specific* desirable program rules and verification approaches are not possible since they can not be encoded as frame-preserving updates. One good example of this is the previously mentioned work by Charguéraud and Pottier on temporary read-only permissions. It is not that their approach makes it possible to verify entirely new classes of programs as temporary read-only points-to predicates can also be achieved with the bookkeeping overhead of fractional permissions. Rather, the benefit of their approach is that it is simple and elegant, and to achieve the particular rules in their program logic, non-frame-preserving updates are necessary.

In this paper we present a novel modality that facilitates making changes to resources in ways that are *not* frame-preserving. The modality is called the **nextgen** modality, since it supports reasoning about what happens "in the next generation," after a non-frame-preserving update. The modality makes it possible to change resources as described by any (well-behaved) *transformation* function chosen by the user of the logic. We develop the modality as an extension to Iris. Usually, new features for Iris are developed *within* the logic. But since Iris, at the fundamental level of its *base logic* provides no means for expressing the kind of non-local updates that we are interested in, we have to extend the base logic itself. We remark that the Iris base logic has been relatively stable since 2017 [Kre+17b; Tim+18] (except for experiments with transfinite versions of Iris [Spi+21]) and find it noteworthy that this is one of the instances where the base logic needs changing.

Of course, we can not in a single paper develop entire program logics for all the motivating examples above. Instead, we choose to focus on one of them, namely, as mentioned, a program logic for a language with stack-allocated values. This case study demonstrates both how to use our nextgen modality and is a contribution in its own right. In the program logic, we use the nextgen modality to account for the way in which returns invalidate the call-stack frame. The result is the first separation logic that supports reasoning about a high-level language with stack allocation and where, in the operational semantics, returning from a function invalidates the call-stack frame of the returning function.

Our focus on one case-study naturally means that we do cannot conclude with certainty whether the nextgen modality, in some form, can or cannot be of benefit in all the examples mentioned above. But we are quite certain that at least some other examples than the stack allocation example can benefit from the nextgen modality; see the discussion in the future work section.

In short, the contributions of the paper are as follows:

- We extend the Iris base logic with a new modality, the nextgen modality. This modality makes it possible to make non-frame-preserving changes to

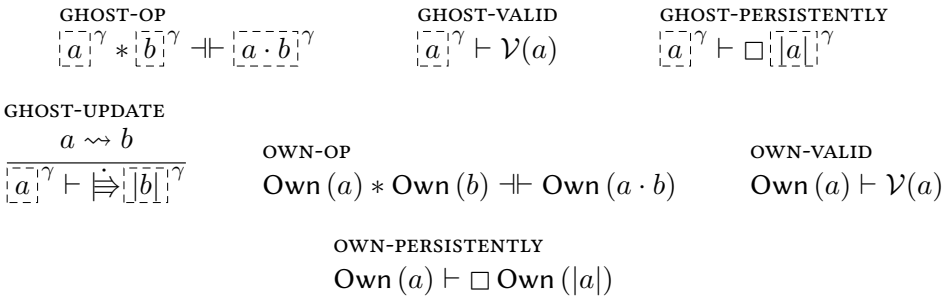


Figure 5.1: A few of the Iris rules related to ghost state.

resources in Iris which was previously not possible. We have extended the Iris implementation in Coq to include the new modality and also adapted the Iris Proof Mode [KTB17c] to include support for the nextgen modality.

- We develop a program logic for a language StackLang. In this language the physical state contains a call-stack and values can be allocated on the stack. Returning from a function clears the call-stack from the returning function from the physical state. By using the nextgen modality in the proof rule for returns we arrive at a proof rule that is simple and easy to use. We have formalized the new program logic and examples using it in Coq in our extended version of the Iris implementation.

The rest of the paper proceeds as follows. In Section 5.2 we give the necessary Iris background to explain our contributions, and we describe the most closely related work. Section 5.3 introduces the *basic* nextgen modality, its rules in the logic, and its model. Section 5.4 describes the operational semantics of StackLang and the program logic we construct for it. In Section 5.5 we compare against related work not covered earlier in the paper and discuss future work.

The Coq development accompanying this chapter is available online on GitHub: `https://github.com/logsem/iris-nextgen`.

5.2 Background and Related Work

We first cover a bit of Iris background and the most closely related work. The background material includes some aspects of Iris that are perhaps not part of the typical Iris user's repertoire of Iris features, but that, nevertheless, are important in order to explain our contributions and situate them in comparison to the related work.

5.2.1 Iris Background

A central feature in Iris is its support for user-defined ghost state. Users of the logic can define and choose their own *resource algebras* (RAs) to capture the behavior of their desired ghost state. With much flexibility one can mix and match RAs and use many of them in the logic. For any RA A and element $a \in A$, the proposition $\boxed{a}^{\gamma}$ asserts ownership over a at some *ghost location* distinguished by a *ghost name* $\gamma \in \textsc{GName}$. We do not recall the full definition of what an RA is, but it includes an associative and commutative monoidal operation, which gives meaning to separating conjunction and ownership, cf. the GHOST-OP bi-entailment in Figure 5.1. As not all combinations are meaningful, a validity predicate $\mathcal{V} : A \to \textsc{Prop}$ identifies the valid elements. The logic maintains the property that only valid elements can be owned, cf. GHOST-VALID. A partial function called the core $|-|$ extracts from elements their *duplicable* part. That is, for every $a \in A$, its core $|a|$ is the duplicable part of a, meaning in particular that $|a| = |a| \cdot |a|$. The persistently modality $\Box$ removes all non-duplicable resources by applying the core operation to all resources, cf. GHOST-PERSISTENTLY. Elements of an RA are ordered w.r.t. an *extension order*: $a \preccurlyeq b \triangleq \exists c.\, a \cdot c = b$. A resource a can be updated to another resource b via a frame-preserving update denoted $a \rightsquigarrow b$ and defined as:

$$a \rightsquigarrow b \triangleq \forall c. \mathcal{V}(a \cdot c) \Rightarrow \mathcal{V}(b \cdot c)$$

This definition matches the intuition we gave in the introduction: a resource can be updated as long as it remains valid in combination with any frame with which it was also valid before. Frame-preserving updates are internalized into the logic through the update modality $\dot{\Rightarrow}$ and the rule GHOST-UPDATE.

Both the ability to use several RAs and the ghost ownership assertion $\boxed{a}^{\gamma}$ are not present in the Iris *base logic*, but is provided by constructions that are *defined* within the base logic. Instead, the Iris base logic is parameterized over just a single "global" RA M and thus, a user of the base logic can in fact pick only a single RA to instantiate the logic with. In the base logic, the assertion $\mathsf{Own}\,(a)$, where $a \in M$, denotes ownership over elements of the single resource RA M.

We now recall the constructions that make it possible to use several RAs and named ghost ownership assertions on top of the base logic. First, one chooses a sequence of all the RAs that are to be used in the logic: $M_1, \ldots, M_n$, where n is the number of RAs. Then, the single global resource algebra M is chosen to be a "resource algebra of resource algebras" in the following way:

$$M \triangleq \prod_{i \in I} \textsc{GName} \xrightarrow{\text{fin}} M_i \tag{5.1}$$

This construction has two levels. The first level is a product indexed by the number of RAs $I = \{1, \ldots, n\}$. This is such that multiple different RAs can be used. The next level is a finite map over ghost names. This is such that multiple independent instances of the same RA can be used. The set M is itself an RA whose operation

simply combines the two layers point-wise. The familiar ghost ownership proposition is now defined in terms of the basic ownership Own assertion:

$$\boxed{a : M_i}^{\gamma} \triangleq \mathsf{Own}\,(\lambda j.\, \text{if } i = j \text{ then } \{\gamma \mapsto a\} \text{ else } \varnothing)$$

We emphasize that the full path to a ghost location consists of both an index $i \in I$ and a ghost name γ. The notation for ownership at ghost locations, however, usually leaves out the i as it can be inferred from the type of the element at the location.

For modularity, proofs carried out in Iris do not specify exactly what the sequence of available RAs should be. Instead, they require that M has the form above, and that indices exist in the sequence of RAs that contain the RAs necessary for the given proof. For instance, if a proof requires an RA A then the proof will simply assume that there exists an $i \in I$ such that $M_i = A$. A proof with such a requirement can modularly be combined with other proofs making similar constraints. Only to obtain a closed "final" proof does one need to fully determine M, and at this point one can do so while ensuring that it contains all the RAs required by sub-proofs.

A program logic constructed inside of Iris usually relies on certain *global ghost names* for ghost locations that contain ghost state used by the program logic itself. We use $\vec{\gamma}$ to refer to such a collection of global ghost names. For instance, an Iris based program logic for a programming language with a heap keeps ghost state for the heap at a global ghost name. In other words, there would be a ghost name inside $\vec{\gamma}$ specifically for the heap ghost state which we could write as $\vec{\gamma}.\mathrm{heap}$. Since points-to assertions are modeled using this ghost state they make use of the global ghost name. Global ghost names are usually left implicit both on paper and in Coq, but we could write a points-to assertion like this

$$\ell \hookrightarrow^{\vec{\gamma}.\mathrm{heap}} v$$

to make explicit the ghost name it makes use of. That is, points-to assertions are in fact parameterized over the global ghost name that they use. Similarly, as the concrete values of the global ghost names do not matter, proofs and the program logic itself are parameterized over the collection of the global ghost names. When proofs are carried out in Coq the global ghost names are assumed as an implicit context parameter. On paper one should imagine that there is an implicit "$\forall\vec{\gamma}$." in the beginning of proofs, making the names $\vec{\gamma}$ always "in scope".

5.2.2 Perennial's Post-Crash Modality

The work most closely related to ours is the *post-crash modality* by Tej Chajed and Joseph Tassarotti [CTc22].[1] They developed the modality specifically for reasoning about crashes in the Perennial program logic [Cha22; Cha+21], but the idea behind

[1]While crucial to the workings of Perennial, the post-crash modality is unfortunately not described in any of the published papers about Perennial. We therefore directly cite the Coq mechanization of Perennial where the modality appears.

their modality can also be applied more generally to reason about the kind of non-local resource changes we have described. We continue to use the name post-crash modality, but emphasize that the modality is not only applicable to reasoning about crashes.

Perennial is a program logic for proving crash-safety in a setting with volatile memory and a durable disk. The post-crash modality, $\langle \mathsf{PC} \rangle$, is used to express the way in which resources change due to a system crash. As an example, the modality discards resources that correspond to the parts of the physical state that resides in volatile memory and preserves resources that correspond to the parts of the physical state that reside on the durable disk. As mentioned in the Introduction, the change to the physical state that occurs at a crash can not be expressed as a frame-preserving update to the ghost state for the physical state.

The key idea of the post-crash modality is to forgo updating the existing ghost state and instead allocate *new* ghost locations. That is, instead of updating a resource $\boxed{a}^{\gamma}$ to $\boxed{b}^{\gamma}$, which would require there to be a frame-preserving update $a \rightsquigarrow b$, a *new* ghost ownership assertion $\boxed{b}^{\gamma'}$, for a new ghost name γ', is allocated instead. The resources at the new ghost locations need not be frame-preserving updates of the earlier existing resources. Thus this approach side-steps the issue of not being able to make non-frame-preserving changes to ghost state. Since the new ghost locations have no inherent relation to the old ghost locations, some relationship between the two must be explicitly established, and it is required that one immediately stops using the old ghost name γ and switches to the new ghost name γ'. At a crash, new ghost assertions are allocated for the ghost state used internally in the program logic. Since allocating new ghost assertions results in new ghost names, this has the effect that the global ghost names that the proof is parameterized over are now obsolete as they refer to ghost locations prior to the crash. The post-crash modality then mediates between ghost state for the old global ghost names and ghost state for the new global ghost names.

In order to do this, the modality does not take an assertion of type iProp as argument, but instead has the type

$$\langle \mathsf{PC} \rangle : (\mathsf{GlobalGnames} \to \mathsf{iProp}) \to \mathsf{iProp},$$

where GlobalGnames is a record of all the ghost names used by the program logic in question (originally, Perennial). The modality is then defined by:

$$[\![\langle \mathsf{PC} \rangle\, P]\!] \triangleq \forall \sigma, \sigma', \vec{\gamma}'.\, R(\sigma, \sigma', \vec{\gamma}') \mathrel{-\!\!*} P(\vec{\gamma}') * R(\sigma, \sigma', \vec{\gamma}').$$

The R above is part of the definition of the post-crash modality. It relates the global ghost names and physical state before the crash σ with the physical state after the crash σ' and the new global ghost names $\vec{\gamma}'$.

When used, the post-crash modality is usually given an argument of the form $\lambda \vec{\gamma}'.\, Q$ where Q has to use the ghost names in $\vec{\gamma}'$ for its global ghost names and *not* use the old global ghost names $\vec{\gamma}$. Following this, rules for the post-crash modality

are of the form $P \vdash \langle \mathsf{PC} \rangle (\lambda \vec{\gamma}'.\, Q)$. When proving such rules one ends up with goals of the form

$$P * R(\sigma, \sigma', \vec{\gamma}') \mathrel{-\!\!*} Q(\vec{\gamma}') * R(\sigma, \sigma', \vec{\gamma}').$$

For instance, to prove the rule

$$\ell \hookrightarrow^{\vec{\gamma}.\mathrm{heap}} v \mathrel{-\!\!*} \langle \mathsf{PC} \rangle (\lambda \vec{\gamma}'.\, \ell \hookrightarrow^{\vec{\gamma}'.\mathrm{heap}} v)$$

for points-to assertions, one would have to prove

$$\ell \hookrightarrow^{\vec{\gamma}.\mathrm{heap}} v * R(\sigma, \sigma', \vec{\gamma}') \mathrel{-\!\!*} \ell \hookrightarrow^{\vec{\gamma}'.\mathrm{heap}} v * R(\sigma, \sigma', \vec{\gamma}').$$

The crux of the proof is to turn the "old" points-to assertion into the "new" points-to assertion. Making this possible is the purpose of the R resource. It serves as a catalyst to make this transition possible, without being consumed itself. In our particular example with points-to assertions, R could be defined as

$$R \triangleq \ell \hookrightarrow^{\vec{\gamma}.\mathrm{heap}} v \lor \ell \hookrightarrow^{\vec{\gamma}'.\mathrm{heap}} v.$$

More generally, R is defined such that all the relevant resources can be "exchanged" from old to new in this manner.

We now describe some of the limitations and problematic aspects of the above approach. Later on, we will show how our new nextgen modality addresses these shortcomings.

Poor interaction with the $\Box$ modality. The following rule is not possible to prove for the post-crash modality

$$\Box \langle \mathsf{PC} \rangle P \vdash \langle \mathsf{PC} \rangle \Box P.$$

Unfolding the model of the post-crash modality we see that this amounts to proving

$$\Box(\forall \sigma, \sigma', \vec{\gamma}'.\, R(\sigma, \sigma', \vec{\gamma}') \mathrel{-\!\!*} P(\vec{\gamma}') * R(\sigma, \sigma', \vec{\gamma}')) \vdash$$
$$\forall \sigma, \sigma', \vec{\gamma}'.\, R(\sigma, \sigma', \vec{\gamma}') \mathrel{-\!\!*} \Box P(\vec{\gamma}') * R(\sigma, \sigma', \vec{\gamma}')$$

We need to be able to show that P holds persistently, but we only know that a wand implying P holds persistently. Since we do not have R persistently, when we apply the wand to R we do not get P persistently. Thus, the lemma can not be proven.

The persistently modality plays a crucial role in Iris and the Iris proof-mode (IPM) for Coq [KTB17c]. The IPM keeps a so-called *persistent context* which consists of propositions that hold under the $\Box$ modality. When introducing the post-crash modality (using the `iModIntro` tactic) the IPM requires the rule

$$\frac{P \vdash \langle \mathsf{PC} \rangle Q}{\Box P \vdash \langle \mathsf{PC} \rangle \Box Q}$$

in order to be able to transform the persistent context. However, for reasons similar to the above, we can not prove this rule. This makes the post-crash modality more challenging and cumbersome to use in practice in Coq.

Invariants and the post-crash modality A key feature in Iris is *invariants*. An invariant is denoted $\boxed{I}^{\iota}$ and means that the assertion I is an invariant that a program maintains at every step of execution (the ι is not important for our purposes). It is not clear how invariants that contain ghost state, that uses global ghost names which are changed by the post-crash modality, can work with the post-crash modality as the invariant assertion is constant. At the very least, such invariants would have to be parameterized by the global collection of ghost names in order for the post-crash modality to be able to update them, and, as such, details of the post-crash modality would leak into invariants. In Section 5.4.3 we give an example of using our nextgen modality together with invariants, where the rules only have natural and necessary changes compared to normal Iris invariants.

No interaction with custom ghost state. As we have seen, the R resource in the model of the post-crash modality facilitates an exchange between old resources and new resources. This means that knowledge of certain global ghost names and resources are baked-in or hard-coded into the definition of the post-crash modality. The implication of this is that the reach of the modality can not extend to user-defined ghost state. Specifically, for an RA A and a ghost location γ, unknown to the definition of $\langle \mathsf{PC} \rangle$, no rule of the form

$$\boxed{a}^{\gamma} \vdash \langle \mathsf{PC} \rangle \boxed{b}^{\gamma}$$

where $a \neq b$, can exist. In other words, the only such rule is the one where $a = b$, meaning that the $\langle \mathsf{PC} \rangle$ modality can have no interaction with user-defined ghost state. This means that it is not possible to use the postcrash modality to give logically atomic specifications for user-defined durable concurrent data structures under a weak consistency models (such as the one in [VB23a]) whose specification relies on user-defined ghost state.

Not principled. While the post-crash modality cleverly works around the limitations in Iris for updating ghost state, we find that the mechanism it uses is not as principled as one could want. As we have shown, the workings of the post-crash modality rely on changing otherwise globally fixed ghost names. This can be confusing, both on paper and in Coq. For instance, it means that two points-to assertions that are notationally the same, can in fact be different as they "invisibly" use two different ghost names. The post-crash modality does not remove or otherwise invalidate old resources; it is up to the user of the modality to carefully apply lemmas that translate old resources, while also making sure that no old resources are still used. In Coq the modality relies on creating multiple instances of a type class that contains the global ghost names. Having multiple instances of a type class is not an idiomatic use of type classes — and while it does work in practice, we have not found any Coq documentation about which instance of a type class Coq uses when multiple are in scope. The modality therefore relies on undocumented behavior of the implementation of Coq.

5.3 The Basic Nextgen Modality

In this section we introduce the *basic* nextgen modality. Here the word "basic" means that the modality is a low-level addition to the Iris *base logic*. It is a minimal extension to Iris that enables one to express non-frame-preserving updates. The basic nextgen modality then facilitates the definition of higher-level nextgen modalities for specific purposes. This approach follows the Iris tradition of keeping the base logic minimal and simple, while defining more complex notions *inside* the logic. Technically, the nextgen modality can be seen as a *family* of modalities in that it is parameterized by a so-called generational transformation (defined below), and we show that the nextgen modality encompasses two existing Iris modalities, namely the persistently and the plain modalities.

The basic nextgen modality is written $\dot{\Rightarrow}^t P$ where $P : \mathsf{iProp}$ is an assertion and $t : M \to M$ is a function on the global RA. The dot above the symbol indicates that this is the *basic* nextgen modality. We call the function t a *generational transformation* or sometimes just a *transformation*. The assertion $\dot{\Rightarrow}^t P$ should be read "given a generational transformation described by t then P holds in the next generation".

In order for the modality to be sensible, the generational transformation needs to satisfy a few basic properties. Whenever we write $\dot{\Rightarrow}^t$, we assume that t ranges over functions with these properties. The requirements for t are given in the following definition.

Definition 5.3.1 (Generational transformation)**.** *Given a resource algebra A, a generational transformation is a function $t : A \to A$ that satisfies the following conditions.*

1. *It is monotone with respect to the inclusion order of the resource algebra.*

$$\forall x, y.\, x \preccurlyeq y \Rightarrow t(x) \preccurlyeq t(y)$$

2. *It preserves validity of elements.*

$$\forall x.\, \mathcal{V}(x) \Rightarrow \mathcal{V}(t(x))$$

3. *It is non-expansive with respect to the ordered family of equivalences (OFE) for A.*

$$\forall n, x, y.\, x \overset{n}{=} y \Rightarrow f(x) \overset{n}{=} f(y)$$

The first two conditions should seem reasonable. The first condition is necessary as the model of Iris uses monotone predicates over RAs and as we see in Section 5.3.4 this condition ensures that the meaning of $\dot{\Rightarrow}^t P$ is monotone in the model. The second condition is necessary as Iris maintains the property that the owned resources are always valid, hence the generational transformation needs to maintain this validity. The third condition pertains to an aspect of RAs that we have not described, namely that they contain an OFE or a "step indexed equality" [Jun+18a]. We include the condition here for completeness and for readers who are familiar with OFEs.

BNG-OWN
$\mathsf{Own}\,(a) \vdash \Rightarrow^t \mathsf{Own}\,(t(a))$

BNG-MONO
$$\frac{P \vdash Q}{\Rightarrow^t P \vdash \Rightarrow^t Q}$$

BNG-CONJ
$\Rightarrow^t P \land \Rightarrow^t Q \dashv\vdash \Rightarrow^t (P \land Q)$

BNG-DISJ
$\Rightarrow^t P \lor \Rightarrow^t Q \dashv\vdash \Rightarrow^t (P \lor Q)$

BNG-LATER
$\triangleright \Rightarrow^t P \dashv\vdash \Rightarrow^t \triangleright P$

BNG-EXISTS
$\Rightarrow^t \exists x.\, P \dashv\vdash \exists x.\, \Rightarrow^t P$

BNG-FORALL
$\Rightarrow^t \forall x.\, P \dashv\vdash \forall x.\, \Rightarrow^t P$

BNG-SEP
$$\frac{\forall x, y.\, t(x \cdot y) = t(x) \cdot t(y)}{\Rightarrow^t P * \Rightarrow^t Q \vdash \Rightarrow^t (P * Q)}$$

BNG-PERS
$$\frac{\forall x.\, t(|x|) = |t(x)|}{\Box \Rightarrow^t P \dashv\vdash \Rightarrow^t \Box P}$$

BNG-TRANS
$\Rightarrow^{t_1} \Rightarrow^{t_2} P \dashv\vdash \Rightarrow^{t_2 \circ t_1} P$

BNG-IDEMP
$$\frac{\forall x.\, t(t(x)) = t(x)}{\Rightarrow^t \Rightarrow^t P \vdash \Rightarrow^t P}$$

BNG-PLAINLY
$\Rightarrow^t \blacksquare P \dashv\vdash \blacksquare P$

Figure 5.2: Rules for the basic nextgen modality.

5.3.1 Rules

Figure 5.2 shows a selection of rules for the basic nextgen modality. The first rule, BNG-OWN is the nextgen modality's raison d'être. It states that the transformation t is applied to owned ghost state. As we are at the level of the base logic this rule concerns the Own assertion and not ghost locations.

The following rules in the figure state that the nextgen modality is monotone (BNG-MONO), commutes with conjunction (BNG-CONJ), disjunction (BNG-DISJ), the later modality (BNG-LATER), exist (BNG-EXISTS), and forall (BNG-FORALL). Together these rules ensures that the basic nextgen modality is well-behaved and convenient to work with. However, not all rules of this form that we would want hold without making further requirements on the transformation. If we look at the next rule BNG-SEP, we see that it states an additional demand on t. In Definition 5.3.1 we defined the *essential* properties that a transformation must possess, but in practice, transformations usually satisfy more properties than those, and in those cases more rules are sound. In the case of BNG-SEP the requirement is that the transformation commutes with the monoid operation of the RA. If this is the case, then two assertions under a nextgen modality can be combined under one nextgen modality. Here it is fairly clear how the requirement on t relates to the rule. The same is the case for the two rules BNG-PERS and BNG-IDEMP. If the transformation commutes with the core of the RA (when it is defined) then the modality commutes with the persistently modality. And, if the transformation is idempotent then so is the modality (BNG-TRANS holds for all transformations though). In all of these cases, the rule for the nextgen modality quite directly reflects the property of the transformation.

Remark It may be a bit surprising that BNG-SEP only holds in one direction. We emphasize that the direction that *does* hold is the important direction. For instance,

this direction is used by the Iris proof mode when introducing the modality. To give some intuition as for why the direction

$$\text{⧗}^t(P * Q) \vdash \text{⧗}^t P * \text{⧗}^t Q$$

is not sound, the left-hand side means that there is some resource a such that $t(a)$ satisfies $P*Q$. To show the right-hand side we would need to split $t(a)$ into resources b and c such that $t(a) = b \cdot c$, b satisfies $\text{⧗}^t P$, and c satisfies $\text{⧗}^t Q$. However, the disjunction on the left-hand side only implies that $t(a)$ can be split into two resource satisfying P and Q. We get stuck on the fact that $t(a) = b \cdot c$ does not imply that there exists b' and c' such that $a = b' \cdot c'$, $t(b') = b$, and $t(c') = c$. In more plain words: being able to split resources in the next generation does not necessarily mean that there was is way to split resources in the current generation. The rule above does hold if this property is required of the transformation. But, in practice transformations do not satisfy it and we have had no need for it.

Since the nextgen modality modifies resources, it has no effect on propositions that do not rely on resources. In Iris, such propositions are described with the *plainly modality* $\blacksquare P$, which means that P holds without using any resources. The rule BNG-PLAINLY states that the nextgen modality has no effect in the presence of the plainly modality. If we rephrase this lemma a bit we get what we call the soundness rule for the nextgen modality:

$$\frac{\vdash \text{⧗}^t P \qquad \text{plain}(P)}{\vdash P}\ \text{BNG-SOUND}$$

This states that if an assertion P is plain (meaning that $P \vdash \blacksquare P$) and can be derived under the nextgen modality, then the basic nextgen modality can be eliminated. A consequence of this rule is that results shown under the nextgen modality also has meaning outside of the nextgen modality, which is crucial when one wishes to prove an overall soundness or adequacy result for a program logic that makes use of the basic nextgen modality.

Just as important as the rules that do hold, is the one that does not. The following frame rule is *not* sound

$$Q * \text{⧗}^t P \not\vdash \text{⧗}^t(Q * P).$$

If Q holds in the current generation and P holds in the next generation then it is not necessarily sensible to move Q unchanged into the next generation. The equivalent rule for the update modality holds and is crucial for that modality's purpose. For the nextgen modality the opposite is the case: invalidating the frame rule is clearly *necessary* to arrive at a modality that can express non-frame-preserving changes to ghost state. Another rule that, quite naturally, is not sound is commutativity between the basic nextgen modality and the update modality:

$$\text{⧗}^t \mathrel{\dot{\Rightarrow}} P \not\vdash \mathrel{\dot{\Rightarrow}} \text{⧗}^t P.$$

As we see in Section 5.4.3 this has an impact on the way adequacy is proven for program logics that use the nextgen modality.

5.3.2 Comparison to the Post-Crash Modality

In a nutshell, the difference between the post-crash modality and the nextgen modality is that where the post-crash modality works *around* the limitation that Iris does not allow for non-frame-preserving updates, the nextgen modality addresses the problem head-on by lifting the limitation through an extension of the Iris base logic. In doing this the nextgen modality addresses the limitations we identified for the post-crash modality. In particular, since the nextgen modality modifies existing resources, it does *not* rely on changing ghost names and hence it does not incur the problems associated with that. We are able to prove all the expected rules, including BNG-PERS that does not hold for the post-crash modality.

5.3.3 Special Cases of the Basic Nextgen Modality

We now show that the persistently and the plainly modalities are special cases of the basic nextgen modality and thus, in a sense, our "extension" of the Iris base logic is perhaps better referred to as a "generalization and simplification" of the Iris base logic.

Example 5.3.2. *The basic nextgen modality can be used to define a modality equivalent to the persistently modality in Iris. This is achieved by taking the transformation to be the persistent core of the global RA.*

$$\text{⇾}^{|-|}\, P \dashv\vdash \Box\, P \tag{5.2}$$

In generational terms, this corresponds to a generation that only keeps duplicable resources.

Example 5.3.3. *The basic nextgen modality can be used to define a modality equivalent to the plainly modality, by taking the transformation to be the constant function that returns the unit element of the global RA.*

$$\text{⇾}^{\lambda a.\varepsilon}\, P \dashv\vdash \blacksquare\, P \tag{5.3}$$

In generational terms, this corresponds to a generation that throws away all resources.

The equivalences (5.2) and (5.3) are both easy to prove using the semantics of the basic nextgen modality, which we present in the following section.

5.3.4 Model

We now explain the semantics of the basic nextgen modality in the model of Iris.

To simplify the presentation and to focus on the interesting parts, we pretend that the semantic domain of Iris propositions is simply monotone predicates over resources:

$$[\![\mathsf{iProp}]\!] \triangleq M \xrightarrow{\mathsf{mon}} \textsc{Prop}.$$

The gap between this simplified definition and the full model of Iris is largely orthogonal to the semantics of the nextgen modality. We ignore the recursive domain equation arising from higher-order ghost state and step indices for the later modality. The benefit is that this simplifies the presentation and makes it easier to understand for readers who are not familiar with the particularities of the model of Iris, but who might be familiar with the more widely used predicates-over-resources model of separation logic. Our mechanization of the nextgen modality in Coq, of course, uses the "full" model of Iris, and we refer readers interested in all the details to the accompanying Coq formalization.

The model of the nextgen modality is exactly what one would expect from its behavior in the logic:

$$[\![\mathrel{⤇}^{t} P]\!] \triangleq \lambda x.\, [\![P]\!](t(x))$$

In order for this definition to be well-defined it must be monotone.

Lemma 5.3.4. *If* $x \preccurlyeq y$ *then* $[\![P]\!](t(x))$ *implies* $[\![P]\!](t(y))$.

Proof. Since $[\![P]\!]$ is monotone it suffices to show that $t(x) \preccurlyeq t(y)$. This follows from condition 1 of Definition 5.3.1. □

With this model all the rules that we have seen are sound.

5.3.5 Generational Resource Algebras

When using the nextgen modality with particular resources, one usually picks the type of resources and the transformations for it in unison. We use the term *generational RA* to mean a RA together with transformation function over it or a set of such functions. For many of the existing RAs in Iris there are obvious transformation functions that one could use with them. As an example, for the well known authoritative RA $\textsc{Auth}(A)$ and a transformation $t : A \to A$, there is a transformation t_{A} that applies t to both the authoritative element and fragments such that

$$t_A(\bullet a) \triangleq \bullet(t_A(a)) \qquad\qquad t_A(\circ b) \triangleq \circ(t_A(b)).$$

This transformation is part of the generational RA that we use in Section 5.4.

Just like Iris contains a library of RAs constructions that one can combine for concrete proofs, one can imagine a similar library of constructions for generational RAs. Our Coq mechanization contains a few such building blocks.

5.3.6 A Transformation for Ghost Locations

So far, we have seen the basic nextgen modality that applies a transformation to owned elements of the global RA. As described in Section 5.2.1, Iris is usually instantiated with a global RA of a particular shape. To arrive at higher-level nextgen

modalities, the first step is to use transformation functions that preserve this shape. To this end we will specify point-wise what should happen to each ghost location and thus we will use a map of transformations:

$$TM \triangleq \prod_{i \in I} \text{GName} \xrightarrow{\text{fin}} (M_i \to M_i)$$

This definition is equal to the global RA in Equation (5.1) except that the type of the "leafs" is changed from M_i to $M_i \to M_i$. From a map of transformations $tm \in TM$, we can construct a transformation on the global RA in the natural way:

$$T^{tm} : M \to M$$

$$T^{tm}(m) = \lambda i, \gamma. \begin{cases} tm(i,\gamma)(m(i,\gamma)) & \text{if } \gamma \in \text{dom}(tm(i)) \text{ and } \gamma \in \text{dom}(m(i)) \\ m(i,\gamma) & \text{if } \gamma \in \text{dom}(m(i)) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Spelled out, an element m of the global RA M is transformed such that each leaf $m(i,\gamma)$, where $i \in I$ and $\gamma \in \text{GName}$, is transformed by the function $tm(i,\gamma)$ if this exists in the transformations map. Otherwise, the element at the leaf is left unchanged.

For any $tm \in TM$, we then obtain the following rules for the basic nextgen modality and ownership of an $a : M_i$ at a ghost location γ.

$$\frac{\gamma \in tm(i)}{\boxed{a}^{\gamma} \vdash \text{⇜}^{t_{tm}} \boxed{tm(i,\gamma)(a)}^{\gamma}} \qquad \frac{\gamma \notin tm(i)}{\boxed{a}^{\gamma} \vdash \text{⇜}^{t_{tm}} \boxed{a}^{\gamma}}$$

This construction provides the foundation for building higher-level nextgen modalities.

A simpler variant of this construction is one where the map has the form $\prod_{i \in I}(M_i \to M_i)$. That is, where the transformation is given only per type of RA and not per type of RA *and* ghost name. Which variant to use depends on the circumstances, in the next section we see an example of using the simpler one.

5.3.7 Mechanization in Coq

As mentioned earlier we have mechanized the nextgen modality in Coq. The development contains the definition of the basic nextgen modality and its rules. Through type class instances the nextgen modality is integrated into the Iris Proof Mode such that it works as seamlessly as existing modalities.

Despite the nextgen modality being an extension to the base logic, we do not need to fork or modify the existing Iris Coq development. Due to the way Iris is mechanized one can define new constructs in terms of the model as long as the semantic domain is unchanged.

The mechanization also contains a number of generational transformations for common RAs and the transformation for ghost locations from the previous section.

5.4 Case Study of the Nextgen Modality

The basic nextgen modality lays the foundation for expressing non-frame-preserving updates. However, thus far, we've left out exactly *how* a concrete instance of a nextgen modality is defined. In this section, we present a case study of the nextgen modality.

To present a compelling case study, we need a language that exhibits some kind of non-local changes to its physical state. We thus begin this section by presenting a language with such behaviors, for which it would be difficult to define a modular and practical program logic without non-frame-preserving updates.

Next, we will give a concrete definition of a nextgen modality, by defining the right generational transform function t. Finally, we use the nextgen modality to define an elegant program logic for the language in question.

5.4.1 Presenting StackLang

We now present a language with non-local updates to its physical state, called StackLang. At its core, StackLang is a language with a high-level representation of a call-stack, where stack frames (henceforth referd to as stack regions) are pushed and popped in a well-bracketed way, and where stack allocated data must follow the derived lifetime behavior of its region. Upon return of a function call, stack regions are popped, and all the associated stack locations get deallocated. As such, function returns trigger a non-local change to the physical state that is hard to capture as a frame-preserving update.

A sound program logic for StackLang must therefore somehow deal with the deallocation of stack regions. A naïve approach may simply require the program logic rule for function returns to depend on the relevant ghost state in the precondition. Such a rule would define a precondition containing *all* ghost state fragments that would get deallocated by the return expression. Unfortunately, this approach counteracts the benefits of local reasoning typically granted by separation logic. Instead, our goal will be to construct a program logic with a rule for function returns that does not directly depend on fragments from the stack region.

The end goal of this section is to create a modular and practical program logic for StackLang. While other approaches exist, we want a program logic that does not require a lot of bookkeeping, or any instrumentation of the language itself. In other words, we want to define a program logic that does not require sophisticated ghost state leaking into the program rules, or any fundamental changes to the operational semantics of the language. But first, let's begin with a presentation of the syntax and semantics of the language.

5.4.2 Syntax and Semantics of StackLang

Figure 5.3 defines the syntax of StackLang values, expressions, and evaluation contexts. The definition and behavior of continuations follows Timany and Birkedal

$$
\begin{array}{llll}
\text{Index} & i & \triangleq & \mathbb{N} \\
\text{LocalityTag} & \mu & ::= & \mathsf{global} \mid \mathsf{local}(i) \\
\text{Value} & v & ::= & \mathsf{true} \mid \mathsf{false} \mid n \mid 1 \mid \lambda^{\mu}\, k, x.e \mid \ell^{\mu} \mid \mathsf{cont}^{i}(K) \mid (v, v) \\
\text{Expression} & e & ::= & x \mid \mathsf{true} \mid \mathsf{false} \mid n \mid 1 \mid \lambda^{\mu}\, k, x.e \mid \ell^{\mu} \mid \mathsf{cont}^{i}(K) \mid \\
 & & & e \oplus e \mid (e, e) \mid \pi_{\{1,2\}} e \mid e(e) \mid \mathsf{Return}(e)(e) \mid \mathsf{let}\ x := e\ \mathsf{in}\ e \mid \\
 & & & \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{salloc}(e) \mid \mathsf{halloc}(e) \mid !e \mid e \leftarrow e \\
\text{Evaluation} & K & ::= & \cdot \mid K \oplus e \mid v \oplus K \mid (K, e) \mid (v, K) \mid \pi_{\{1,2\}} K \mid \\
\text{Context} & & & K(e) \mid v(K) \mid \mathsf{Return}(K)(e) \mid \mathsf{Return}(v)(K) \mid \mathsf{let}\ x := K\ \mathsf{in}\ e \mid \\
 & & & \mathsf{let}\ x := v\ \mathsf{in}\ K \mid \mathsf{if}\ K\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{salloc}(K) \mid \mathsf{halloc}(K) \mid \\
 & & & !K \mid K \leftarrow e \mid v \leftarrow K
\end{array}
$$

Figure 5.3: StackLang syntax

[TB19]'s work on mechanized verification of programs with continuations, who define continuations $\mathsf{cont}^{i}(K)$ as suspended evaluation contexts. A key difference, is that continuations in StackLang are labelled with an index i, specifying which stack region the continuation belongs to.

Similarly, we label function closures $\lambda^{\mu}\, k, x.e$ and locations ℓ^{μ} with a locality tag μ, which specifies their *lifetime*. A global tag means the function or location has a permanent lifetime (*i.e.,* the heap), while a $\mathsf{local}(i)$ tag means the function or location has the same lifetime as stack region i. The index i in locality $\mathsf{local}(i)$ is *relative* to the top of the stack. For instance, $\ell^{\mathsf{local}(0)}$ refers to a stack allocated location in the topmost stack region. Likewise, a continuation with index i refers to the i^{th} stack region from the top, and invoking it will thus deallocate the i most recent regions.

The locality tags form an order based on their lifetime. We write $\mu_1 \sqsubseteq \mu_2$ whenever μ_1 has a shorter lifetime than μ_2, defined as follows:

$$\mu \sqsubseteq \mathsf{global} \qquad\qquad \frac{i_1 \leq i_2}{\mathsf{local}(i_1) \sqsubseteq \mathsf{local}(i_2)}$$

By default, any value without a locality tag (such as integers and booleans) implicitly have a permanent lifetime, and can thus be interpreted as having a global tag. This lets us lift the $\sqsubseteq$ relation to values. We write $v_1 \sqsubseteq v_2$ to state that the lifetime of v_1 is smaller than the lifetime of v_2, and we write $\mu \sqsubseteq v$ to state that the lifetime of v is at least μ.

New locations are allocated using $\mathsf{halloc}(e)$, which allocates locations with a global tag, and $\mathsf{salloc}(e)$, which allocates locations with a $\mathsf{local}(0)$ tag. The remaining values and expressions are defined as in a typical lambda calculus with references, where x is a variable, n stands for any natural number, and $\oplus$ is shorthand for binary operators. Finally, evaluation contexts define a left-to-right and call-by-value evaluation strategy.

$$(h, s, \mathsf{let}\ x := v\ \mathsf{in}\ e) \to_K (h, s, e[v/x]) \qquad (h, s, \pi_1(e_1, e_2)) \to_K (h, s, e_1)$$

$$(h, s, \pi_2(e_1, e_2)) \to_K (h, s, e_2) \qquad \frac{v_1 \oplus v_2 = v}{(h, s, v_1 \oplus v_2) \to_K (h, s, v)}$$

$$(h, s, \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2) \to_K (h, s, e_1)$$

$$(h, s, \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2) \to_K (h, s, e_2)$$

$$\frac{\mathsf{global} \sqsubseteq v \qquad \ell \notin dom(h)}{(h, s, \mathsf{halloc}(v)) \to_K (h \uplus \{\ell \mapsto v\}, s, \ell^{\mathsf{global}})}$$

$$\frac{s[0] = f \qquad \ell \notin dom(f) \qquad s' = s[0 := f \uplus \{\ell \mapsto v\}]}{(h, s, \mathsf{salloc}(v)) \to_K (h, s', \ell^{\mathsf{local}(0)})}$$

$$\frac{s[i](\ell) = v \qquad \mathit{shift}(v, i) = v'}{(h, s, !\ell^{\mathsf{local}(i)}) \to_K (h, s, v')} \qquad \frac{h(\ell) = v}{(h, s, !\ell^{\mathsf{global}}) \to_K (h, s, v)}$$

$$\frac{\mathsf{global} \sqsubseteq v \qquad \ell \in dom(h)}{(h, s, \ell^{\mathsf{global}} \leftarrow v) \to_K (h \uplus \ell \mapsto v, s, 1)}$$

$$\frac{\begin{array}{c}\mathsf{local}(i) \sqsubseteq v \\ s[i] = f \qquad \ell \in dom(f) \qquad \mathit{shift}(v, -i) = v' \qquad s' = s[i := f \uplus \{\ell \mapsto v'\}]\end{array}}{(h, s, \ell^{\mathsf{local}(i)} \leftarrow v) \to_K (h, s', 1)}$$

$$\frac{\mathit{shift}(v, 1) = v'}{(h, s, (\lambda^{\mathsf{global}}\ k, x.e)(v)) \to_K (h, \emptyset +\!\!+ s, \mathsf{Return}(\mathsf{cont}^1(K))(e[\mathsf{cont}^1(K)/k][v'/x]))}$$

$$\frac{\mathit{shift}(v, 1) = v' \qquad \mathit{shift}(e, i + 1) = e'}{(h, s, (\lambda^{\mathsf{local}(i)}\ k, x.e)(v)) \to_K (h, \emptyset +\!\!+ s, \mathsf{Return}(\mathsf{cont}^1(K))(e'[\mathsf{cont}^1(K)/k][v'/x]))}$$

Figure 5.4: StackLang inner step relation

The small-step operational semantics of StackLang is defined, as in [TB19], over two step relations. Each relation is defined over configurations (h, s, e), where h is the heap, e is the expression, and s is an ordered list of stack regions. The head of the stack describes the state of the topmost stack region, and function calls appends a new empty region to the head of the list, while function returns remove a specified number of regions from the list. We will refer to the heap and stack pair (h, s) as the store.

The first step relation $\rightarrow_K$ defines steps taken under some evaluation context K (Figure 5.4). Note that since the locality of locations, closures and continuations are potentially relative, parameters and return values are shifted to accurately reflect their new relative position. Likewise, a local function may enclose local values, which also needs to be shifted (we will get back to the reduction step of function calls below, after explaining function returns). Shifting values and expressions is handled by $\mathit{shift}(e, i)$, a partial function that shifts any stack location or continuation by the integer i. If the shift would put an index below zero, it fails, *i.e.,* it is undefined.

Loading from a location uses the lifetime tag to access the appropriate heap or region location. Likewise, storing to a location uses the lifetime tag to modify the appropriate heap or region location. Moreover, the locality of the store is guaranteed to be *monotone with respect to* $\sqsubseteq$, meaning that a location with locality μ can only store values v such that $\mu \sqsubseteq v$. This guarantee is enforced in the step relation by side-conditions over values that are added to the store via allocation or storing. In order to maintain relative positioning, values are shifted when stored to and loaded from the stack.

As highlighted above, we have designed StackLang such that it enforces the monotonicity of lifetimes of the store with dynamic requirements over the stored value. This is not essential, but means that StackLang can be interpreted as a kind of capability language with locality. It is noteworthy to point out that other capability languages, such as the CHERI capability machine ISA, also uses a locality bit to distinguish between the heap and the stack, with similar dynamic checks depending on the permission of the destination capability [Woo+14]. A notable difference however, is that closures are not *by construction* monotone. As such, there may be StackLang programs that execute and break monotonicity. However, in this work, our goal is not to define a capability safe language. Rather, what's important is that only well-behaved programs will provably satisfy a specification in the program logic we present below.

The second step relation $\rightarrow$ is built on top of $\rightarrow_K$, and defines the operational semantics of StackLang:

$$\textsc{ctx-bind}\ \frac{(h, s, e) \rightarrow_K (h', s', e')}{(h, s, K[e]) \rightarrow (h', s', K[e'])}$$

$$\textsc{ctx-ret}\ \frac{i \leq \mathit{length}(s) \qquad \mathit{shift}(v, -i) = v'}{(h, s, \mathsf{Return}(\mathsf{cont}^i(K))(v)) \rightarrow (h, \mathit{pop}^i(s), K[v'])}$$

The step for $\mathsf{Return}(\mathsf{cont}^i(K))(v)$ shifts v by $-i$, pops the top i regions from s, and is considered stuck whenever the continuation points to a stack region that does not exist. It is important that all function returns pop the appropriate number of stack regions. This includes function that return "normally", meaning the body simply reduces to a value. However, given the CTX-BIND rule, that might mean the topmost region is not popped as expected. To resolve this issue, function calls reduce to a return expression, surrounding the body of the function. Thus, if the body reduces to value, a proper return is still triggered.

The following program displays various features of StackLang.

$$(\emptyset, [\emptyset], (\lambda^{\mathsf{global}}\ k, x.!x{+}!(\mathsf{salloc}(41)))(\mathsf{halloc}(1))) \tag{5.4}$$
$$\to(\{\ell_1 \mapsto 1\}, [\emptyset], (\lambda^{\mathsf{global}}\ k, x.!x{+}!(\mathsf{salloc}(41)))(\ell_1^{\mathsf{global}})) \tag{5.5}$$
$$\to(\{\ell_1 \mapsto 1\}, [\emptyset; \emptyset], \mathsf{Return}(\mathsf{cont}^1(\cdot))(!\ell_1^{\mathsf{global}}{+}!\mathsf{salloc}(41))) \tag{5.6}$$
$$\to(\{\ell_1 \mapsto 1\}, [\emptyset; \emptyset], \mathsf{Return}(\mathsf{cont}^1(\cdot))(1{+}!\mathsf{salloc}(41))) \tag{5.7}$$
$$\to(\{\ell_1 \mapsto 1\}, [\{\ell_2 \mapsto 41\}; \emptyset], \mathsf{Return}(\mathsf{cont}^1(\cdot))(1{+}!\ell_2^{\mathsf{local}(0)})) \tag{5.8}$$
$$\to(\{\ell_1 \mapsto 1\}, [\{\ell_2 \mapsto 41\}; \emptyset], \mathsf{Return}(\mathsf{cont}^1(\cdot))(1 + 41)) \tag{5.9}$$
$$\to(\{\ell_1 \mapsto 1\}, [\{\ell_2 \mapsto 41\}; \emptyset], \mathsf{Return}(\mathsf{cont}^1(\cdot))(42)) \tag{5.10}$$
$$\to(\{\ell_1 \mapsto 1\}, [\emptyset], 42) \tag{5.11}$$

Note how the function call appends an empty region to the head of the stack (line 5.6), which gets subsequently popped when the function returns (line 5.10). We will use this example in Section 5.4.3 when introducing the program logic for StackLang.

5.4.3 A Program Logic for StackLang

Semantic interpretation of the store

The first step towards building a program logic for StackLang is to define a semantic interpretation of its store, as separation logic predicates. The StackLang store has two components: the heap h (a map from locations to values) and the stack s (an ordered list of maps from locations to values). To enable local reasoning about individual locations, we interpret both the heap and the stack such that we get separation logic points-to predicates.

In Iris, points-to predicates are typically defined using a special authoritative resource algebra for maps called a $\mathsf{gmapView}(K, V)$, where K is the domain, and V is the co-domain of the map. Since the stack is a *list* of maps, we first transform it into a map from index and location pairs to values, where the index represents the location's position in the stack list. Unlike the relative stack region index of values, this index represents its *real and global* stack region index. For example, consider the following stack: $[f_0; f_1; \{\ell \mapsto v\}]$. In this stack, the location ℓ is two regions down from f_0. The currently executing program (i.e. the owner of f_0) can thus reference the location via the value $\ell^{\mathsf{local}(2)}$. However, globally, it belongs to the 0^{th} stack region, and is thus indexed by $(0, \ell)$.

Thus far, we have two distinct resource algebras:

gmapView Location Value	resource algebra for the heap
gmapView ($\mathbb{N} \times$ Location) Value	resource algebra for the stack

Points-to predicates for heap and stack locations are then derived from their respective resource algebra. Going back to the above example, the state of ℓ is described by a points-to predicate mapping $(0, \ell)$ to v, denoted by $\boxed{0}\ \ell \mapsto v$. In order to connect the relative value in $\ell^{\mathsf{local}(2)}$ to the absolute value in the points-to predicate, it is necessary to keep track of the current size of the stack. We thus introduce a third resource algebra to track the size of the stack, by using the exclusive authoritative resource algebra construction of Iris, defined over natural numbers.

ExclAuth $\mathbb{N}$	resource algebra for the stack size

In summary, we define three resource algebras, used to define the following three separation logic predicates:

$\ell \mapsto v$	states that the heap location ℓ points to value v
$\boxed{k}\ \ell \mapsto v$	states that the stack location ℓ of region index k points to value v
$\boxminus\, m$	states that the stack is currently made up of m regions

Picking a transformation function

Next, we want to define a nextgen modality that describes what happens at function returns. More precisely, we want to define a modality that deallocates the relevant stack points-to predicates, while leaving unrelated predicates intact. To that end, we use the construction outlined in Section 5.3.6 to build a map of transformations, that together form a transformation on the global resource algebra used to interpret the StackLang store.

Since it is only the deallocation of stack locations that is non-frame-preserving, we are only interested in defining a transformation function defined over the resource algebra for the stack. We apply the identity transformation for all other resource algebras, including the ones outlined above, and any subsequently defined custom resource algebras.

Meanwhile, the transformation of the stack resource needs to exhibit very specific behavior. Specifically, the desired transformation of the stack resource algebra for a return that tagerts region n, denoted SCut^n (of type gmapView ($\mathbb{N} \times$ Location) Value $\to$ gmapView ($\mathbb{N} \times$ Location) Value) needs to filter out all the elements with an index of at least n.

Note that SCut^n is parameterized by a natural number n, and thus we define not just one, but a family of transformations. Indeed, in order to distinguish between returns that target different stack regions, we will need a family of nextgen modalities. We realize this construction by maintaining a transformation map tm which maps the heap and stack size resource algebras to the identity transformation, while leaving the transformation for the stack resource algebra *undefined*. We then *insert* the

relevant SCut^n transformation into t_{tm}, to define the appropriate nextgen modality. Below we formally define SCut^n, and the derived definition for ICut^n, which inserts SCut^n into the globally defined tm and uses the construction from Section 5.3.6 to create the transformation function.

$$\mathsf{SCut}^n(m) = m' \quad \textit{where} \quad \begin{array}{l} dom(m') \subseteq dom(m) \textit{ and} \\ \forall (k,\ell) \in dom(m), (k < n \land m'(k,\ell) = m(k,\ell)) \\ \quad \lor (k \geq n \land (k,\ell) \notin dom(m')) \end{array}$$

$$\mathsf{ICut}^n = T^{tm\{i := \mathsf{SCut}^n\}} \quad \textit{where} \quad i \in I \textit{ is the globally scoped id of the stack resource algebra}$$

By picking ICut^n as the generational transformation function, we can then finally formally define a nextgen modality for stack region deallocation.

$$\mathrel{\dot{\mapsto}}^n \triangleq \mathrel{\dot{\mapsto}}^{\mathsf{ICut}^n}$$

By the definition of ICut^n, we then prove the following introduction rules:

CUT-HEAP-INTRO
$$\ell \mapsto v \mathrel{\dot{\mapsto}}^n \ell \mapsto v$$

CUT-STACK-INTRO
$$\frac{k < n}{\boxed{k}\, \ell \mapsto v \vdash \mathrel{\dot{\mapsto}}^n \boxed{k}\, \ell \mapsto v}$$

CUT-STACK-INTRO-EMP
$$\frac{k \geq n}{\boxed{k}\, \ell \mapsto v \vdash \mathrel{\dot{\mapsto}}^n \top}$$

CUT-SIZE-INTRO
$$\boxminus\, m \vdash \mathrel{\dot{\mapsto}}^n \boxminus\, m$$

The introduction rule for stack points-to predicate requires that the stack location of a region k is lower than the deallocation at n. If k is at or above n, the fragment is deallocated, as expressed by the trivial rule CUT-STACK-INTRO-EMP.

Weakest precondition

We define a program logic for StackLang by using a variant of Iris weakest preconditions, denoted $\mathsf{wp}\ e\ \{\Phi\}$. In broad strokes, $\mathsf{wp}\ e\ \{\Phi\}$ expresses that the expression e does not get stuck, and if it terminates at some value v, then the predicates $\Phi(v)$ holds. Below we first recall a simplified version of the existing definition of Iris weakest preconditions for a single-threaded language and then we present our new variant incorporating the nextgen modality, similarly simplified (for the full definition, see the accompanying Coq formalization). The simplified Iris weakest precondition predicate is the unique predicate satisfying the following equation:

$$\mathsf{wp}\ e\ \{\Phi\} \triangleq \begin{cases} \Rrightarrow_\top \Phi(e) & \textit{if } e \textit{ is a value} \\ \forall \sigma, \mathsf{stateInterp}(\sigma) \mathrel{-\!\!*}\ {}_\top\Rrightarrow_\emptyset \textit{ e is reducible} & \textit{otherwise} \\ \land \triangleright \forall e_2, \sigma_2, (\sigma, e) \to (\sigma_2, e_2) \mathrel{-\!\!*}\ {}_\emptyset\Rrightarrow_\top & \\ \qquad \mathsf{stateInterp}(\sigma_2) * \mathsf{wp}\ e_2\ \{\Phi\} & \end{cases}$$

$$\textsc{Stk-Load}\quad \frac{\triangleright(\boxed{n}\,\ell \mapsto v * \boxminus\, m \twoheadrightarrow \mathsf{wp}\ K[\mathit{shift}(v,i)]\ \{\Phi\}) \qquad \boxed{n}\,\ell \mapsto v \qquad \boxminus\, m \qquad n = m - i - 1}{\mathsf{wp}\ K[!\ell^{\mathsf{local}(i)}]\ \{\Phi\}}$$

$$\textsc{Heap-load}\quad \frac{\triangleright(\ell \mapsto v \twoheadrightarrow \mathsf{wp}\ K[v]\ \{\Phi\}) \qquad \ell \mapsto v}{\mathsf{wp}\ K[!\ell^{\mathsf{global}}]\ \{\Phi\}}$$

$$\textsc{Salloc}\quad \frac{\triangleright(\boxminus\, m * \boxed{m-1}\,\ell \mapsto v \twoheadrightarrow \mathsf{wp}\ K[\ell^{\mathsf{local}(0)}]\ \{\Phi\}) \qquad \boxminus\, m \qquad 0 < m}{\mathsf{wp}\ K[\mathsf{salloc}(v)]\ \{\Phi\}}$$

$$\textsc{Halloc}\quad \frac{\triangleright(\ell \mapsto v \twoheadrightarrow \mathsf{wp}\ K[\ell^{\mathsf{global}}]\ \{\Phi\})}{\mathsf{wp}\ K[\mathsf{halloc}(v)]\ \{\Phi\}}$$

$$\textsc{Call-Global}\quad \frac{\triangleright(\boxminus\, m + 1 \twoheadrightarrow \mathsf{wp}\ K[\mathsf{Return}(\mathsf{cont}^1(K))(e[\mathsf{cont}^1(K)/k][\mathit{shift}(v,1)/x])]\ \{\Phi\}) \qquad \boxminus\, m}{\mathsf{wp}\ K[\lambda^{\mathsf{global}}\ k, x.e(v)]\ \{\Phi\}}$$

$$\textsc{Return}\quad \frac{\triangleright(\boxminus\, m - i \twoheadrightarrow \Rsh^{(m-i)}\ \mathsf{wp}\ K'[\mathit{shift}(v,-i)]\ \{\Phi\}) \qquad \boxminus\, m \qquad i \le m}{\mathsf{wp}\ K[\mathsf{Return}(\mathsf{cont}^i(K'))(v)]\ \{\Phi\}}$$

Figure 5.5: Excerpt of the Program Logic Rules for StackLang

If e is a value, then the postcondition Φ holds for that value. In the above definition, the postcondition is declared to hold under a fancy update modality. If e is not a value, it must be able to take a step. More concretely, given any state σ, assuming the ownership of the semantic interpretation of σ (in the case of StackLang, this semantic interpretation is defined as authoritative views of the three previously defined resource algebras), the expression e is reducible, and for any configuration it steps to, we have the semantic interpretation of the new state, and a weakest precondition of the new expression.

We define a variant of the above definition, which applies the nextgen modality at the appropriate expression, namely function returns. Henceforth, this is the definition we will be referring to.

$$\mathsf{wp}\ e\ \{\Phi\} \triangleq \begin{cases} \Rrightarrow_{\top} \Phi(e) & \textit{if } e \textit{ is a value} \\ \forall \sigma, \mathsf{stateInterp}(\sigma) \twoheadrightarrow {}_{\top}\Rrightarrow_{\emptyset} e \textit{ is reducible} & \textit{if } e = K[\mathsf{Return}(\mathsf{cont}^i(K'))(v)] \\ \land \triangleright \forall e_2, \sigma_2, (\sigma, e) \to (\sigma_2, e_2) \twoheadrightarrow {}_{\emptyset}\Rrightarrow_{\top} & \\ \quad (\mathrel{\text{¿}\!\!\to}^{length(\sigma.2)-i} \mathsf{stateInterp}(\sigma_2)) & \\ \quad * (\mathrel{\text{¿}\!\!\to}^{length(\sigma.2)-i} \mathsf{wp}\ e_2\ \{\Phi\}) & \\ \forall \sigma, \mathsf{stateInterp}(\sigma) \twoheadrightarrow {}_{\top}\Rrightarrow_{\emptyset} e \textit{ is reducible} & \textit{otherwise} \\ \land \triangleright \forall e_2, \sigma_2, (\sigma, e) \to (\sigma_2, e_2) \twoheadrightarrow {}_{\emptyset}\Rrightarrow_{\top} & \\ \quad \mathsf{stateInterp}(\sigma_2) * \mathsf{wp}\ e_2\ \{\Phi\} & \end{cases}$$

In this definition, the state interpretation and weakest precondition of the reduced expression is guarded by a nextgen modality, reflecting that a stack deallocation has occured.[2] The presence of this modality is crucial for proving the program logic rule for return expressions.

Since we are using a new definition of Iris weakest preconditions, it is important to prove that it is sound. To this end, we prove the following adequacy theorem.[3]

Theorem 5.4.1 (Adequacy of the nextgen weakest precondition)**.** *Let Φ be a first-order pure predicate. Assume $\vdash \mathsf{wp}\ e\ \{\Phi\}$, and $(\sigma, e) \to (\sigma_2, e_2)$, then the following two facts hold:*

1. *either (σ_2, e_2) is reducible, or e_2 is a value*
2. *if e_2 is a value, then $\Phi(e_2)$ holds*

Proof. The proof largely resembles the adequacy proof of Iris weakest preconditions, with the added burden of dealing with interweaved instances of the nextgen modality. The key difficulty lies in applying the various soundness rules for all the relevant Iris modalities, including the new soundness rule for the nextgen modality (BNG-SOUND); since the nextgen modality does not commute with the fancy update modality, the final result is an interweaved sequence of modalities that do not collapse into a finite number of modalities. As such, the proof must eliminate each modality one at a time in a proof by induction. □

Program logic rules

We now introduce the program logic for StackLang. Figure 5.5 presents a selection of program logic rules, namely for load, allocation, call and return. Since invoking a continuation discards the current surrounding evaluation context, the typical

[2] Here we present a version of the definision that is tailored specifically to StackLang. In the Coq mechanization, we define a version that parametrizes over an arbitrary programming language.

[3] Here again tailored to StackLang, but proved for a general single-threaded language in the Coq mechanization, without support for later credits.

bind-rule for Iris weakest preconditions does not work for StackLang expressions. Therefore each program logic rule is defined around a filled evaluation context K, and is presented in a continuation style.

The rules for stack and heap loads require a points-to predicate to the location in question. In the case of stack load, since the stack location reference is relative, it is additionally required to know the current size of the stack via the stack size resource. In each case, the continuing weakest precondition fills K with the loaded value, which is shifted in case of a stack load.

The rule for stack and heap allocations gives the continuation a new points-to predicate for the allocated location. In case of a stack allocation, the size of the stack determines the index of the stack points-to predicate.

Finally, the stack size resource itself is updated in the rules for calls and returns. The rule for a call simply increases the size of the stack by one. Note that the new region starts out as empty, and thus no resources are allocated. On the flipside, the rule for a return must not only decrease the stack size resource, it must somehow handle the *deallocation* of a number of stack regions, which may now be non-empty. In other words, the rule for return is only sound if it handles the deallocation of *all* stack points-to predicates associated to popped regions. Luckily, this is exactly expressed by the nextgen modality for stack region deallocation. As a result, it suffices to guard the continuing weakest precondition with $\mathrel{\dot{\mapsto}}^{(m-i)}$, which states that the next weakest precondition cannot depend on any points-to predicates for stack locations above $m - i$.

Having seen the return rule, we can now explain why this rule could not be realized with a frame-preserving update. As mentioned, we use the RA $\mathsf{gmapView}(\mathbb{N} \times \mathrm{Location}, \mathrm{Value})$ to model points-to predicates. When using this RA the weakest precondition contains an authoritative element for the stack denoted $\mathsf{gmapViewAuth}(\mathsf{flat}(s))$ where s is the physical stack from the operational semantics and flat converts the list of stores into the map over $\mathbb{N} \times \mathrm{Location}$ that we use for points-to predicates. When proving soundness of the rule for Return this authoritative ghost state must change from $\mathsf{gmapViewAuth}(\mathsf{flat}(s))$ into $\mathsf{gmapViewAuth}(\mathsf{flat}(pop^i(s)))$ to match the change in the operational semantics. For this to be a frame-preserving update one would need the resources for *all* the fragments for the affected locations. Hence, to prove a rule for Return using frame-preserving updates, the rule would require the user of the logic to supply all points-to predicates for all stack locations affected by the Return. This would be completely infeasible and non-modular as these points-to predicates could be shared in invariants, handed out to sub-parts of the proof, *etc.* With the nextgen modality we can instead change the resource in a non-frame-preserving way, and obtain the much simpler rule.

Example

Let's use these rules to prove a specification of the previously presented example program. The program starts executing in a configuration with a stack of size 1. Our

goal is to show the following specification:

$$\boxminus 1 \vdash \mathsf{wp}\ (\lambda^{\mathsf{global}}\ k, x.!x{+}!(\mathsf{salloc}(41)))(\mathsf{halloc}(1))\ \{\lambda v, v = 42\}$$

we prove the specification by applying the program logic rules given in Fig. 5.5. The first expression to execute is $\mathsf{halloc}(1)$. We thus begin by applying the rule for heap allocation HALLOC, and the new goal becomes:

$$\boxminus 1 \vdash \triangleright(\ell_1 \mapsto 1 \mathrel{-\!\!*} \mathsf{wp}\ (\lambda^{\mathsf{global}}\ k, x.!x{+}!(\mathsf{salloc}(41)))(\ell^{\mathsf{global}})\ \{\lambda v, v = 42\})$$

We introduce the new points-to predicate into the context. Next, we apply the rules for call, stack allocation, load and binary operations to reach the following context and goal:

$$\boxminus 2\ *\ \ell_1 \mapsto 1\ *\ \boxed{1}\,\ell_2 \mapsto 41 \vdash \mathsf{wp}\ \mathsf{Return}(\mathsf{cont}^1(\cdot))(42)\ \{\lambda v, v = 42\}$$

At this point, we must apply the rule for return. Since the continuation has offset 1, this will decrease the stack by 1. After applying the rule for return, we are left with the following goal:

$$\boxminus 1\ *\ \ell_1 \mapsto 1\ *\ \boxed{1}\,\ell_2 \mapsto 41 \vdash \mathrel{\dot{\Rightarrow}}^1\ \mathsf{wp}\ 42\ \{\lambda v, v = 42\}$$

Crucially, the new goal is guarded by the $\mathrel{\dot{\Rightarrow}}^1$ modality. The only way to introduce it is by applying monotonicity of the nextgen modality (BNG-MONO). Therefore, the next step is to introduce $\mathrel{\dot{\Rightarrow}}^1$ in front of all the relevant predicates in the context, and discard those which don't have such an introduction rule. We can apply CUT-HEAP-INTRO and CUT-SIZE-INTRO and introduce the modality in front of the heap points-to predicate and the stack size resource. However, we can't apply the introduction rule for the stack points-to predicate (CUT-STACK-INTRO), since the region index of $\boxed{1}\,\ell_2 \mapsto 41$ is not stricly smaller than 1. In fact, the whole purpose of $\mathrel{\dot{\Rightarrow}}^1$ is exactly to deallocate such points-to predicates. As such, we discard it, and apply BNG-SEP to get the following goal:

$$\mathrel{\dot{\Rightarrow}}^1(\boxminus 1\ *\ \ell_1 \mapsto 1) \vdash \mathrel{\dot{\Rightarrow}}^1\ \mathsf{wp}\ 42\ \{\lambda v, v = 42\}$$

We can then finally conclude by applying monotonicity, and prove the post-condition.

While the above proof sketch manually applies the introduction rule for the nextgen modality, the rule for commuting over separation conjuction, and monotonicity; each of these steps are automated when using our mechanization in Coq. Due to the integration of the nextgen modality with the Iris Proof Mode, described in Section 5.3.7, introducing the nextgen modality is handled by a single tactic, leading to a seamless experience when using the program logic for STACKLANG in Coq.

Custom ghost state and invariants

In the above example, the specification and proof serves to illustrate the use of the nextgen modality. However, given the simplicity of the program, it does not take

advantage of the full expressive power of the Iris logic. Notably, it does not depend on any custom ghost state or invariants, and does not display how they may interact with the new nextgen modality.

Since we have defined the $\Rightarrow^n$ modality to apply the identity transformation on any non-stack resource, any custom ghost state can easily introduce the modality. In constrast, more interesting questions arise when we consider the interaction between Iris invariants and the nextgen modality. Since an Iris invariant is guaranteed to hold at every step of a program's execution, how can it enclose stack allocated resources that might disappear at function returns? Clearly, it would not be sound for such invariants to outlive the stack values they correspond to. One possible sound solution would be to only allow invariants that do not enclose any stack points-to predicates. However, such a limitation would disallow interesting use-cases of Iris invariants, such as defining a *temporary* invariant that holds until a region has ended.

The ideal solution to the above would be invariants that can contain stack points-to predicates and that live for exactly as long as those stack locations. This is precisely what we achieve by creating a variant of Iris invariants that interacts with the nextgen modality.

Our variant of Iris invariants is parameterized either by natural number $\mathbf{n}$, or by a special value ∞, denoted $\boxed{P}^{\mathcal{N},\mathbf{n}}$ and $\boxed{P}^{\mathcal{N},\infty}$ respectively. One can think of this as the lifetime of the invariant. Invariants are allocated with the following allocation rules:

$$\textsc{inv-alloc}\quad \frac{\blacksquare\, \forall m.\, n < m \Rightarrow P \vdash \Rightarrow^m P \qquad \triangleright P}{\Rrightarrow_{\mathcal{E}} \boxed{P}^{\mathcal{N},\mathbf{n}}} \qquad \textsc{inv-alloc-any}\quad \frac{\blacksquare\, \forall m.\, P \vdash \Rightarrow^m P \qquad \triangleright P}{\Rrightarrow_{\mathcal{E}} \boxed{P}^{\mathcal{N},\infty}}$$

In the first rule, for allocating a promise for $\mathbf{n}$, one must prove that the body of the invariant is unaffected by a nextgen modality that discards stack regions above $\mathbf{n}$. In the second rules, for allocating an invariant with infinite lifetime, one most show that the body of the invariant is unaffected by any nextgen modality. This effectively ensures that the invariant can not contain stack points-to predicates.

The interaction between invariants and the nextgen modality depends on the parameter on the invariant:

$$\textsc{cut-inv-intro}\quad \frac{k < n}{\boxed{P}^{\mathcal{N},\mathbf{k}} \vdash \Rightarrow^n \boxed{P}^{\mathcal{N},\mathbf{k}}} \qquad \textsc{cut-inv-any-intro}\quad \boxed{P}^{\mathcal{N},\infty} \vdash \Rightarrow^n \boxed{P}^{\mathcal{N},\infty}$$

With this new invariant construction, it is possible to allocate invariants that enclose stack points-to predicates, and prove specifications of programs that may depend on them. As such, not only can we define invariants that are not impacted by $\Rightarrow^n$, we can also define invariants that may themselves be deallocated by a particular instance of $\Rightarrow^n$. The invariants exist for as long as it would be sound for them to do so, and are removed by the nextgen modality accordingly. This new definition of invariants displays the flexibility of the nextgen modality, which allows us to define

arbitrary transformations over ghost state, including the ghost state of invariants themselves.

We leave out the technical details of the definition of the new invariants, and refer to the Coq mechanization for those.[4] The key idea is to apply a transformation to invariants, which mimics the transformation function for stack points-to predicates, by indexing invariants by stack regions. The requirements on P when allocating invariants is carried over to the definition of the so-called world satisfaction relation, which is the internal Iris definition which tracks and stores all invariants, and then used to prove the soundness of the nextgen introduction rules for invariants.

5.5 Related and Future Work

As mentioned, the post-crash modality from Perennial is the work most closely related to the nextgen modality. We have already compared the two earlier in Section 5.3. To the best of our knowledge there is no other work that gives a general mechanism for performing non-frame-preserving updates in separation logic.

We think that there is much exciting future work to be done, and hope that we have just scratched the surface of the usefulness of the nextgen modality. Exploring the motivating examples that we sketched in the introduction is one possible avenue for future work. We are currently exploring the application of the nextgen modality to a concurrent setting with crashes and durable storage, and our current results seem very promising. One interesting challenge in this setting is that under a weak persistency model, crashes are non-deterministic and there is thus not a fixed transformation that can be applied to ghost state at a crash.

We think our nextgen modality can be used as a foundation to implement temporary read-only points-to predicates in Iris in the style of [CP17]. Our initial investigation into this seems to indicate that defining the resources for this and the nextgen modality itself is quite straightforward. However, defining a weakest precondition that validates the expected proof rules seems quite tricky. In particular, the "framed sequencing rule" of *op. cit.* is non-trivial to prove for a weakest precondition that contains a nextgen modality. We think solving this hurdle is very exciting future work, as read-only point-to predicates bring many benefits that Iris users are currently missing.

Turning to work related to our program logic for StackLang, we first remark that the program logic rules for StackLang make explicit use of evaluation contexts because returns may discard the current evaluation context. This style of proof rules is inspired by Timany and Birkedal's work on a program logic for programs with continuations [TB19].

[4] The technical details behind the definition of nextgen invariants are slightly more involved and include additional ghost state to remember the upper bound $\mathbf{n}$, and a transformation over this ghost state that alters it in lockstep with SCut^n. We have also generalized it to work with any arbitrary indexing type and order.

We are not aware of previous separation logics that explicitly account for deallocation of stack frames. The most closely related work is the work of Timany et al. [Tim+18] for reasoning about encapsulation of local state in a sequential programming language with a state monad and a Haskell-style polymorphically-typed `runST` construct. Timany et al. define a logical relation of the type system with `runST` in Iris and use it to show that `runST` encapsulates computations with local state and that such computations use regions allocated in a stack-like manner. A key point of *op. cit.* is that the operational semantics of the language is a standard operational semantics with a global heap, capturing how the language would be implemented in reality, whereas the logical relation allows one to reason *as if* regions were stack-allocated physically. This is achieved by a clever use of ghost state, which tracks the virtual stack of regions and connects it to the physical memory. To account for virtual deallocation of regions (popping the virtual stack of regions), Timany et al. essentially mark regions as dead in the ghost state and a key step in their proof of soundness of the logical relations model of the type system is then to show that the type system guarantees that one does not try to access a region that is dead. Thus Timany et al. manage to account for virtual deallocation using only frame-preserving updates, but it comes at the expense of having a global ghost resource that is threaded around in the reasoning, rather than having more modular local points-to predicates, and it is not clear how this approach would scale to a concurrent language, since it does not seem possible to share the global ghost resource among several threads. In contrast, the nextgen modality allows for more modular local reasoning and scales to concurrent languages (cf. our current explorations of the nextgen modality to a concurrent setting with crashes and durable storage mentioned above).

Chapter 6

A Nextgen Modality For Crashes In Spirea

6.1 Introduction

In this chapter we introduce a nextgen modality for crashes made to accommodate for the needs of Spirea. The modality is denoted

$$\langle \mathsf{NG} \rangle P.$$

That is, unlike previous nextgen modalities (such as the basic nextgen modality and the nextgen modality we saw for StackLang in the last chapter) it is not parameterized by anything. Instead, the transformation function that gets applied is hidden within the model of the modality and is "linked" or "connected" to separation logic resources. It is these resources that determine the transformation. As we will see, this novel idea is crucial to being able to handle the non-deterministic crashes in λ_{pmem} and it fundamentally increases the power and flexibility of the nextgen modality. The modality has three key features:

1. *Picks*, a form a dynamic choice about what transformations get applied to ghost locations.
2. *Promises*, a dynamic way of restricting the possible future transformations that can be picked for a ghost location.
3. *Dependent* promises, that makes it possible for the promise for one ghost location to depend on transformations for other ghost locations.

We note that the nextgen modality here also supports the same fundamental rules as the ones for the basic nextgen modality in Figure 5.2. In this exposition we focus on the novel aspects of the modality.

At present, the above features probably sound rather abstract to the reader and the full modality with all its features is rather intricate. Hence, in an attempt to ease

the reader into the modality the chapter proceeds as follows. We begin in Section 6.2 by explaining the key reason why Spirea needs a better modality for crashes. In Section 6.3 we sketch at a high-level the requirements that the modality needs to satisfy in order to be able to replace the post-crash modality. Motivated by non-deterministic crashes we introduce, in Section 6.4, a simplified version of the modality that features picks but not promises. The modality is then extended, in Section 6.5, with promises. This is a considerable additional complication necessitated by the fact that sometimes certain facts are known about what cannot happen at a crash. In Section 6.6 we demonstrate how the modality and its features are useful for Spirea, by defining a new state interpretation for BaseSpirea with generational ghost state designed for the modality. We show how this state interpretation in combination with the modality can accommodate for the crash step in λ_{pmem} and be used to model the assertions in BaseSpirea. This serves both as a case study on how to use picks and promises and as a new model for BaseSpirea that incorporates the benefits that the nextgen modality brings. In Section 6.8 we discuss future the work that is still missing.

6.2 Why Spirea Needs the Nextgen Modality

In this section we explain the key motivation for changing Spirea to use a nextgen-based modality for crashes instead of the post-crash modality. We have already described the general advantages of the nextgen modality compared to the post-crash modality. The advantage that is absolutely critical for Spirea is the nextgen modality's support for interacting with user-defined ghost state.

To see why this is useful, suppose we were to verify a durable concurrent data-structure in Spirea with respect to a specification with a strength comparable to a HOCAP-style or logically atomic triple specification. That is, a specification decidedly stronger than those we gave in Chapter 4 to the stack and the queue. To carry out a proof of such a specification one needs to keep precise track of the abstract state of the data-structure. We have seen this both in the verification of the MS queue the MPMC queue. To do this, the proof will almost always make use of ghost state. For instance, in the verification of the MPMC queue we used a *ghost list* that was closely related to the abstract and the physical state of the queue. Ghost state that corresponds to the physical state must be updated when the physical state is changed. Furthermore, for a durable concurrent data-structure the *abstract* state of the data-structure changes at its linearization point. Hence, ghost state that is related to the abstract state must be updated at the linearization point. But, both the abstract and the physical state of a data-structure can change at a crash, and therefore it is clear that the ghost state much change as well at a crash in accordance to this. For instance, imagine that a concurrent operation is carried out on a data-structure and a crash happens *between* the operation's linearization point and its persist point[1]

[1]Recall that a persist point is the program point after which the operation is certain to have been persisted.

Since the crash is after the linearization point any ghost state that corresponds to the physical state has already been updated, but if the change to the abstract state is lost at the crash, then the ghost state needs to be "rolled back" at the crash. This "rollback" is most likely not going to be a frame-preserving update and it concerns user-defined ghost state.

In summary, if we wish to verify durable concurrent data-structures w.r.t. stronger specifications that go beyond those that we have seen thus far, then we need user-defined ghost state that can change at a crash. For some user-defined RA A the user must also be able to describe inside the logic how elements of this resource might change at a crash. In other words, they need to be able to prove rules of the form

$$\boxed{a}^{\gamma} \vdash \langle \mathsf{NG} \rangle \boxed{a'}^{\gamma}$$

where the relationship between a and a' can be restricted or specified somehow. This capability is crucial to achieve stronger specifications of the sort described but is not possible with the post-crash modality. Hence, we need to use the nextgen modality.

6.3 Requirements

With the above in mind, one could state our present goal as: Define a nextgen modality that can be substituted for the post-crash modality everywhere in BaseSpirea and Spirea while making it possible for user-defined ghost state to interact with the nextgen modality. The former means that the new modality should subsume the post-crash modality in the sense that, after the substitution, all the existing rules are still sound. Of these rules, the most interesting ones those that involve the post-crash modality directly. Rules of this form usually describe how assertions in the logic change at a crash. Two such examples are Pc-persisted and Pc-points-to in Figure 4.7 on page 93. In addition to the rules of that form, is the Htr-idempotence rule. This rule also contains the post-crash modality, and it is the rule that links the modality to the crash execution step. In order for this rule to be sound, the state interpretation $\mathsf{stateInterp}$ used in the program logic and the nextgen modality must satisfy the following rule:

STATE-INTERP-NEXTGEN

$$\frac{\langle \sigma, \mathcal{P} \rangle \xrightarrow{\lightning} \langle \sigma', \mathcal{C}' \rangle}{\mathsf{stateInterp}(\langle \sigma, \mathcal{P} \rangle) \vdash \dot{\Rrightarrow} \langle \mathsf{NG} \rangle \dot{\Rrightarrow} \mathsf{stateInterp}(\langle \sigma', \mathcal{C}' \rangle).}$$

This rule says that if a machine configuration can take a crash step into a new machine configuration, then the state interpretation for the old machine configuration implies the state interpretation for the new machine configuration under the nextgen modality.

6.4 A Nextgen Modality With Picks

6.4.1 Why We Need Picks

As the reader is well-aware by now, the crash step in λ_{pmem} is non-deterministic. Recall that crash step is encoded by the relation $\xrightarrow{\lightning}$ which is generated by the single rule M-CRASH. We repeat the rule here for ease of reference:

$$\frac{\begin{array}{cc}\mathcal{P} \sqsubseteq \mathcal{C} & \text{consistent}(\sigma, \mathcal{C}) \\ \text{dom}(\sigma') = \text{dom}(\mathcal{C}) & \forall \ell \in \text{dom}(\mathcal{C}).\, \sigma'(\ell) = \{0 \mapsto \langle \sigma(\ell)(\mathcal{C}(\ell)).v, \bot, \bot, \bot \rangle\}\end{array}}{\langle \sigma, \mathcal{P} \rangle \xrightarrow{\lightning} \langle \sigma', \text{viewToZero}(\mathcal{C}) \rangle}$$

For resources in the logic, the non-determinism means that if a resource has a relationship to the physical state, then this resource might need to change in one of many ways at crash. How the resource needs to change is determined by the particular instance of the crash step rule M-CRASH chosen during the execution. In this rule, the source of the non-determinism is the crash view $\mathcal{C}$.

The above is evident in, for instance the points-to assertion in BaseSpirea. How this resource changes at a crash is captured by the rule PC-POINTS-TO where the existentially quantified crash view $\mathcal{C}$ encodes the non-determinism. As a simpler example, recall the $\text{crashedAt}(\mathcal{C})$ assertion that represents the crash view at the last crash. In BaseSpirea this assertion is modeled using a simple RA of agreement on views: AG(VIEW). The crashedAt assertion and the state interpretation contains the ownership:

$$\boxed{\text{ag}(\mathcal{C})}^{\gamma_{\text{cv}}}$$

At a crash this resource must by change, by the nextgen modality, into whatever the new crash view is. That is, for any crash view $\mathcal{C}'$, that satisfies the consistency constraints in M-CRASH, we must be able to show:

$$\boxed{\text{ag}(\mathcal{C})}^{\gamma_{\text{cv}}} \vdash \dot{\Rightarrow}\langle \text{NG} \rangle \dot{\Rightarrow} \boxed{\text{ag}(\mathcal{C}')}^{\gamma_{\text{cv}}}$$

Note that this is just the requirement STATE-INTERP-NEXTGEN from the previous section singled down to this one component of the state interpretation.

The purpose of picks is to support the non-determinism in λ_{pmem}. It does this by making it possible to *pick*, the transformation function that gets applied to ghost state such that it corresponds to the change at the crash step. This pick, or decision, is made *before* the nextgen modality is eliminated, but *after* having observed the crash step taken during the execution. Note that this matches the stated goal above: When proving STATE-INTERP-NEXTGEN the crash step has already informed us about what happened at the crash, and now we just need to ensure that our ghost state changes accordingly.

6.4.2 Rules for Picks

Having motivated the purpose of picks, we now cover the assertions and rules that this feature consists of.

The exclusive right to pick which transition gets applied to the ghost state at a ghost location γ at the change of generation is represented by the resource $\mathsf{token}(\gamma)$. This resource is created alongside a piece of ghost state at allocation time as seen in the rule for allocation GEN-ALLOC. The rule allocates the element a a new ghost location (this part is identical to the normal allocation rule in Iris) and provides the allocator with the assertion $\mathsf{token}(\gamma)$. We note that the dashed box notation is not identical to the usual dashed box notation, instead, it represents "generational ownership". This ownership behaves similarly to and satisfies the same rules as normal ownership in Iris, but the underlying model is different as it also needs to account for the additional bookkeeping needed for picks (such as the token resource).

The right afforded by ownership over a token is expressed in the rule TOKEN-PICK. The rule states that, given ownership over $\mathsf{token}(\gamma)$ one can pick any transformation t and, after an update, acquire two resources. The first resource, $\mathsf{usedToken}(\gamma)$, is similar to $\mathsf{token}(\gamma)$ except that the right to decide a transition function has now been expended. The second resource, $\mathsf{pickedOut}(\gamma, t)$, denotes the knowledge that the function t has been picked as the one that will be applied to the ghost location at γ going *out* of the current generation.

The $\mathsf{usedToken}(\gamma)$ resource can not be used for anything in the current generation. However, at the shift into the next generation a used token is turned back into an unused token as per USED-TOKEN-NEXTGEN. This means that at every generation a used token is "reset" such that the rule TOKEN-PICK can be applied anew to pick the function for the next generation.

As stated in the rule PICKED-OUT-NEXTGEN, if one has the knowledge $\mathsf{pickedOut}(\gamma, t)$ then in the next generation that becomes the knowledge $\mathsf{pickedIn}(\gamma, t)$. In other words, the picked transformation that points *out* of one generation points *in* to the next generation. Both pickedIn and pickedOut are persistent assertions, and they provide agreement in the sense expressed by PICKED-IN-AGREE and PICKED-OUT-AGREE. As such, these resources serve to share the knowledge about transition functions and conclude agreement as only one such function can exist.

The rule OWN-NEXTGEN shows how ownership is affected by generations. If one owns a for a ghost name γ then in the next generation there exists some transformation t and one now owns $t(a)$. The transformation t must necessarily have been the one picked for γ, and hence the rule also provides $\mathsf{pickedIn}(\gamma, t)$. Of course, without any further knowledge, nothing can be known about t as any function could have been picked with TOKEN-PICK. In the case where one knows exactly which function was picked before the next generation, the following rule can be shown:

$$\boxed{a}^{\gamma} * \mathsf{pickedOut}(\gamma, t) \vdash \langle \mathsf{NG} \rangle \mathsf{pickedIn}(\gamma, t) * \boxed{t(a)}^{\gamma}$$

This rule can easily be derived by combining OWN-NEXTGEN, PICKED-OUT-NEXTGEN, and PICKED-IN-AGREE; and demonstrates how these rule work in tandem. In this rule the existential in OWN-NEXTGEN has been eliminated as one has full information from the pickedOut assertion. As such, the existential in OWN-NEXTGEN does, in a sense, not

represent non-determinism. Instead, it comes from a lack of information: If one does not know which function was picked, then one only know that some (existentially quantified) function must have been used. This lack of non-determinism in the setup might seem a bit odd given the motivation we began with. The idea is not to have non-determinism about the picks, instead the idea is that the non-deterministic crash step determines what transformation function we pick. In this way the non-determinism in the crash step propagates into the picked functions, and ends up reflecting it. This should become more clear in Section 6.6 where we show exactly how this is done.

6.4.3 Using Picks For The Crash View Resource

We can now prove the rule for the crash view resource mentioned earlier.

We first extend the state interpretation such that it contains the token $\mathsf{token}(\gamma_c)$ for the crash view resource. This means that the state interpretation owns the right to decide how the crashed at resource should change at a crash.

The goal is then to show:

$$\boxed{\mathsf{ag}(\mathcal{C})}^{\gamma_c} * \mathsf{token}(\gamma_c) \vdash \dot{\Rrightarrow}\langle \mathsf{NG} \rangle \dot{\Rrightarrow} \boxed{\mathsf{ag}(\mathcal{C}')}^{\gamma_c} * \mathsf{token}(\gamma_c)$$

We do this, quite simply, by using TOKEN-PICK to choose the constant transformation $\lambda\_.\,\mathcal{C}'$. This is the only idea in the proof (and it is barely an idea), the rest follows trivially from the rules. Despite the triviality, this still show the general idea of how picks solves the problem stemming from the non-determinism: The crash step chooses some arbitrary consistent $\mathcal{C}'$ and our resources needs to change accordingly. To do this we take the specific $\mathcal{C}'$ into account and use it to pick a transformation based on it. This is only possible because the nextgen modality does not apply a fixed transformation to ghost locations, but instead lets that be a dynamic decisions driven by resources.

6.4.4 To Pick Or Not To Pick

Before we proceed to extend the above with promises we remark that one is never forced to pick a transformation function for every ghost name. An unused token is simply unchanged at a crash (see TOKEN-NEXTGEN). Requiring one to picks a transformation function for all ghost names would not be easy, among other things, because Iris is an affine separation logic and one could simply throw away the token resource. It also would not really be beneficial, as this extra obligation would only be a nuisance. It does however raise the question of what happens to a ghost location if no transformation is picked? From a user of the logic's point of view, the answer is that they will not be able to tell (as they will have no way to gain any information about the transformation). From the point of view of the model, the answer is that the identity function is used in the absence of any user-chosen transformation.

GEN-ALLOC
$$\frac{a \in \mathcal{V}}{\vdash \dot{\Rightarrow} \exists \gamma. \boxed{a}^{\gamma} * \mathsf{token}(\gamma)}$$

TOKEN-PICK
$$\mathsf{token}(\gamma) \vdash \dot{\Rightarrow} \mathsf{usedToken}(\gamma) * \mathsf{pickedOut}(\gamma, t)$$

PICKED-IN-AGREE
$$\mathsf{pickedIn}(\gamma, t_1) * \mathsf{pickedIn}(\gamma, t_2) \vdash t_1 = t_2$$

PICKED-OUT-AGREE
$$\mathsf{pickedOut}(\gamma, t_1) * \mathsf{pickedOut}(\gamma, t_2) \vdash t_1 = t_2$$

TOKEN-NEXTGEN
$$\mathsf{token}(\gamma) \vdash \langle \mathsf{NG} \rangle \mathsf{token}(\gamma)$$

USED-TOKEN-NEXTGEN
$$\mathsf{usedToken}(\gamma) \vdash \langle \mathsf{NG} \rangle \mathsf{token}(\gamma)$$

PICKED-OUT-NEXTGEN
$$\mathsf{pickedOut}(\gamma, t) \vdash \langle \mathsf{NG} \rangle \mathsf{pickedIn}(\gamma, t)$$

OWN-NEXTGEN
$$\boxed{a}^{\gamma} \vdash \langle \mathsf{NG} \rangle \exists t.\, \mathsf{pickedIn}(\gamma, t) * \boxed{t(a)}^{\gamma}$$

Figure 6.1: Rules for the nextgen modality with picks

6.5 Extending the Modality With Promises

We now extend the modality and the rules from the previous section with a feature we call *promises*.

Conventions and notation In the following we use the term *predicate* to mean a unary function of the type $A \to \textsc{Prop}$ for some A. We use the term *relation* to mean any n-ary function, where $n > 1$, of the type $A_1 \to \cdots \to A_n \to \textsc{Prop}$ for some $A_1, \ldots, A_n$. For any n-ary predicate or relation R_1 and R_2 we write $R_1 \subseteq R_2$ to mean that for all $a_1 \in A_1, \ldots, a_n \in A_n$ it is the case that $R_1(a_1, \ldots, a_n) \Rightarrow R_2(a_1, \ldots a_n)$.

6.5.1 Why We Need Promises

To a first approximation a promise is a predicate over transformation functions $P : (A \to A) \to \textsc{Prop}$ for a resource algebra A. For a ghost location γ, a promise P represents the *guarantee* that only transformations t that satisfy $P(t)$ will be picked for γ. This makes it possible to restrict what transformations can be applied to a ghost state, such that others can *rely* on the picked transformation satisfying the promise. Assuming that a promise P has been made for a ghost location γ, below is a preliminary sketch of what this would look like in terms of the rules:

TOKEN-PICK-SKETCH
$$P(t) * \mathsf{token}(\gamma) \vdash \dot{\Rightarrow} \mathsf{usedToken}(\gamma) * \mathsf{pickedOut}(\gamma, t)$$

OWN-NEXTGEN-SKETCH
$$\boxed{a}^{\gamma} \vdash \langle \mathsf{NG} \rangle \exists t.\, P(t) * \mathsf{pickedIn}(\gamma, t) * \boxed{t(a)}^{\gamma}$$

In the first rule, when a transformation is picked, one needs to satisfy the promise that P holds for the picked transformation. In the second rule, one can now rely on the promise being upheld and gain the knowledge that whatever transformation was picked satisfies $P(t)$.

We need promises in Spirea, as writes that have been flushed and synchronously fenced are guaranteed to have persisted. This means that certain things can be ensured to *not* happen at a crash. The simplest example of this is the rule PC-PERSISTED from BaseSpirea:

$$\mathrm{persisted}(\mathcal{P}) \vdash \langle \mathsf{PC} \rangle\, \mathrm{persisted}(\mathrm{viewToZero}(\mathcal{P})) * \exists \mathcal{C} \sqsupseteq \mathcal{P}.\ \mathrm{crashedAt}(\mathcal{C})$$

In this rule the assertion $\mathrm{persisted}(\mathcal{P})$ imposes a restriction on what $\mathcal{C}$ can be in the next generation. As we saw in the last section, the assertion $\mathrm{crashedAt}(\mathcal{C})$ in the next generation depends on what transformation is picked for its underlying resource. With the rules in the last section, any transformation can always be picked, and this makes it impossible to prove the above rule. There is no way to define the model of $\mathrm{persisted}$ such that it places any restrictions on the possible crash views in the next generation. Making this possible is the first problem that promises solve. Using promises, we can imagine that the assertion $\mathrm{persisted}(\mathcal{P})$ contains knowledge about a promise for the crashed view resource given as the predicate

$$\lambda t.\, \exists \mathcal{C}.\, (t = \lambda\_.\, \mathcal{C}) \wedge \mathcal{P} \sqsubseteq \mathcal{C}.$$

This promise states that the transformation picked for the crash view resource is a constant function of the form we described in the last section, as well as the requirement that the constant function returns a view that is greater than $\mathcal{P}$. With this knowledge we could hope to be able to be able to prove the above rule.

A second observation is that sometimes the changes that occur to one resource at a crash depends on what happens to some other resource. One example of this is the rule PC-POINTS-TO, reproduced here:

$$\ell \hookrightarrow_{\mathsf{h}} h \vdash \langle \mathsf{PC} \rangle\, \exists \mathcal{C}.\ \mathrm{crashedAt}(\mathcal{C}) * \left(\ell \notin \mathsf{dom}(\mathcal{C}) \vee \left(\exists t, m. \begin{array}{l} h(t) = m * \mathcal{C}(\ell) = t * m.\mathcal{P} \sqsubseteq \mathcal{C} * \\ \ell \hookrightarrow_h \{0 \mapsto \langle m.v, \bot, \bot, \bot \rangle\} \end{array} \right) \right)$$

Here we can see that the state of a points-to predicate after a crash depends on the state of the crash view after the crash. Phrased in terms of picks, the transformation that gets picked for the ghost heap (the resource underlying points-to predicates) must take into account the transformation that is picked for the crashed view resource. Allowing for dependencies of this sort is the second problem that promises address. To do this, promise are not simply predicates. They can, more generally, be relations between the transformation that is picked for a ghost location and the transformations that is picked for other ghost locations that it depend upon. This makes it possible to establish *dependencies* between different ghost locations. In Section 6.6 we show how dependencies can be used to validate the above rule. Here the promise for the ghost heap will be a relation between the transformations over the crashed view resource and the ghost heap.

6.5.2 What Is a Promise?

With the above motivation in mind, we now give the full definition of a promise.

Due to the dependencies between ghost locations that promises enable, whenever we want to use a RA A in the logic, we also need to specify the RAs that it depends upon as sequence of RAs $D_1, \ldots, D_n$. These "dependency RAs" are used in the definition of a promise below.

Definition 6.5.1 (Promise). *For a RA A and a sequence of RAs $D_1, \ldots, D_n$ a* promise *consists of*

- *an $(n+1)$-ary relation*

$$R : (D_1 \to D_1) \to \cdots \to (D_n \to D_n) \to (A \to A) \to \textsc{Prop},$$

- *a predicate*

$$P : (A \to A) \to \textsc{Prop},$$

- *and a sequence of predicates*

$$P_1 \in (D_1 \to D_1) \to \textsc{Prop}, \ldots, P_n \in (D_n \to D_n) \to \textsc{Prop}.$$

The data must satisfy the following conditions:

- *Whenever the relation holds for a sequence of transformations, it implies the predicate for the last transformation in the sequence.*

$$\begin{aligned}&\forall t_1 \in (D_1 \to D_1), \ldots, t_n \in (D_n \to D_n), t \in (A \to A).\\ &\quad R(t_1, \ldots, t_n, t) \Rightarrow P(t)\end{aligned}$$

- *Given a sequence of n transformations, that each satisfies the respective predicate in the list of predicates. There exists a transformation that, together with the given sequence, satisfies the relation.*

$$\begin{aligned}&\forall t_1 \in D_1 \to D_1, \ldots, t_n \in D_n \to D_n.\\ &\quad P_1(t_1) \wedge \cdots \wedge P_n(t_n) \Rightarrow \exists t.\, R(t_1, \ldots, t_n, t)\end{aligned}$$

The "meat" of a promise is the relation R. The predicate P is a practical convenience and not essential. However, it is often useful to know what one can rely on regarding the possible transformation for a given ghost location irrespective of how the transformation relates to the transformations for its dependencies. The predicate serves to make this more convenient. One could always pick P to be $\lambda t. \exists t_1, \ldots, t_n.\, R(t_1, \ldots, t_n, t)$, but for concrete promises P can often be chosen to be a simpler predicate that still captures the necessary information. The first condition in the definition states that $P(t)$ is always implied by R whenever R holds

for t and some transformations for the dependencies. This ensures that P satisfies the intended purpose described.

The utility of keeping P is evident already in the definition of a promise itself. The sequence of predicates $P_1, \ldots, P_n$ is exactly such promised predicates for each of the dependencies. Had we used relations for each of the dependency RAs the type of each relation R_i would have depended on the dependency's dependency RAs. This would have made the definition much more complicated. The use of predicates makes it possible to talk about what has been promised about dependencies while avoiding talking about the dependencies of the dependencies.

The sequence of predicates is used in the second condition. This condition ensures that a promise is not impossible to fulfill. If, for instance, it was possible to make a promise where $R(t_1, \ldots, t_n, t) \Rightarrow \mathsf{False}$ then the rules as a whole would be unsound. We thus demand that one can only make promises where one can offer evidence t that satisfies the promise. When giving this evidence one can assume transformations for each of the dependencies that satisfy the predicates that have been assumed for each of the dependencies.

If R_1 and P_1 is the relation and predicate for one promise and R_2 and P_2 is the relation and predicate for some other promise, we say that the former promise is *stronger* if $R_1 \subseteq R_2$ and $P_1 \subseteq P_2$.

6.5.3 Rules

As before the assertion token denotes the right to pick a transformation for a ghost location. It now additionally denotes ownership over the promise for the ghost location and the right to strengthen the promise. To accommodate for these additional purposes the assertion is extended to:

$$\mathsf{token}(\gamma, \vec{\gamma}, R, P)$$

Here R and P is a relation and a predicate corresponding to a promise as described in the definition of a promise. The parameter $\vec{\gamma}$ is a sequence $\gamma_1, \ldots, \gamma_n$ of ghost names for each dependency of the ghost location. The number of dependencies and their type of RAs is *static* information that is determined when instantiating Iris. However, the actual ghost locations that a ghost location depends on (*i.e.,* the ghost names) is decided when the ghost location is allocated. Allocation is done with the rule GEN-ALLOC where one must, as usual, pick a valid element a and one then gets ownership over a for a fresh ghost name γ. The ghost locations that the newly allocated ghost location should depend on is determined by instantiating the rule with the desired list of ghost names $\vec{\gamma}$. After allocation one gets a token of the form $\mathsf{token}(\gamma, \vec{\gamma}, \mathsf{True}_{n+1}, \mathsf{True})$. We use the notation True_i to denote an i-ary predicate that is always true:

$$\mathsf{True}_1 \triangleq \lambda\_.\, \mathsf{True} \qquad\qquad \mathsf{True}_i \triangleq \lambda\_.\, \mathsf{True}_{i-1}$$

In this case the relation has arity $n + 1$ corresponding to the transformations for the n dependencies and the one for the ghost location itself. The promise starts

out being trivially satisfied and can later be strengthened after allocation. For each dependency $i \in 1, \ldots, n$ one must provide the resource $\mathsf{rely}_{\mathsf{self}}(\vec{\gamma}_i, \vec{P}_i)$. We explain the $\mathsf{rely}_{\mathsf{self}}$ assertion below. In this rule its presence merely serves as a "sanity check" on the list of ghost names to ensure that they actually correspond to allocated ghost names. Had some requirement of this sort not been in place, one could have applied the rule with arbitrary nonsensical ghost names. As such the content of $\mathsf{rely}_{\mathsf{self}}$ does not matter, any assertion for the dependency ghost names proving that they exist could have been used.

Observe, that since a ghost location can only depend on other ghost locations that where allocated prior to its allocation, there can be no cycles in the graph of dependencies. This is crucial in order for the rules to be sound.

As mentioned, $\mathsf{token}(\gamma, \vec{\gamma}, R, P)$ denotes ownership over a promise and the right to strengthen it. Since promises can only be replaced with a stronger promise, the current promise can never be invalidated. We can therefore use the rule TOKEN-TO-RELY to extract a persistent assertion $\mathsf{rely}(\gamma, \vec{\gamma}, R, P)$ from the token. The rely assertion only denotes that a promise with R and P has been made, but not that this is the most recent and strongest promise that has been made. Since the token assertion always contains the most recent promise, any relation in rely must be implied by the relation in token as per TOKEN-RELY-COMBINE. Furthermore, since promises can only increase in strength, if one has two rely assertions then the relation in one of them must be stronger than the other as per RELY-RELY-COMBINE. If one cares only about the predicate promised for a ghost location γ, and not about the relation, one can use the rule RELY-TO-RELY-SELF which results in the resource $\mathsf{rely}_{\mathsf{self}}(\gamma, P)$. This resource is similar to rely expect it discards any information pertaining the dependencies for γ.

A central rule is PROMISE-STRENGTHEN. This rule strengthens a promise of R_1 and P_1 into a new promise of R_2 and P_2. The rule is quite a mouthful, so we cover each hypothesis in turn:

1. $R_2 \subseteq R_1$. First of all, the new relation must be stronger than the previous relation. This, after all, is why the rule is a *strengthening* rule.

2. $P_2 \subseteq P_1$. Similarly, the new predicate must be stronger than the previous predicate.

3. $\forall t_1, \ldots, t_n, t. R_2(t_1, \ldots, t_n, t) \Rightarrow P_2(t)$. For a promise, the relation must always imply the predicate. This is the first requirement in the definition of a promise. Hence, this must be shown.

4. $\forall i. \mathsf{rely}_{\mathsf{self}}(\vec{\gamma}_i, P_i)$. The rule is parameterized over the predicates $P_1, \ldots, P_n$ assumed to have been promised for each of the dependencies. Hence, for every $i \in 1, \ldots, n$ one must show that the predicate P_i has in fact been promised.

5. $\forall t_1, \ldots, t_n. P_1(t_1) \wedge \cdots \wedge P_n(t_n) \Rightarrow \exists t. R_2(t_1, \ldots, t_n, t)$. Finally, one must show that the promised relation can in fact be satisfied by some list of transformation. This is the second requirement from the definition of a promise.

Once all the above hypotheses have been shown one can strengthen a promise.

The rule for picking a transformation, TOKEN-PICK, is quite similar to before. But it is now parameterized by a sequence of transformations for the dependencies $t_1, \ldots, t_n$ and has two additional hypotheses. The first hypothesis requires a pickedOut resource for the transformation for every dependency. This effectively ensures that one can only pick a transformation for a ghost location once transformations have already been picked for all of its dependencies. The second hypothesis demands that the chosen transformation along with the dependency transformations satisfy the promised relation. This is the place in the rules where the guarantee offered by the promise is to be respected.

The rule OWN-NEXTGEN is exactly as before. This is because normal ownership contains no information about promises, and hence the rule does not make use of them. The rule RELY-NEXTGEN for rely is where the promise provides useful guarantees. The rule states that in a next generation there will exist transformations for all dependencies as well as a transformation for the resource itself. For each of these transformations a pickedIn assertion is obtained and all the transformations together satisfy the promised relation.

Promises themselves are unchanged at a crash. The rules TOKEN-NEXTGEN and USED-TOKEN-NEXTGEN show how the promised relation and predicate are carried unchanged into the next generation. This means that promises are "cross-generational" and can only ever increase in strength.

Note that usedToken can not be used to strengthen a promise. That is, after a transformation has been picked one can not strengthen the promise further until the next generation where the used token is reverted back into an unused token.

6.6 A Generation-Aware State Interpretation for BaseSpirea

We now show how the nextgen modality with promises can be used to model the state interpretation and the assertions in BaseSpirea. This is a significant step toward the goal of a variant of BaseSpirea that does not use the post-crash modality, but instead uses the nextgen modality with promises. To do this we must come up with generational ghost state that can support the assertions in BaseSpirea and define a state interpretation that makes use of this ghost state. Our generational ghost state should be able to account for the way in which the physical state can change non-deterministically at a crash. Our goal is to come up with generational resources that allow us to prove all the same rules for BaseSpirea that we saw in Section 4.5. To do this we have to use both the ability to pick transformation functions and the possibility of making promises.

The model of BaseSpirea uses the following four different resources:

1. A resource for the crash view: $\textsc{Ag}(\textsc{View})$. That is, simple agreement on the last crash view. This resource is used in the model of the assertion $\mathrm{crashedAt}(\mathcal{C})$

GEN-ALLOC

$$\frac{a \in \mathcal{V}}{(\forall i.\, \mathsf{rely}_{\mathsf{self}}(\vec{\gamma}_i, P_i)) \vdash \dot{\Rrightarrow} \exists \gamma.\, \boxed{a}^{\gamma} * \mathsf{token}(\gamma, \vec{\gamma}, \mathsf{True}_{n+1}, \mathsf{True}_1)}$$

TOKEN-PICK

$$\frac{\forall i.\, \mathsf{pickedOut}(\vec{\gamma}_i, t_i) \qquad R(t_1, \ldots, t_i, t)}{\mathsf{token}(\gamma, \vec{\gamma}, R, P) \vdash \dot{\Rrightarrow} \mathsf{usedToken}(\gamma, \vec{\gamma}, R, P) * \mathsf{pickedOut}(\gamma, t)}$$

PROMISE-STRENGTHEN

$$\frac{\begin{array}{c} R_2 \subseteq R_1 \qquad P_2 \subseteq P_1 \qquad \forall t_1, \ldots, t_n, t. R_2(t_1, \ldots, t_n, t) \Rightarrow P_2(t) \\ \forall i. \mathsf{rely}_{\mathsf{self}}(\vec{\gamma}_i, P_i) \qquad \forall t_1, \ldots, t_n. P_1(t_1) \wedge \cdots \wedge P_n(t_n) \Rightarrow \exists t. R_2(t_1, \ldots, t_n, t) \end{array}}{\mathsf{token}(\gamma, \vec{\gamma}, R_1, P_1) \vdash \dot{\Rrightarrow} \mathsf{token}(\gamma, \vec{\gamma}, R_2, P_2).}$$

TOKEN-NEXTGEN

$$\mathsf{token}(\gamma, \vec{\gamma}, R, P) \vdash \langle \mathsf{NG} \rangle \mathsf{token}(\gamma, \vec{\gamma}, R, P)$$

USED-TOKEN-NEXTGEN

$$\mathsf{usedToken}(\gamma, \vec{\gamma}, R, P) \vdash \langle \mathsf{NG} \rangle \mathsf{token}(\gamma, \vec{\gamma}, R, P)$$

PICKED-OUT-NEXTGEN

$$\mathsf{pickedOut}(\gamma, t) \vdash \langle \mathsf{NG} \rangle \mathsf{pickedIn}(\gamma, t)$$

OWN-NEXTGEN

$$\boxed{a}^{\gamma} \vdash \langle \mathsf{NG} \rangle \exists t.\, \mathsf{pickedIn}(\gamma, t) * \boxed{t(a)}^{\gamma}$$

RELY-NEXTGEN

$$\mathsf{rely}(\gamma, \vec{\gamma}, R, P) \vdash \langle \mathsf{NG} \rangle \mathsf{rely}(\gamma, \vec{\gamma}, R, P) * \exists t_1, \ldots, t_n, t.\, R(t_1, \ldots, t_n, t) * \mathsf{pickedIn}(\gamma, t) * (\forall i.\, \mathsf{pickedIn}(\vec{\gamma}_i, t_i))$$

RELY-SELF-NEXTGEN

$$\mathsf{rely}_{\mathsf{self}}(\gamma, P) \vdash \langle \mathsf{NG} \rangle \mathsf{rely}_{\mathsf{self}}(\gamma, P) * \exists t.\, P(t) * \mathsf{pickedIn}(\gamma, t)$$

TOKEN-RELY-COMBINE

$$\mathsf{token}(\gamma, \vec{\gamma}_1, R_1, P_1) * \mathsf{rely}(\gamma, \vec{\gamma}_2, R_2, P_2) \vdash \vec{\gamma}_1 = \vec{\gamma}_2 * R_1 \subseteq R_2$$

RELY-RELY-COMBINE

$$\mathsf{rely}(\gamma, \vec{\gamma}_1, R_1, P_1) * \mathsf{rely}(\gamma, \vec{\gamma}_2, R_2, P_2) \vdash \vec{\gamma}_1 = \vec{\gamma}_2 * (R_1 \subseteq R_2 \vee R_2 \subseteq R_1)$$

TOKEN-TO-RELY

$$\mathsf{token}(\gamma, \vec{\gamma}, R, P) \vdash \mathsf{rely}(\gamma, \vec{\gamma}, R, P)$$

RELY-TO-RELY-SELF

$$\mathsf{rely}(\gamma, \vec{\gamma}, R, P) \vdash \mathsf{rely}_{\mathsf{self}}(\gamma, P)$$

Figure 6.2: Rules for the nextgen modality with picks and promises

that makes it possible to know the value of the crash view at the last crash.

2. A resource for the persist view: Auth(View). This resource is used in the model of the assertion $\mathrm{persisted}(\mathcal{P})$ that asserts a lower-bound on the persist view by using a fragment. The authoritative part of the RA is in the state interpretation.

3. A resource for the heap using a standard Iris RA construction for heaps: gmapView(Loc, History). This resource is used in the model of points-to predicates.

4. A resource for the maximum possible store view: Auth(View). This resource is used in the model of $\mathrm{valid}(\mathcal{S})$.

6.6.1 Challenges

To begin with, we consider the first two of these, that is the resources needed to represent the *crash view* and the *persist view*. It turns out, that once we are able to handle these two resources, the remaining follow along similar lines. Hence, focusing on these simplifies the presentation while preserving the key insights. After we have covered these two resources we sketch how to represent the heap. Since the resource for the maximum store view has a trivial interaction with crashes we do not cover it.

Recall that the behavior at a crash is encoded by the relation $\xrightarrow{\lightning}$ which is generated by the single crash step defined by the rule M-crash. We repeat the rule here for ease of reference:

$$\frac{\begin{array}{cc}\mathcal{P} \sqsubseteq \mathcal{C} & \mathsf{consistent}(\sigma, \mathcal{C}) \\ \mathsf{dom}(\sigma') = \mathsf{dom}(\mathcal{C}) & \forall \ell \in \mathsf{dom}(\mathcal{C}).\, \sigma'(\ell) = \{0 \mapsto \langle \sigma(\ell)(\mathcal{C}(\ell)).v, \bot, \bot, \bot\rangle\}\end{array}}{\langle \sigma, \mathcal{P}\rangle \xrightarrow{\lightning} \langle \sigma', \mathrm{viewToZero}(\mathcal{C})\rangle}$$

Recall that the view $\mathrm{viewToZero}(\mathcal{C})$ is the view with the same domain as $\mathcal{C}$ but with every value being zero. Recall also that the crash view is the view $\mathcal{C}$ in the rule and the persist view is the second component $\mathcal{P}$ in the machine configuration (*i.e.,* the physical state). Looking at M-crash we can identify two key challenges that we face when representing these views and the crash step with generational ghost state:

1. The first challenge is that $\mathcal{C}$ and $\mathcal{P}$ are *interdependent*. The condition $\mathcal{P} \sqsubseteq \mathcal{C}$ makes the crash view dependent on the persist view. And as the persist view after the crash changes into $\mathrm{viewToZero}(\mathcal{C})$ the persist view also depends on the crash view.

2. The second challenge stems from the fact that the crash view and the persist view do not grow monotonically. Promises, on the other hand, can only increase in strength and they persist across generations. As such, if we establish a promise that states a lower bound on $\mathcal{C}$ in the next generation (corresponding

to $\mathcal{P} \sqsubseteq \mathcal{C}$) then this promise will still exist in the next generation but it will no longer be meaningful.

A key rule relating the two assertions is PC-PERSISTED:

$$\text{persisted}(\mathcal{P}) \vdash \langle \mathsf{PC} \rangle \, \text{persisted}(\text{viewToZero}(\mathcal{P})) * \exists \mathcal{C} \sqsupseteq \mathcal{P}.\ \text{crashedAt}(\mathcal{C})$$

This rule internalizes the role of $\mathcal{P}$ in the crash step M-CRASH and shows how the crash view depends on the persist view.

6.6.2 Resource For The Crash View

Since we want to make promises about the crash view, and since promises can only grow in strength, we design the resource for the crash view such that it only grows monotonically. This is unlike the operational semantics, where the crash view from one crash to the next have no relation. We do this by storing a view $\mathcal{SC}$ that we call the *summed crash view*. This view is the sum of all crash views, *i.e.,* before any crash it is $\bot$ and after n crashes with the crash views $\mathcal{C}_1, \ldots, \mathcal{C}_n$ it is $\mathcal{C}_1 + \ldots, + \mathcal{C}_n$.[2] This clearly ensures that the $\mathcal{SC}$ is monotinically increasing. In addition to the $\mathcal{SC}$ we introduce another view $\mathcal{O}$ called the *offset view*. This view is the sum of all crash views *except* for the last crash view. The key utility of this view is that if the last crash view is $\mathcal{C}$ then $\mathcal{SC} - \mathcal{O} = \mathcal{C}$. In other words, from $\mathcal{SC}$ and $\mathcal{O}$ we can recover the crash view.

To store these two views we use the following RA for the crash view:

$$\textsc{CrashedAt} \triangleq \textsc{Ag}(\textsc{View}) \times \textsc{Ag}(\textsc{View})$$

The first component in the pair is the offset view and the second is the summed crash view.

The crash view resource has no dependencies, but we will want to make promises about the crash view resource that state a lower bound on what it will grow to in the next generation. The transition functions we use have the following form:

$$t_C(\mathcal{SC}') \triangleq \lambda \langle \_, \mathcal{SC} \rangle.\, \langle \mathcal{SC}, \text{ag}(\mathcal{SC}') \rangle$$

Here $\mathcal{SC}'$ is the new summed crash view, and the transition function moves the previous summed crash view into the offset view. To give a lower bound $\mathcal{L}$ on what the next summed crash view should be, the following predicate is used:

$$P_C(\mathcal{L})(t) \triangleq \exists \mathcal{SC}.\, \mathcal{L} \sqsubseteq \mathcal{SC} \wedge t = t_C(\mathcal{SC})$$

That is, a promise with the predicate $\mathcal{P}_C(\mathcal{L})$ guarantees that the transition with be of the form $t_C(\mathcal{SC})$ for an $\mathcal{SC}$ that is greater than $\mathcal{L}$.

Using this resource we can give a model for the crashedAt assertion:

[2] Addition and subtraction on views is given as point-wise addition or subtraction of the timestamps in the views.

$$\begin{aligned}\mathrm{crashedAt}(\mathcal{C}) \triangleq \exists \mathcal{O}, \mathcal{SC}, \mathcal{L}.\, \mathcal{SC} - \mathcal{O} = \mathcal{C} * \\ \boxed{(\mathsf{ag}(\mathcal{O}), \mathsf{ag}(\mathcal{SC}))}^{\gamma_C} * \\ \mathsf{rely}_{\mathsf{self}}(\gamma_C, P_C(\mathcal{L}))\end{aligned}$$

The first line ensures that the crash view corresponds to $\mathcal{SC}$ and $\mathcal{O}$ as it should. The next line asserts ownership over the two views where the agreement ensures that the existentially quantified views are the right ones. The last line states that a promise has been made for the crashed at resource with a lower bound of $\mathcal{L}$. As the view $\mathcal{L}$ is existentially quantified no guarantees are actually given, but keeping the resource is convenient. Note, that as the crash view resource have no dependencies rely and $\mathsf{rely}_{\mathsf{self}}$ are almost identical in meaning.

6.6.3 Resource For The Persist View

For the persist view we use the following resource:

$$\textsc{Persisted} \triangleq \textsc{Auth}(\textsc{View})$$

As is common, the state interpretation contains the authoritative part of the RA and the $\mathrm{persisted}$ assertion contains fragments. Following the same rationale as for the crash view resource, we store a *summed persist view* $\mathcal{SP}$ in the ghost state. The non-summed persist view $\mathcal{P}$ can be recovered per the equality $\mathcal{P} = \mathcal{SP} - \mathcal{SC}$.

The persist view resource depends on the crash view resource. Specifically, the persist view resource has 1 dependency which is a RA of the type CrashedAt. The following relation will be promised for the resource:

$$\begin{aligned}R_P(t_C, t_P) \triangleq \exists \mathcal{SC}.\, t_C = t_C(\mathcal{SC}) \wedge \\ t_P = \mathrm{fmap\_auth}(\lambda\_.\, \mathcal{SC}).\end{aligned}$$

The relation states that there exists some summed crash view and that the transformation for the dependency (*i.e.,* for the crashed at resource) has a form that can be described based on this view. The transformation for the crashed at resource then applies a constant function to the authoritative RA. The function fmap_auth is defined such that

$$\mathrm{fmap\_auth}(t)(\bullet a) \triangleq \bullet t(a) \qquad \mathrm{fmap\_auth}(t)(\circ a) \triangleq \circ t(a)$$

This promise is never strengthened and is fixed for the lifetime of the persist view resource. However, due to the dependency on the crash view resource, any promise about the crash view resource transitively affects the persisted resource.

We can now give a model for the $\mathrm{persisted}$ assertion.

$$\begin{aligned}\mathrm{persisted}(\mathcal{P}) \triangleq \exists \mathcal{SC}, \mathcal{SP}, \mathcal{O}. \\ \mathcal{SP} - \mathcal{SC} = \mathcal{P} * \\ \boxed{(\mathsf{ag}(\mathcal{O}), \mathsf{ag}(\mathcal{SC}))}^{\gamma_C} * \mathsf{rely}_{\mathsf{self}}(\gamma_C, P_C(\mathcal{SP})) * \\ \boxed{\circ \mathcal{SP}}^{\gamma_P} * \mathsf{rely}(\gamma_P, [\gamma_C], R_P, \mathsf{True}_1)\end{aligned}$$

The definition states the existence of a summed crash view, a summed persist view, and an offset view. The persist view is equal to subtracting the summed crash view from the persist view. We own agreements for the crash view resource and a fragment of the summed persist view. On the last line we have a promise about the crash view resource. This promise ensures that that summed crash view in the next generation will be at last $\mathcal{SP}$. Next to this is the promise for the persist view resource.

Having defined the model of $\mathrm{crashedAt}$ and $\mathrm{persisted}$ we can now prove soundness of the rule PC-PERSISTED but with the post-crash modality replaced by the nextgen modality:

Lemma 6.6.1. *The rule*

$$\mathrm{persisted}(\mathcal{P}) \vdash \langle \mathit{NG} \rangle\, \mathrm{persisted}(\mathrm{viewToZero}(\mathcal{P})) * \exists \mathcal{C} \sqsupseteq \mathcal{P}.\ \mathrm{crashedAt}(\mathcal{C})$$

is sound.

Proof. We have the resources in the definition of $\mathrm{persisted}$ for some $\mathcal{SC}, \mathcal{SP}$, and $\mathcal{O}$. For both of the ghost overship assertions we have a corresponding rely assertion. The goal contains a nextgen modality and to introduce this modality[3] we must observe how our resources changes by the nextgen modality. This is described by the rules OWN-NEXTGEN, RELY-NEXTGEN, and RELY-SELF-NEXTGEN. After using these rules to introduce the nextgen modality and after using PICKED-IN-AGREE to identify the transformations for the same ghost location we have:

$$\boxed{(\mathsf{ag}(\mathcal{SC}), \mathsf{ag}(\mathcal{SC}'))}^{\gamma_C} * \mathsf{rely}_{\mathsf{self}}(\gamma_C, P_C(\mathcal{SP})) *$$
$$\boxed{\circ \mathcal{SC}'}^{\gamma_P} * \mathsf{rely}(\gamma_P, [\gamma_C], R_P, \mathsf{True}_1)$$

Here $\mathcal{SC}'$ is the new summed crash view which satisfies $\mathcal{SP} \sqsubseteq \mathcal{SC}'$. This crash view is acquired from the promise for the crashed at resource. The rely resources themselves are unchanged.

To prove the goal we provide $\mathrm{persisted}(\mathrm{viewToZero}(\mathcal{P}))$ and $\exists \mathcal{C} \sqsupseteq \mathcal{P}.\ \mathrm{crashedAt}(\mathcal{C})$.

To prove $\mathrm{persisted}(\mathrm{viewToZero}(\mathcal{P}))$ we provide the following for the three views for the existential quantifiers in $\mathrm{persisted}$: $\mathcal{SC}', \mathcal{SP}, \mathcal{SC}$. With this choice of views most of the goal can simply be frame away. What is left is to prove $\mathcal{SP} - \mathcal{SC}' = \mathrm{viewToZero}(\mathcal{P})$ and $\boxed{\circ \mathcal{SC}'}^{\gamma_P}$. The former holds due to how subtraction is defined. The view $\mathcal{SC}'$ is greater than $\mathcal{SP}$ and since we use subtraction on natural numbers where $n - m = 0$ if $n \leq m$ and a subtraction for views that preserves the domain of the first argument, the result is that the equality holds.

To prove $\exists \mathcal{C} \sqsupseteq \mathcal{P}.\ \mathrm{crashedAt}(\mathcal{C})$ We provide $\mathcal{SC}' - \mathcal{SC}$ for the existential. We must show that $\mathcal{P} \sqsubseteq \mathcal{SC}' - \mathcal{SC}$. Per the equality in $\mathrm{persisted}$ we have $\mathcal{P} = \mathcal{SP} - \mathcal{SC}$ and the above inclusion then holds because $\mathcal{SP} \sqsubseteq \mathcal{SC}'$. Proving the $\mathrm{crashedAt}$ assertion itself is trivial. □

[3] By "introduce the modality" we mean putting the resources in our context under the nextgen modality using rules for this, combining the resources in under a single nextgen, and then applying monotonicity to arrive at a goal without a nextgen modality. All of this is what the `iModIntro` tactic in IPM carries out.

6.6.4 The State Interpretation

We can now define a state interpretation that covers the persist view and the crash view.

$$
\begin{aligned}
&S(\langle \sigma, \mathcal{P} \rangle) \triangleq \exists \mathcal{O}, \mathcal{SC}. \\
&\quad \boxed{(\mathsf{ag}(\mathcal{O}), \mathsf{ag}(\mathcal{SC}))}^{\gamma_C} * \mathsf{token}(\gamma_C, [], P_C(\mathcal{SC}+\mathcal{P}), P_C(\mathcal{SC}+\mathcal{P})) * \\
&\quad \boxed{\bullet \mathcal{SC} + \mathcal{P}}^{\gamma_P} * \mathsf{rely}(\gamma_P, [\gamma_C], R_P, \mathsf{True}_1)
\end{aligned}
$$

We keep the token for the crash view resource such that we can choose a transformation that matches the new crash view after a crash. Since the transformation for the persist view resource is fully determined by the crash view we do not need to store a token for that resource (in fact the token can be discarded once the promise for it has been established).

We can now prove the key result for the state interpretation and the nextgen modality.

Theorem 6.6.2. *If a machine configuration can take a crash step into a new machine configuration*

$$\langle \sigma, \mathcal{P} \rangle \xrightarrow{\lightning} \langle \sigma', \mathcal{C}' \rangle.$$

Then the state interpretation for the old machine configuration implies the state interpretation for the new machine configuration under update modalities and a nextgen modality

$$S(\langle \sigma, \mathcal{P} \rangle) \vdash \dot{\Rrightarrow} \langle \mathsf{NG} \rangle \dot{\Rrightarrow} S(\langle \sigma', \mathcal{C}' \rangle).$$

Proof. From the definition of the crash step we know that the step must be an instance of the single rule for the crash step. This means that there exists a crash view $\mathcal{C}$ as in M-CRASH. We use the token for γ_C to pick the transformation function $t_C(\mathcal{SC}+\mathcal{C})$ using the rule TOKEN-PICK. The remainder of the proof is fairly straightforward using the rules that we have seen. □

6.6.5 Resource For The Heap

We now briefly sketch how to model the heap using generational ghost state. As mentioned, the heap uses the standard RA construction in Iris for modelling heaps:

$$\textsc{Heap} \triangleq \textsc{gmapView}(\textsc{Loc}, \textsc{History})$$

We do not need to change the RA used for the heap, but, similarly to what we did for the crash view and the persist view, the way we use the RA is slightly different. Where the physical history for each location only stores the message that was recovered at the last crash and any messages written since the crash, in the ghost state we will store histories that contain all messages written across crashes that where not lost at a crash.

To make the above a bit more clear, consider an example where the history for a location ℓ is $\{0 \mapsto v_1, 4 \mapsto v_2, 6 \mapsto v_3\}$ during an execution prior to any crashes. If, at a crash, the second value is recovered and the last value is lost, then the crash view $\mathcal{C}$ for ℓ satisfies $\mathcal{C}(\ell) = 4$ and the history after the crash step is $\{0 \mapsto v_2\}$. In the ghost state however, we only drop the suffix corresponding to the lost writes, and we keep the timestamps as it. Thus, in the ghost state we have the history $\{0 \mapsto v_1, t \mapsto v_2\}$. The effect is then that the ghost state for the history only grows monotonically and that it maintains a correspondence to the summed crash view and the summed persist view. Among other things, the real physical history can be recovered from this logical one by dropping a prefix from every history corresponding to the summed crash view.

For the heap ghost state we make a promise that is based upon the exact same ideas as the one we used for the persist view.

$$\begin{aligned} R_H \triangleq \exists t_C, t_H, \exists \mathcal{SC}.& \\ & t_C = t_C(\mathcal{SC}) \wedge \\ & t_H = \lambda h.\, \mathsf{mapEntryLiftGmapView}(\mathsf{dropAboveHist}(\mathcal{SC}, h)). \end{aligned}$$

On the second line we establish a correspondence with the transformation for the crashed at view which ensures that $\mathcal{SC}$ is the correct view On the third line we use $\mathcal{SC}$ to construct the transformation for the function. We do not show the definitions of the two functions used here, but the result corresponds exactly to the explanation above: all writes in the history that are not included in $\mathcal{SC}$ are dropped from the history.

With these ideas in place extended the state integration for the heap is not too difficult, and the full details can be found in Coq.

6.7 Model

We now give an overview over the model of the nextgen modality with promises and picks. The full model and soundness proof of its rules is very intricate, so here we only, in broad strokes, highlight some of the most interesting key ideas. The presentation here should also serve as a useful primer for readers who want to study the full Coq implementation of the modality.

6.7.1 Meta Ghost State

With the nextgen modality in this chapter, every ghost location now contains promises and picks in addition to the actual ghost state it stores. To handle this data, every ghost location meant for the nextgen modality stores *meta resources*. More concretely, if a user of the logic wishes to use a RA A that depends on $D_1, \ldots, D_n$ the actual RA that is used under the hood is $\mathsf{GenerationalRA}(A, [D_1, \ldots, D_n])$. This construction is of the form

$$\mathsf{GenerationalRA}(A, [D_1, \ldots, D_n]) \triangleq A^? \times \textsc{Meta}(A, [D_1, \ldots, D_n])^?.$$

Hence, an element of this RA can contain an element of the RA A and/or element of the metadata. The construction $\textsc{Meta}(A, [D_1, \ldots, D_n])$ contains all the resources needed to represent picks and promises. Some of RAs used are fairly typical, for instance, picks use the agreement RA and the promises use a prefix list RA to store the increasing promises. However, it also uses an interesting generational RA that we think have wider applications in the context of nextgen modalities, and hence we describe this construction in detail.

The construction GenNC is parameterized over any camera A and is defined as:

$$\textsc{GenNC}(A) \triangleq A^? \times A^?$$

As such, the elements of the RA contains, potentially, two elements of the underlying RA. The first element in the pair is known as the element for the *next* generation and the second element is for the *current* generation. We use the following definitions to denote ownership over an $a \in A$ in the next and current generation, the next generation only, and the current generation only.

$$\text{gNC}(a) \triangleq (a, a) \qquad \text{gN}(a) \triangleq (a, \bot) \qquad \text{gC}(a) \triangleq (\bot, a)$$

Note that this satisfies

$$\text{gNC}(a) = \text{gN}(a) \cdot \text{gC}(a).$$

We now get to the thing that makes this construction a *generational* RA—the intended transformation.

$$t((a_g, a_c)) \triangleq (a_g, a_g)$$

The transformation simply preserves the next generation element and copies it into the current generation element. This has the consequence that, at a new generational, the current element disappears and the next generation element becomes the new current element. Notice that as the next generational element is unchanged it becomes a sort of "cross-generational" or "permanent" element. For instance, if the underlying RA is an exclusive token

$$\mathsf{token} = \mathsf{ex}(()) \in \text{Ex}(1)$$

then one can split the element $\text{gNC}(\mathsf{token})$ into $\text{gN}(\mathsf{token})$ and $\text{gC}(\mathsf{token})$. One can then "give up" the token for the current generation, $\text{gC}(\mathsf{token})$, and then get it back in the next generation by keeping $\text{gN}(\mathsf{token})$.

While simple this construction is very useful. We now show how it can be used to create a generational variant of the well known one-shot resource algebra. Recall that the one-shot resource algebra is defined as

$$\textsc{OneShot}(V) \triangleq \textsc{Ex}() + \textsc{Ag}(V)$$

where the exclusive left injection denotes the right to make the decision (or shoot) and the duplicable right injection denotes the knowledge that a decision has been made. We can now simply combine GenNC with OneShot as $\textsc{GenNC}(\textsc{OneShot}(V))$ and

arrive a generational variant. With this combination the element $\mathrm{gN}(\mathsf{inl}(1))$ represents the right to shoot across generations and $\mathrm{gC}(\mathsf{inl}(1))$ the right to shoot in the current generation. By never firing the former element, a single decision (or shot) can be carried out every generation. Sticking to the gun metaphor, we might say that the gun is reloaded at every generation, and we have a "one-shot-per-generation" RA. This is exactly what we use in the RA for picks where a pick can be made exactly once per generation.

6.7.2 The Model of $\langle \mathbf{NG} \rangle$

One of the key ideas of the nextgen modality is to tie the transformation function to resources. Since the transformation function applies point-wise to individual ghost names, it is constructed from a *transmap*, a map of transformations as defined back in Section 5.3.6:

$$TM \triangleq \prod_{i \in I} \textsc{GName} \xrightarrow{\text{fin}} (M_i \to M_i)$$

The first step in the model of the nextgen modality is to universally quantify over a transmap and construct the transformation function from this

$$\langle \mathsf{NG} \rangle P \triangleq \forall tm \in TM.\ \Rightarrow^{\mathsf{buildTrans}(tm)} P.$$

Here buildTrans turns the transmap into a transformation function. The reader can think of it as simply building a transformation that applies the transformations in the map point-wise as in Section 5.3.6, but for technical reasons the actual construction is slightly more complicated.

With the transmap universally quantified nothing can be known about the transformation function except the fact that it maintains the structure of the global RA. Consider the $\mathsf{pickedOut}(\gamma, t)$ assertion that contains *partial* information about the global transformation function. It essentially restricts its behavior when applied to an element of the global RA that contains an entry at i and γ (where $i \in I$ the index of the RA in the list of globally available RAs). To support this we extend the definition with an existentially quantified transmap:

$$\begin{aligned}\langle \mathsf{NG} \rangle P \triangleq\ & \exists\, picks \in TM. \\ & \mathsf{ownPicks}(picks) * \\ & \forall tm \in TM.\ picks \subseteq tm \mathrel{-\!\!*} \Rightarrow^{\mathsf{buildTrans}(tm)} P.\end{aligned}$$

The existential makes it possible to choose a smaller transmap *picks* that the "full" transmap *tm* should be an extension of. This is expressed by the condition $picks \subseteq tm$ which means that every entry defined in *picks* is defined in *tm* with the same value. Now P only needs to be proven for such a transformation function that *picks* gives certain knowledge about. The price to pay for choosing a *picks* is that the resource $\mathsf{ownPicks}(picks)$ is demanded by the definition. This resource essentially amounts to having pickedOut for every entry in *picks*.

With the above addition we have the machinery for supporting the assertions for picks, but we still need to extend the model further for promises. We do this with a second existentially quantified variable for the promises:

$$\begin{aligned}\langle \mathsf{NG}\rangle P \triangleq \exists &\mathit{picks} \in \mathit{TM},\ \mathit{promises} \in \textsc{List}(\textsc{PromiseAt}).\\ &\mathsf{ownPicks}(\mathit{picks}) * \mathsf{ownPromises}(\mathit{promises}) *\\ &\forall \mathit{tm} \in \mathit{TM}.\\ &\quad \mathit{picks} \subseteq \mathit{tm} * \mathsf{respectsPromises}(\mathit{tm}, \mathit{promises}) \twoheadrightarrow \Rightarrow^{\mathsf{buildTrans}(\mathit{tm})} P.\end{aligned}$$

Now a list of promises can also be chosen. The name PromiseAt denotes a type that contains both a promise (as defined in Definition 6.5.1) and a path for the ghost location that the promise is for (an index $i \in I$ and a ghost name $\gamma \in$ GName). Similarly to for picks, one must provide a resource for the chosen promises $\mathsf{ownPromises}(\mathit{promises})$ which roughly amounts to having $\mathsf{rely}_{\mathsf{self}}$ for every promise in the list. Now the final *tm* is guaranteed to *respect* the promises ($\mathsf{respectsPromises}(\mathit{tm}, \mathit{promises})$), which means that *tm* satisfies every promised relation and predicate in *promises*. The promises are stored in a list because promises can depend on each other. The list of promises needs to be *well-formed*. This means, among other things, that promises in the list are ordered such that every promise is before any promises it depends upon. Furthermore, a specific ghost location can only have one promise for it in the list and a promise can only be in the list if all promises it depends upon appear later in the list.

The two assertions ownPicks and ownPromises are the assertions that tie the chosen picks and promises to separation logic resources. And since the picks and promises end up as restrictions on the transmap used to construct the transformation, it is also tied to resources.

The full definition of the model of the nextgen modality is slightly more complicated, but the above illustrate the key components.

6.7.3 Soundness of the Rules

Having defined the model for the nextgen modality, the next step is to prove soundness of its rules. Here we note a few things about a few of the rules where the soundness proof is particularly challenging or otherwise interesting.

The Rule for Separating Conjunction

One such rule is the rule for the nextgen modality and separating conjunction:

$$\langle \mathsf{NG}\rangle P * \langle \mathsf{NG}\rangle Q \vdash \langle \mathsf{NG}\rangle (P * Q)$$

This rules looks a lot like the BNG-SEP for the basic nextgen modality. But, a significant difference is that the nextgen modality has to parameters that determine the transformation function. For the basic nextgen modality the rule holds for two basic nextgen modalities where the exact same transformation function is used. This is

quite natural, but it does limit modularity in the sense that two different assertions proven in the next generation can only be combined if it has been coordinated that they apply the exact same transformation. For the nextgen modality here such coordination is not needed as the modality has no parameters at all. Proving the separation rule in this setting is significantly more interesting as the soundness of the rules ultimately depends on the construction being sufficiently modular in how the resources that determine the transformation is used.

When proving the rule with respect to the model, the two nextgen modalities on the left-hand side results in two pairs of picks and promises, picks_1 and $\mathit{promises}_1$ for the first nextgen and picks_2 and $\mathit{promises}_2$ for the second nextgen. The proof of P holds for a transformation function that satisfies the first picks and promises and the proof of Q requires the second pair. Hence, to get both P and Q we must build a *combined* map of picks and a combined map of promises that satisfies both. Here it is crucial that the model does not make exact requirements on what the transmap should be, but only "lower bounds" of requirements.

Building a combined transmap of picks is not too difficult. We have the two assertions $\mathsf{ownPicks}(\mathit{picks}_1)$ and $\mathsf{ownPicks}(\mathit{picks}_2)$, and since these contain an agreement resource (like pickedOut) the two transmaps are guaranteed to be equal in their overlap. Hence a "union" of the two can easily be constructed.

Building a combined list of promises, on the other hand, is very involved. As mentioned the lists need to be well-formed and to ensure this the two lists of promises needs to be merged with this property in mind. A given ghost location at i and γ might have a promise for it in *both* lists and in this case the strongest of the two promises must be chosen. To know that one promise is necessarily stronger than the other the resources $\mathsf{ownPromises}(\mathit{promises}_1)$ and $\mathsf{ownPromises}(\mathit{promises}_2)$ must be used. Intuitively, the promises in the lists and the dependencies between them can be though of as a tree-like graph, and merging the two lists amounts to merging the graphs while keeping the strongest promise when their is an overlap. Proving this formally is quite challenging, among other things, because setting up the induction is not straightforward.

The Rule for the Plainly Modality

Another interesting rule is the one for the plainly modality:

$$\langle \mathsf{NG} \rangle \blacksquare P \dashv\vdash \blacksquare P$$

Since this rule eliminates the nextgen modality, in the soundness proof one must provide a transmap for the universally quantified *tm*. This map must respect some map of picks and some list of promises. The challenging thing is to build a transamp that satisfies all the promises since they have dependencies between each other. To find transformations for ghost locations where nothing has been picked, but something has been promised we use the second condition from the definition of a promise Definition 6.5.1. This condition gives us a way to get a suitable transformation provided that we already have suitable transformations for all the dependencies

of the promise. Hence, to use the condition we must traverse the promises "from the bottom up" if one considers the dependencies as a tree. Fortunately the list of promises is stored exactly in an order that corresponds to such a bottom up traversal. With this order already in place the hardest part of the work has already been done and building a full map that satisfies all the promises is fairly straightforward.

6.7.4 Coq Mechanization

The nextgen modality is fully mechanized in Coq. One noteworthy aspect of the mechanization is that it makes heavy used of dependent types. There are various reasons why we need dependent types. For instance, the arity of the relation in a promise depends on the number $n \in \mathbb{N}$ of dependencies it has. AS another example, the transformations that satisfy a promise all have different types, and hence to store them we use a heterogeneous list. Additionally, all the structures related to the global RA, such as transmaps *TM* and resources for these such as ownPicks, also uses dependent types. Working with dependent types in Coq can be very difficult, and therefore the mechanization is overall quite advanced.

6.8 Conclusion and Future Work

In this chapter we have introduced a nextgen modality with promises and picks. We claim that this modality can replace the post-crash modality in both BaseSpirea and Spirea, to address the limitations around user-defined ghost state and crashes that we identified. We have shown how the resources used in BaseSpirea can be adapted to use the features of the nextgen modality such that they support the required rules in combination with the nextgen modality. Additionally, we have shown how the state interpretation and the model of the assertions in BaseSpirea can be defined with these adapted resource.

While the nextgen modality with promises and picks along with the resources for BaseSpirea are important steps towards an improved version of Spirea that address the issues we identified in the beginning of the chapter, more work is still needed to arrive at a complete program logic.

To do this a variant of Perennial's recovery weakest precondition that uses the nextgen modality from this chapter would have to be defined. The most challenging piece to such an endeavor is to prove an adequacy statement for this recovery weakest precondition. With this in place one would have an improved variant of BaseSpirea that uses the nextgen modality. This would address the issues for BaseSpirea. The higher-level Spirea uses additional resources used to model the features it contains. It extends the state interpretation from the base logic with ghost state for these additional resources. In order to adapt the Spirea to use the nextgen modality one would have to adapt these resources, similarly to what we have done in this section for BaseSpirea.

Bibliography

[BBT07] Bodil Biering, Lars Birkedal, and Noah Torp-Smith. "BI-hyperdoctrines, higher-order separation logic, and abstraction." In: *ACM Trans. Program. Lang. Syst.* 29.5 (2007), p. 24. DOI: 10.1145/1275497.1275499 (cit. on p. 58).

[Bil+22] Eleni Vafeiadi Bila, Brijesh Dongol, Ori Lahav, Azalea Raad, and John Wickerson. "View-Based Owicki-Gries Reasoning for Persistent x86-TSO." In: *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*. Ed. by Ilya Sergey. Vol. 13240. Lecture Notes in Computer Science. Springer, 2022, pp. 234–261. DOI: 10.1007/978-3-030-99336-8_9 (cit. on pp. 13, 76, 108, 112, 120).

[BB20] Lars Birkedal and Aleš Bizjak. *Lecture Notes on Iris: Higher-Order Concurrent Separation Logic*. 2020. URL: https://iris-project.org/tutorial-material.html (cit. on p. 120).

[BB21] Lars Birkedal and Aleš Bizjak. *Lecture Notes on Iris: Higher-Order Concurrent Separation Logic*. 2021. URL: https://iris-project.org/tutorial-material.html (cit. on p. 57).

[Bir+21] Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. "Theorems for free from separation logic specifications." In: *Proc. ACM Program. Lang.* 5.ICFP (2021), pp. 1–29. DOI: 10.1145/3473586 (cit. on pp. 12, 74).

[BB18] Ales Bizjak and Lars Birkedal. "On Models of Higher-Order Separation Logic." In: *Proceedings of the Thirty-Fourth Conference on the Mathematical Foundations of Programming Semantics, MFPS 2018, Dalhousie University, Halifax, Canada, June 6-9, 2018*. Ed. by Sam Staton. Vol. 341. Electronic Notes in Theoretical Computer Science. Elsevier, 2018, pp. 57–78. DOI: 10.1016/j.entcs.2018.03.016 (cit. on p. 21).

[Bor+05] Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. "Permission accounting in separation logic." In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14,*

2005. Ed. by Jens Palsberg and Martín Abadi. ACM, 2005, pp. 259–270. ISBN: 1-58113-830-X. DOI: 10.1145/1040305.1040327 (cit. on p. 21).

[Boy03] John Boyland. "Checking Interference with Fractional Permissions." In: *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*. 2003, pp. 55–72. DOI: 10.1007/3-540-44898-5_4 (cit. on p. 21).

[Bro20] Nathan Bronson. *On the origins of the MPMC Queue*. Personal Communication. Nov. 2020 (cit. on p. 49).

[Bro+13] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. "TAO: Facebook's Distributed Data Store for the Social Graph." In: *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX Association, June 2013, pp. 49–60. ISBN: 978-1-931971-01-0 (cit. on p. 49).

[Cai+20] Wentao Cai, Haosen Wen, H. Alan Beadle, Chris Kjellqvist, Mohammad Hedayati, and Michael L. Scott. "Understanding and optimizing persistent memory allocation." In: *ISMM '20: 2020 ACM SIGPLAN International Symposium on Memory Management, ISMM 2020, virtual [London, UK], June 16, 2020*. Ed. by Chen Ding and Martin Maas. ACM, 2020, pp. 60–73. ISBN: 978-1-4503-7566-5. DOI: 10.1145/3381898.3397212 (cit. on p. 76).

[Cai+21] Wentao Cai, Haosen Wen, Vladimir Maksimovski, Mingzhe Du, Rafaello Sanna, Shreif Abdallah, and Michael L. Scott. "Fast Nonblocking Persistence for Concurrent Data Structures." In: *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference)*. Ed. by Seth Gilbert. Vol. 209. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 14:1–14:20. ISBN: 978-3-95977-210-5. DOI: 10.4230/LIPIcs.DISC.2021.14 (cit. on p. 76).

[Cha22] Tej Chajed. "Verifying a concurrent, crash-safe file system with sequential reasoning." PhD thesis. Machetutes Institute of Technology, May 2022 (cit. on pp. 77, 114, 122, 129).

[Cha+17] Tej Chajed, Haogang Chen, Adam Chlipala, M. Frans Kaashoek, Nickolai Zeldovich, and Daniel Ziegler. "Certifying a file system using crash hoare logic: correctness in the presence of crashes." In: *Commun. ACM* 60.4 (2017), pp. 75–84. DOI: 10.1145/3051092 (cit. on p. 124).

[CTc22] Tej Chajed, Joseph Tassarotti, and contributors. *Post-crash modality in Perennial's Coq Mechanization*. Permanent link for the version available at the time of writing: https://github.com/mit-pdos/perennial/blob/ec3f44007d88b6ba28d1392807b444751588ed39/src/goose_lang/crash_modality.v. 2022. URL: https://github.com/

mit-pdos/perennial/blob/master/src/goose_lang/crash_modality.v (cit. on p. 129).

[Cha+19] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. "Verifying concurrent, crash-safe systems with Perennial." In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. Ed. by Tim Brecht and Carey Williamson. ACM, 2019, pp. 243–258. ISBN: 978-1-4503-6873-5. DOI: 10.1145/3341301.3359632 (cit. on pp. 77, 122, 124).

[Cha+21] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. "GoJournal: a verified, concurrent, crash-safe journaling system." In: *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. Ed. by Angela Demke Brown and Jay R. Lorch. USENIX Association, 2021, pp. 423–439 (cit. on pp. 77, 122, 124, 129).

[Cha+15] Soham Chakraborty, Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. "Aspect-oriented linearizability proofs." In: *Log. Methods Comput. Sci.* 11.1 (2015). DOI: 10.2168/LMCS-11(1:20)2015 (cit. on p. 74).

[CP17] Arthur Charguéraud and François Pottier. "Temporary Read-Only Permissions for Separation Logic." In: *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. Ed. by Hongseok Yang. Vol. 10201. Lecture Notes in Computer Science. Springer, 2017, pp. 260–286. ISBN: 978-3-662-54433-4. DOI: 10.1007/978-3-662-54434-1_10 (cit. on pp. 46, 125, 126, 151).

[Che+16] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. "Using Crash Hoare Logic for Certifying the FSCQ File System." In: *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*. Ed. by Ajay Gulati and Hakim Weatherspoon. USENIX Association, 2016 (cit. on pp. 77, 122).

[Che+20] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. "FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory." In: *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*. Ed. by James R. Larus, Luis Ceze, and Karin Strauss. ASPLOS 2020 was canceled because of COVID-19. ACM, 2020, pp. 1077–1091. ISBN: 978-1-4503-7102-5. DOI: 10.1145/3373376.3378515 (cit. on p. 76).

[Cho+21] Kyeongmin Cho, Sung Hwan Lee, Azalea Raad, and Jeehoon Kang. "Revamping hardware persistency models: view-based and axiomatic persistency models for Intel-x86 and Armv8." In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*. Ed. by Stephen N. Freund and Eran Yahav. ACM, 2021, pp. 16–31. ISBN: 978-1-4503-8391-2. DOI: 10.1145/3453483.3454027 (cit. on p. 122).

[Dan+20] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. "RustBelt meets relaxed memory." In: *Proc. ACM Program. Lang.* 4.POPL (2020), 34:1–34:29. DOI: 10.1145/3371102 (cit. on pp. 12, 81).

[Dan+22] Hoang-Hai Dang, Jaehwang Jung, Jaemin Choi, Duc-Than Nguyen, William Mansky, Jeehoon Kang, and Derek Dreyer. "Compass: strong and compositional library specifications in relaxed memory separation logic." In: *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*. Ed. by Ranjit Jhala and Isil Dillig. ACM, 2022, pp. 792–808. ISBN: 978-1-4503-9265-5. DOI: 10.1145/3519939.3523451 (cit. on pp. 12, 122).

[Din+13] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. "Views: Compositional Reasoning for Concurrent Programs." In: *POPL*. 2013, pp. 287–300. DOI: 10.1145/2429069.2429104 (cit. on p. 58).

[Din+10] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew Parkinson, and Viktor Vafeiadis. "Concurrent Abstract Predicates." In: *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*. Ed. by Theo D'Hondt. Vol. 6183. Lecture Notes in Computer Science. Springer, 2010, pp. 504–528. DOI: 10.1007/978-3-642-14107-2_24 (cit. on p. 48).

[Doh+04] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. "Formal Verification of a Practical Lock-Free Queue Algorithm." In: *Formal Techniques for Networked and Distributed Systems - FORTE 2004, 24th IFIP WG 6.1 International Conference, Madrid Spain, September 27-30, 2004, Proceedings*. Ed. by David de Frutos-Escrig and Manuel Núñez. Vol. 3235. Lecture Notes in Computer Science. Springer, 2004, pp. 97–114. ISBN: 3-540-23252-4. DOI: 10.1007/978-3-540-30232-2_7 (cit. on p. 46).

[DV16] Marko Doko and Viktor Vafeiadis. "A Program Logic for C11 Memory Fences." In: *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*. Ed. by Barbara Jobstmann and K. Rustan M. Leino. Vol. 9583. Lecture Notes in Computer Science. Springer,

2016, pp. 413–430. ISBN: 978-3-662-49121-8. DOI: 10.1007/978-3-662-49122-5_20 (cit. on pp. 77, 121).

[DD13] Brijesh Dongol and John Derrick. "Simplifying proofs of linearisability using layers of abstraction." In: *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.* 66 (2013). DOI: 10.14279/tuj.eceasst.66.889 (cit. on p. 73).

[DD14] Brijesh Dongol and John Derrick. "Verifying linearizability: A comparative survey." In: *CoRR* abs/1410.6268 (2014). arXiv: 1410.6268 (cit. on p. 46).

[DD15] Brijesh Dongol and John Derrick. "Verifying Linearisability: A Comparative Survey." In: *ACM Comput. Surv.* 48.2 (2015), 19:1–19:43. DOI: 10.1145/2796550 (cit. on pp. 10, 50, 74).

[DDH12] Brijesh Dongol, John Derrick, and Ian J. Hayes. "Fractional Permissions and Non-Deterministic Evaluators in Interval Temporal Logic." In: *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.* 53 (2012). DOI: 10.14279/tuj.eceasst.53.792 (cit. on p. 73).

[Fil+10a] Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzky, and Hongseok Yang. "Abstraction for concurrent objects." In: *Theor. Comput. Sci.* 411.51-52 (2010), pp. 4379–4398. DOI: 10.1016/j.tcs.2010.09.021 (cit. on p. 20).

[Fil+10b] Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. "Abstraction for concurrent objects." In: *Theoretical Computer Science* 411.51-52 (2010), pp. 4379–4398. DOI: 10.1016/j.tcs.2010.09.021 (cit. on p. 74).

[FRK02] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. "Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux." In: *Proceedings of Ottawa Linux Symposium*. Vol. 85. 2002, pp. 479–495 (cit. on p. 54).

[Fri+20] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. "NVTraverse: in NVRAM data structures, the destination is more important than the journey." In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. Ed. by Alastair F. Donaldson and Emina Torlak. ACM, 2020, pp. 377–392. ISBN: 978-1-4503-7613-6. DOI: 10.1145/3385412.3386031 (cit. on p. 116).

[Fri+18] Michal Friedman, Maurice Herlihy, Virendra J. Marathe, and Erez Petrank. "A persistent lock-free queue for non-volatile memory." In: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*. Ed. by Andreas Krall and Thomas R. Gross. ACM, 2018, pp. 28–40. ISBN: 978-1-4503-4982-6. DOI: 10.1145/3178487.3178490 (cit. on p. 76).

[FKB18] Dan Frumin, Robbert Krebbers, and Lars Birkedal. "ReLoC: A Mechanised Relational Logic for Fine-Grained Concurrency." In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*. Ed. by Anuj Dawar and Erich Grädel. ACM, 2018, pp. 442–451. DOI: 10.1145/3209108.3209174 (cit. on pp. 13, 48–50).

[FKB20a] Dan Frumin, Robbert Krebbers, and Lars Birkedal. "ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity." In: *CoRR* abs/2006.13635 (2020). arXiv: 2006.13635 (cit. on pp. 13, 49, 50, 72).

[FKB20b] Dan Frumin, Robbert Krebbers, and Lars Birkedal. "ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity." In: *CoRR* abs/2006.13635 (2020). arXiv: 2006.13635 (cit. on pp. 20, 23, 30, 33).

[Geo+20] Jerrin Shaji George, Mohit Verma, Rajesh Venkatasubramanian, and Pratap Subrahmanyam. "go-pmem: Native Support for Programming Persistent Memory in Go." In: *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*. Ed. by Ada Gavrilovska and Erez Zadok. USENIX Association, 2020, pp. 859–872. ISBN: 978-1-939133-14-4 (cit. on p. 76).

[Geo+23] Aına Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. "Cerise: Program Verification on a Capability Machine in the Presence of Untrusted Code." In: *J. ACM* (Sept. 2023). ISSN: 0004-5411. DOI: 10.1145/3623510 (cit. on p. 14).

[Got+07] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. "Local Reasoning for Storable Locks and Threads." In: *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007, Singapore, November 29-December 1, 2007, Proceedings*. 2007, pp. 19–37. DOI: 10.1007/978-3-540-76637-7_3 (cit. on p. 57).

[Gué+23] Armaël Guéneau, Johannes Hostert, Simon Spies, Michael Sammler, Lars Birkedal, and Derek Dreyer. "Melocoton: A Program Logic for Verified Interoperability Between OCaml and C." In: *Proc. ACM Program. Lang.* OOPSLA (2023) (cit. on p. 125).

[HSY04] Danny Hendler, Nir Shavit, and Lena Yerushalmi. "A Scalable Lock-Free Stack Algorithm." In: *SPAA*. 2004, pp. 206–215. DOI: 10.1145/1007912.1007944 (cit. on pp. 71, 73).

[HW90] Maurice Herlihy and Jeannette Wing. "Linearizability: A Correctness Condition for Concurrent Objects." In: *TOPLAS* 12.3 (1990), pp. 463–492. DOI: 10.1145/78969.78972 (cit. on pp. 10, 51, 74).

[HW87] Maurice Herlihy and Jeannette M. Wing. "Axioms for Concurrent Objects." In: *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*. ACM Press, 1987, pp. 13–26. DOI: 10.1145/41625.41627 (cit. on p. 10).

[Hug89] John Hughes. "Why Functional Programming Matters." In: *Comput. J.* 32.2 (1989), pp. 98–107. DOI: 10.1093/comjnl/32.2.98 (cit. on p. 50).

[IMS16] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. "Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model." In: *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*. Ed. by Cyril Gavoille and David Ilcinkas. Vol. 9888. Lecture Notes in Computer Science. Springer, 2016, pp. 313–327. DOI: 10.1007/978-3-662-53426-7_23 (cit. on pp. 11, 76, 78, 79, 82, 116).

[JP11] Bart Jacobs and Frank Piessens. "Expressive modular fine-grained concurrency specification." In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. Ed. by Thomas Ball and Mooly Sagiv. ACM, 2011, pp. 271–282. DOI: 10.1145/1926385.1926417 (cit. on pp. 48, 74).

[Jun+16] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. "Higher-order ghost state." In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. Ed. by Jacques Garrigue, Gabriele Keller, and Eijiro Sumii. ACM, 2016, pp. 256–269. DOI: 10.1145/2951913.2951943 (cit. on p. 49).

[Jun+18a] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic." In: *J. Funct. Program.* 28 (2018), e20. DOI: 10.1017/S0956796818000151 (cit. on pp. 21, 43, 76, 94, 124, 133).

[Jun+18b] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic." In: *J. Funct. Program.* 28 (2018), e20. DOI: 10.1017/S0956796818000151 (cit. on p. 49).

[Jun+20a] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. "The future is ours: prophecy variables in separation logic." In: *Proc. ACM Program. Lang.* 4.POPL (2020), 45:1–45:32. DOI: 10.1145/3371113 (cit. on pp. 12, 50).

[Jun+20b] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. "The future is ours: prophecy variables in separation logic." In: *Proc. ACM Program. Lang.* 4.POPL (2020), 45:1–45:32. DOI: 10.1145/3371113 (cit. on p. 30).

[Jun+15a] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek 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 2015, Mumbai, India, January 15-17, 2015*. Ed. by Sriram K. Rajamani and David Walker. ACM, 2015, pp. 637–650. DOI: 10.1145/2676726.2676980 (cit. on pp. 48, 49, 58, 74).

[Jun+15b] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek 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 2015, Mumbai, India, January 15-17, 2015*. Ed. by Sriram K. Rajamani and David Walker. ACM, 2015, pp. 637–650. DOI: 10.1145/2676726.2676980 (cit. on p. 124).

[Kai+17] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. "Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris." In: *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*. Ed. by Peter Müller. Vol. 74. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 17:1–17:29. ISBN: 978-3-95977-035-4. DOI: 10.4230/LIPIcs.ECOOP.2017.17 (cit. on pp. 12, 62, 77, 81).

[Kai+19] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young-ri Choi. "SLM-DB: Single-Level Key-Value Store with Persistent Memory." In: *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*. Ed. by Arif Merchant and Hakim Weatherspoon. USENIX Association, 2019, pp. 191–205 (cit. on p. 76).

[KL21] Artem Khyzha and Ori Lahav. "Taming x86-TSO persistency." In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–29. DOI: 10.1145/3434328 (cit. on p. 122).

[Kre+18] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. "MoSeL: a general, extensible modal framework for interactive proofs in separation logic." In: *Proc. ACM Program. Lang.* 2.ICFP (2018), 77:1–77:30. DOI: 10.1145/3236772 (cit. on pp. 21, 50).

[Kre+17a] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. "The Essence of Higher-Order Concurrent Separation Logic." In: *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. Ed. by Hongseok Yang. Vol. 10201. Lecture Notes in Computer Science. Springer, 2017, pp. 696–723. DOI: 10.1007/978-3-662-54434-1_26 (cit. on p. 49).

[Kre+17b] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. "The Essence of Higher-Order Concurrent Separation Logic." In: *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. Ed. by Hongseok Yang. Vol. 10201. Lecture Notes in Computer Science. Springer, 2017, pp. 696–723. DOI: 10.1007/978-3-662-54434-1_26 (cit. on p. 126).

[KTB17a] Robbert Krebbers, Amin Timany, and Lars Birkedal. "Interactive proofs in higher-order concurrent separation logic." In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by Giuseppe Castagna and Andrew D. Gordon. ACM, 2017, pp. 205–217. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837 (cit. on pp. 21, 32, 34).

[KTB17b] Robbert Krebbers, Amin Timany, and Lars Birkedal. "Interactive proofs in higher-order concurrent separation logic." In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by Giuseppe Castagna and Andrew D. Gordon. ACM, 2017, pp. 205–217 (cit. on p. 50).

[KTB17c] Robbert Krebbers, Amin Timany, and Lars Birkedal. "Interactive proofs in higher-order concurrent separation logic." In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by Giuseppe Castagna and Andrew D. Gordon. ACM, 2017, pp. 205–217. DOI: 10.1145/3009837.3009855 (cit. on pp. 120, 127, 131).

[LF13a] Hongjin Liang and Xinyu Feng. "Modular verification of linearizability with non-fixed linearization points." In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, 2013, pp. 459–470. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2491956.2462189 (cit. on p. 46).

[LF13b] Hongjin Liang and Xinyu Feng. "Modular verification of linearizability with non-fixed linearization points." In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle,*

WA, USA, June 16-19, 2013. Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, 2013, pp. 459–470. DOI: 10.1145/2491956.2462189 (cit. on pp. 48, 73).

[MP22] Jean-Marie Madiot and François Pottier. "A separation logic for heap space under garbage collection." In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–28. DOI: 10.1145/3498672 (cit. on pp. 14, 125).

[MS91] John M. Mellor-Crummey and Michael L. Scott. "Algorithms for scalable synchronization on shared-memory multiprocessors." In: *TOCS* 9.1 (1991), pp. 21–65. DOI: 10.1145/103727.103729 (cit. on p. 57).

[Met21] Meta. *Folly: Facebook Open-source Library*. https://github.com/facebook/folly, last accessed: 2021-02-25. 2021 (cit. on p. 49).

[MJ21a] Glen Mével and Jacques-Henri Jourdan. "Formal verification of a concurrent bounded queue in a weak memory model." In: *Proc. ACM Program. Lang.* 5.ICFP (2021), pp. 1–29. DOI: 10.1145/3473571 (cit. on pp. 73, 74).

[MJ21b] Glen Mével and Jacques-Henri Jourdan. "Formal verification of a concurrent bounded queue in a weak memory model." In: *Proc. ACM Program. Lang.* 5.ICFP (2021), pp. 1–29. DOI: 10.1145/3473571 (cit. on p. 122).

[MJP20] Glen Mével, Jacques-Henri Jourdan, and François Pottier. "Cosmo: a concurrent separation logic for multicore OCaml." In: *Proc. ACM Program. Lang.* 4.ICFP (2020), 96:1–96:29. DOI: 10.1145/3408978 (cit. on pp. 12, 73).

[MS96a] Maged Michael and Michael Scott. "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms." In: *PODC*. 1996, pp. 267–275. DOI: 10.1145/248052.248106 (cit. on pp. 49, 51).

[MS96b] Maged M. Michael and Michael L. Scott. "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms." In: *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996*. Ed. by James E. Burns and Yoram Moses. ACM, 1996, pp. 267–275. DOI: 10.1145/248052.248106 (cit. on pp. 19, 24).

[MCP23] Alexandre Moine, Arthur Charguéraud, and François Pottier. "A High-Level Separation Logic for Heap Space under Garbage Collection." In: *Proc. ACM Program. Lang.* 7.POPL (2023), pp. 718–747. DOI: 10.1145/3571218 (cit. on p. 125).

[MWB24] Alexandre Moine, Sam Westrick, and Stephanie Balzer. "DisLog: A Separation Logic for Disentanglement." In: *Proceedings of the ACM on Programming Languages* 8.POPL (Jan. 2024). DOI: 10.1145/3632853 (cit. on p. 14).

[MT19] Andrzej S. Murawski and Nikos Tzevelekos. "Higher-order linearisability." In: *J. Log. Algebraic Methods Program.* 104 (2019), pp. 86–116. DOI: 10.1016/j.jlamp.2019.01.002 (cit. on p. 20).

[Nan+14] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. "Communicating State Transition Systems for Fine-Grained Concurrent Resources." In: *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. Ed. by Zhong Shao. Vol. 8410. Lecture Notes in Computer Science. Springer, 2014, pp. 290–310. DOI: 10.1007/978-3-642-54833-8_16 (cit. on p. 58).

[PB05] Matthew Parkinson and Gavin Bierman. "Separation logic and abstraction." In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*. Ed. by Jens Palsberg and Martín Abadi. ACM, 2005, pp. 247–258. DOI: 10.1145/1040305.1040326 (cit. on p. 58).

[PCW14] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. "Memory persistency." In: *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*. IEEE Computer Society, 2014, pp. 265–276. ISBN: 978-1-4799-4396-8. DOI: 10.1109/ISCA.2014.6853222 (cit. on p. 81).

[RLV20] Azalea Raad, Ori Lahav, and Viktor Vafeiadis. "Persistent Owicki-Gries reasoning: a program logic for reasoning about persistent programs on Intel-x86." In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 151:1–151:28. DOI: 10.1145/3428219 (cit. on pp. 13, 76, 78, 113, 114, 120).

[Raa+20] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. "Persistency semantics of the Intel-x86 architecture." In: *Proc. ACM Program. Lang.* 4.POPL (2020), 11:1–11:31. DOI: 10.1145/3371079 (cit. on p. 122).

[RWV19] Azalea Raad, John Wickerson, and Viktor Vafeiadis. "Weak persistency semantics from the ground up: formalising the persistency semantics of ARMv8 and transactional models." In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019), 135:1–135:27. DOI: 10.1145/3360561 (cit. on p. 122).

[RCF21] Pedro Ramalhete, Andreia Correia, and Pascal Felber. "Efficient algorithms for persistent transactional memory." In: *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*. Ed. by Jaejin Lee and Erez Petrank. ACM, 2021, pp. 1–15. ISBN: 978-1-4503-8294-6. DOI: 10.1145/3437801.3441586 (cit. on p. 76).

[Rey02] John C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures." In: *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817 (cit. on p. 21).

[RDG14] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. "TaDA: A logic for time and data abstraction." In: *ECOOP*. Vol. 8586. LNCS. 2014, pp. 207–231. DOI: 10.1007/978-3-662-44202-9_9 (cit. on pp. 12, 74).

[SDW14] Gerhard Schellhorn, John Derrick, and Heike Wehrheim. "A Sound and Complete Proof Technique for Linearizability of Concurrent Data Structures." In: *ACM Trans. Comput. Log.* 15.4 (2014), 31:1–31:37. DOI: 10.1145/2629496 (cit. on p. 46).

[Sch+15] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. "nvm malloc: Memory Allocation for NVRAM." In: *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2015, Kohala Coast, Hawaii, USA, August 31, 2015*. Ed. by Rajesh Bordawekar, Tirthankar Lahiri, Bugra Gedik, and Christian A. Lang. 2015, pp. 61–72 (cit. on p. 76).

[SNB15] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. "Mechanized verification of fine-grained concurrent programs." In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. Ed. by David Grove and Steve Blackburn. ACM, 2015, pp. 77–87. DOI: 10.1145/2737924.2737964 (cit. on p. 48).

[Sha+23] Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. "Grove: a Separation-Logic Library for Verifying Distributed Systems (Extended Version)." In: *CoRR* abs/2309.03046 (2023). DOI: 10.48550/arXiv.2309.03046. arXiv: 2309.03046 (cit. on p. 125).

[Spi+21] Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. "Transfinite Iris: resolving an existential dilemma of step-indexed separation logic." In: *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*. Ed. by Stephen N. Freund and Eran Yahav. ACM, 2021, pp. 80–95. DOI: 10.1145/3453483.3454031 (cit. on p. 126).

[SB14] Kasper Svendsen and Lars Birkedal. "Impredicative Concurrent Abstract Predicates." In: *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. Ed. by Zhong Shao. Vol. 8410.

Lecture Notes in Computer Science. Springer, 2014, pp. 149–168. DOI: 10.1007/978-3-642-54833-8_9 (cit. on p. 48).

[SBP13] Kasper Svendsen, Lars Birkedal, and Matthew Parkinson. "Modular Reasoning about Separation of Concurrent Data Structures." In: *ESOP*. Vol. 7792. LNCS. 2013, pp. 169–188. ISBN: 978-3-642-37036-6. DOI: 10.1007/978-3-642-37036-6_11 (cit. on pp. 12, 74).

[TB19] Amin Timany and Lars Birkedal. "Mechanized relational verification of concurrent programs with continuations." In: *Proc. ACM Program. Lang.* 3.ICFP (2019), 105:1–105:28. DOI: 10.1145/3341709 (cit. on pp. 139, 142, 151).

[Tim+21] Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Léon Gondelman, Abel Nieto, and Lars Birkedal. "Trillium: Unifying Refinement and Higher-Order Distributed Separation Logic." In: *CoRR* abs/2109.07863 (2021). arXiv: 2109.07863 (cit. on p. 14).

[Tim+18] Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. "A logical relation for monadic encapsulation of state: proving contextual equivalences in the presence of runST." In: *Proc. ACM Program. Lang.* 2.POPL (2018), 64:1–64:28. DOI: 10.1145/3158152 (cit. on pp. 126, 152).

[TDB13] Aaron Turon, Derek Dreyer, and Lars Birkedal. "Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency." In: *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. Ed. by Greg Morrisett and Tarmo Uustalu. ACM, 2013, pp. 377–390. DOI: 10.1145/2500365.2500600 (cit. on pp. 48, 50, 70, 73).

[Tur+13a] Aaron Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. "Logical relations for fine-grained concurrency." In: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. Ed. by Roberto Giacobazzi and Radhia Cousot. ACM, 2013, pp. 343–356. DOI: 10.1145/2429069.2429111 (cit. on pp. 48, 71, 73).

[TVD14] Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. "GPS: navigating weak memory with ghosts, protocols, and separation." In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. 2014, pp. 691–707. DOI: 10.1145/2660193.2660243 (cit. on pp. 62, 77, 121).

[TW11] Aaron Turon and Mitchell Wand. "A separation logic for refining concurrent objects." In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. Ed. by Thomas Ball and Mooly Sagiv. ACM, 2011, pp. 247–258. DOI: 10.1145/1926385.1926415 (cit. on p. 48).

[Tur+13b] Aaron Joseph Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. "Logical relations for fine-grained concurrency." In: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. Ed. by Roberto Giacobazzi and Radhia Cousot. ACM, 2013, pp. 343–356. DOI: 10.1145/2429069.2429111 (cit. on pp. 20, 21, 34, 42).

[Vaf08] Viktor Vafeiadis. "Modular fine-grained concurrency verification." PhD thesis. University of Cambridge, 2008 (cit. on p. 48).

[Vaf10] Viktor Vafeiadis. "Automatically Proving Linearizability." In: *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*. Ed. by Tayssir Touili, Byron Cook, and Paul B. Jackson. Vol. 6174. Lecture Notes in Computer Science. Springer, 2010, pp. 450–464. ISBN: 978-3-642-14294-9. DOI: 10.1007/978-3-642-14295-6_40 (cit. on p. 46).

[VN13] Viktor Vafeiadis and Chinmay Narayan. "Relaxed separation logic: a program logic for C11 concurrency." In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. Ed. by Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes. ACM, 2013, pp. 867–884. ISBN: 978-1-4503-2374-1. DOI: 10.1145/2509136.2509532 (cit. on p. 77).

[VP07] Viktor Vafeiadis and Matthew Parkinson. "A Marriage of Rely/Guarantee and Separation Logic." In: *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings*. Ed. by Luís Caires and Vasco Thudichum Vasconcelos. Vol. 4703. Lecture Notes in Computer Science. Springer, 2007, pp. 256–271. DOI: 10.1007/978-3-540-74407-8_18 (cit. on p. 48).

[VP23] Paulo Emílio de Vilhena and François Pottier. "A Type System for Effect Handlers and Dynamic Labels." In: *Programming Languages and Systems - 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings*. Ed. by Thomas Wies. Vol. 13990. Lecture Notes in Computer Science. Springer, 2023, pp. 225–252. DOI: 10.1007/978-3-031-30044-8_9 (cit. on p. 14).

[VB23a] Simon Vindum and Lars Birkedal. "Spirea: A Mechanized Concurrent Separation Logic for Weak Persistent Memory." In: *Proc. ACM Program. Lang.* OOPSLA (2023) (cit. on pp. 124, 132).

[VB20] Simon Friis Vindum and Lars Birkedal. *Contextual Refinement of the Michael-Scott Queue - Coq Artifact*. Version 1.0.0. Dec. 2020. DOI: 10.5281/zenodo.4317021 (cit. on p. 22).

[VB21] Simon Friis Vindum and Lars Birkedal. "Contextual refinement of the Michael-Scott queue (proof pearl)." In: *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*. ACM, 2021, pp. 76–90. DOI: 10.1145/3437992.3439930 (cit. on pp. 13, 16, 66).

[VB23b] Simon Friis Vindum and Lars Birkedal. *Artifact for the paper "Spirea: A Mechanized Concurrent Separation Logic for Weak Persistent Memory" in OOPSLA23*. Sept. 2023. DOI: 10.5281/zenodo.8314888 (cit. on p. 80).

[VB23c] Simon Friis Vindum and Lars Birkedal. "Spirea: A Mechanized Concurrent Separation Logic for Weak Persistent Memory." In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (Oct. 2023). DOI: 10.1145/3622820 (cit. on pp. 14, 17, 84).

[VFB21] Simon Friis Vindum, Dan Frumin, and Lars Birkedal. *Coq development for "Mechanized Verification of a Fine-Grained Concurrent Queue from Meta's Folly Library"*. Version 1.0.0. Dec. 2021. DOI: 10.5281/zenodo.5770802 (cit. on p. 51).

[VFB22] Simon Friis Vindum, Dan Frumin, and Lars Birkedal. "Mechanized verification of a fine-grained concurrent queue from meta's folly library." In: *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*. Ed. by Andrei Popescu and Steve Zdancewic. ACM, 2022, pp. 100–115. DOI: 10.1145/3497775.3503689 (cit. on pp. 13, 16).

[VTS11] Haris Volos, Andres Jaan Tack, and Michael M. Swift. "Mnemosyne: lightweight persistent memory." In: *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*. Ed. by Rajiv Gupta and Todd C. Mowry. ACM, 2011, pp. 91–104. ISBN: 978-1-4503-0266-1. DOI: 10.1145/1950365.1950379 (cit. on p. 76).

[Woo+14] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. "The CHERI capability model: Revisiting RISC in an age of risk." In: *ACM SIGARCH Computer Architecture News* 42.3 (2014), pp. 457–468 (cit. on p. 142).

[Zou+19] Mo Zou, Haoran Ding, Dong Du, Ming Fu, Ronghui Gu, and Haibo Chen. "Using concurrent relational logic with helpers for verifying the AtomFS file system." In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. Ed. by Tim Brecht and Carey Williamson. ACM, 2019, pp. 259–274. DOI: 10.1145/3341301.3359644 (cit. on p. 73).