Scaling Up Relaxed Memory Verification with Separation Logics

Dissertation zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften

der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

vorgelegt von Hoang-Hai Dang

Saarbrücken, 2024

Tag des Kolloquiums
10. September, 2024

Dekan der Fakultät für Mathematik und Informatik
Prof. Dr. Roland Speicher

Prüfungsausschuss

Vorsitzender:	Prof. Dr. Sven Apel
Gutachter:	Prof. Dr. Derek Dreyer
	Prof. Dr. Mark Batty
	Dr. Viktor Vafeiadis
Akademischer Mitarbeiter:	Dr. Léo Stefanesco

Abstract

Reasoning about concurrency in a realistic, non-toy language like C/C++ or Rust, which encompasses many interweaving complex features, is very hard. Yet, *realistic* concurrency involves relaxed memory models, which are significantly harder to reason about than the simple, traditional concurrency model that is sequential consistency. To *scale up* verifications to realistic concurrency, we need a few ingredients: (1) strong but abstract reasoning principles so that we can avoid the too tedious details of the underlying concurrency model; (2) modular reasoning so that we can compose smaller verification results into larger ones; (3) reasoning extensibility so that we can derive new reasoning principles for both complex language features and algorithms without rebuilding our logic from scratch; and (4) machine-checked proofs so that we do not miss potential unsoundness in our verifications. With these ingredients in hand, a logic designer can flexibly accommodate the intricacy of relaxed memory features and the ingenuity of programmers who exploit those features.

In this dissertation, I present how to develop strong, abstract, modular, extensible, and machine-checked separation logics for realistic relaxed memory concurrency in the Iris framework, using multiple layers of abstractions. I report two main applications of such logics: (i) the verification of the Rust type system with a relaxed memory model, where relaxed memory effects are encapsulated behind the safe interface of libraries and thus are not visible to clients, and (ii) the compositional specification and verification of relaxed memory libraries, in which relaxed memory effects are exposed to clients.

Zusammenfassung

Programmverifikation von nebenläufige Programmen in einer realistischen Programmiersprache wie C/C++ oder Rust, die viele komplexe, miteinander verflochtene Sprachkonstrukte enthält, ist sehr schwierig. Realistische nebenläufige Programme basieren auf *schwachen Speicherkonsistenzmodellen*, in denen sich die Beweisführung im Vergleich zum traditionellen, sequentiellen Speicherkonsistenzmodell (SC) erheblich schwieriger gestaltet. Um die Verifikation solcher realistischen nebenläufigen Programme zu ermöglichen benötigen wir mehrere Voraussetzungen: (1) starke Beweisregeln die die mühsamen Details des zugrundeliegenden Speicherkonsistenzmodells abstrahieren, (2) modulare Beweistechniken die es erlauben, die Verifikation in kleinere, mundgerechte Beweise aufzuteilen, (3) eine erweiterbare Verifikationslogik, in der neue Beweistechniken hinzugefügt werden können, ohne die Korrektheit der gesamten Logik erneut beweisen zu müseen (4) maschinengeprüfte Beweise, die die Korrektheit der Logik und der durchgeführten Beweise garantiert. Mit diesen Voraussetzungen kann ein Logikdesigner die Komplexität des schwachen Speicherkonsistenzmodells und den Einfallsreichtum der Programmierer, die sich dessen Funktionenen zu Nutze machen, flexibel berücksichtigen.

In dieser Dissertation stelle ich vor, wie man starke, abstrakte, modulare, erweiterbare und maschinengeprüfte Separationslogiken für realistische schwache Speicherkonsistenz in dem Framework Iris mit Hilfe von mehreren Abstraktionsebenen erstellen kann. Ich berichte über zwei Hauptanwendungen dieser Logiken: (i) die Verifikation des Typsystems von Rust auf Basis eines schwachen Speicherkonsistenzmodells, bei dem die Auswirkungen schwacher Speicherkonsistenz hinter der sicheren Programmschnittstelle abstrahiert und somit für Clients unsichtbar sind, und (ii) die modulare Spezifikation und Verifikation von Programmbibliotheken mit schwacher Speicherkonsistenz, bei denen die Auswirkungen schwacher Speicherkonsistenz für Clients sichtbar sind.

Acknowledgments

My PhD work was long and difficult. Fortunately, in the end, it has been a fruitful journey—I would not have been able to image where I am today. And for that, I am grateful to be able to work with and learn from ingenious and amazing people during that time.

I would like to deeply thank my advisor, Derek Dreyer, for his giving me the opportunity and support to embark on this arduous journey. Derek has always managed to surround himself with the most interesting research problems, and, as a result, I as his student, had multiple chances to work with some of those. I remember my first project with him was about using the very first version of Iris to encode high-order reasoning principles for continuations. I started the project without knowledge on Iris and reasoning about continuations, but Derek had encouraged and helped me through it. For that first project, I would also additionally like to thank David Swasey for his various guiding discussions, even though most did not make sense to me at the time. After that, Derek put me on the true beginning of my PhD life: the iGPS project. I still do not understand how or if Derek had foreseen that I could survive that! With some luck, I got through it and learnt a lot, and for that, I also would like to thank Janno Kaiser, Ori Lahav, and Viktor Vafeiadis for their teaching me about relaxed memory and Coq. After the success of iGPS, I have had the chances to work on more difficult problems, and Derek had always been there for me. I still remember the night of 2018, when we were running towards the deadline of the PLDI submission, and Derek found the bug in the race detector of ORC11, and we had to decide to submit the RustBelt Relaxed paper later. Thank you Derek, for always being vigilant and on the point.

I thank Mark Batty and Viktor Vafeadis for serving as reviewers of my dissertation, and Sven Apel as chair and Léo Stefanesco as academic member of my committee. I would also like to thank all of the collaborators that I have worked with—without them, I would have achieved nothing. I would like to thank Jacques-Henri Jourdan, Ralf Jung, Jeehoon Kang, Azalea Raad, William Mansky, Robbert Krebbers, Jaehwang Jung, Jaemin Choi, Duc-Than Nguyen, Joshua Yanovski, Lennard Gäher, Michael Sammler, and Simon Spies. I am forever grateful to Rose Hoberman for her teaching of English, scientific writing, and communication skills.

Living in Germany would me impossible without my friends. I thank my dear Vietnamese friends in Saarbrücken, in Luxembourg, in Vienna, and in Antibes. I would also my friends that I got to know at MPI-SWS: Heiko Becker, Debasmita Lohar, Reinhard Munz, Ezgi Cicek, Rodolphe Lepigre, Gaurav Parthasarathy, and Raphaël Monat.

Last but not least, I thank my beloved family: my parents and my big brother, for having also been there for me, across the continents. Without them, I would not have even arrived in this continent and would not have met so many great people. I dedicate this dissertation to my mother and my father.

Saarbrücken, September 2024

Đặng Hoàng Hải

Contents

Abstract iii

Zusammenfassung iii

Acknowledgments v

Contents ix

List of Figures xiii

List of Tables xv

Glossary xvii

1 Introduction 1
- 1.1 Reasoning about Relaxed Memory Concurrency . . . 2
- 1.2 RustBelt Relaxed: Verifying the Soundness of Rust's Type System in RMC . . . 3
- 1.3 Compass: Strong and Compositional Specifications of Relaxed-Memory Libraries . . . 5
- 1.4 Structure . . . 8
- 1.5 Publications and Collaborations . . . 9

I OPERATIONAL SEMANTICS FOR RELAXED MEMORY 11

2 Background: Relaxed Memory Models 15
- 2.1 C11, Intuitively . . . 15
- 2.2 RC11, Formally . . . 17

3 ORC11: Operational Repaired C11 27
- 3.1 Understanding Relaxed Memory with Views . . . 27
- 3.2 Basic Machine State Definitions . . . 30
- 3.3 View-based RMC Semantics . . . 34
- 3.4 The Data-Race Detector . . . 39
- 3.5 Comparison with iGPS Race Detector . . . 42
- 3.6 The Correspondence between RC11 and ORC11 . . . 44

4 The Relaxed λ_{Rust} Language 47
- 4.1 Language Syntax . . . 47
- 4.2 Language Expression Reductions . . . 50
- 4.3 The Complete Operational Semantics of Relaxed λ_{Rust} . . . 54

5 Related Work 57

II SEPARATION LOGIC FOR RELAXED MEMORY 59

6 More Background: Iris, A Framework for Concurrent Separation Logics 63
- 6.1 Basic Rules . . . 64
- 6.2 Ghost State and Resource Algebras . . . 65
- 6.3 Invariants and Fancy Updates . . . 67

6.4 Hoare Triples . . . 69
6.5 Adequacy . . . 69
6.6 Some Common Rules for WPs and Hoare Triples . . . 70
6.7 Weakest Pre-conditions and Invariants . . . 71
6.8 Properties of Propositions . . . 72
6.9 The Method of Fictional Separation . . . 73
6.10 The Physical State Interpretation . . . 75
6.11 An Instantiation Example for Simple Heaps . . . 75

7 A Base Logic for RMC in Iris 79
7.1 Thread-local Configurations as Expressions . . . 79
7.2 Basic Local Assertions for View-based RMC . . . 81
7.3 Primitive Memory Rules . . . 84
7.4 Resource Algebras for Basic Local Assertions . . . 92
7.5 State Interpretation . . . 93
7.6 Proofs of Some Primitive Rules and Adequacy . . . 96

8 vProp: View-monotone Predicates 99
8.1 View-monotone Predicates . . . 99
8.2 Model of iRC11 Weakest Pre-conditions . . . 101
8.3 Fence Modalities . . . 102
8.4 Objective Propositions and The Objective Modality . . . 105
8.5 View-explicit Modalities . . . 106
8.6 The Subjective Modality . . . 109

9 Non-Atomic Points-To 111
9.1 The Interface of Non-Atomic Points-To . . . 111
9.2 The Model of Non-Atomic Points-To . . . 112

10 Atomic Points-To 115
10.1 The Interface of the Atomic Points-To Assertion . . . 116
10.2 The Model of the Atomic Points-To Assertion . . . 126

11 Invariants in Relaxed Memory 133
11.1 Objective Invariants . . . 134
11.2 Cancelable Invariants . . . 136
11.3 Non-Atomic Invariants . . . 143

12 Example Verifications with iRC11 145
12.1 Release-Acquire Message-Passing . . . 145
12.2 Release-Acquire Message-Passing with Reclamation . . . 149
12.3 Spawn and Join . . . 154
12.4 A Release-Acquire Treiber Stack . . . 156

13 Related Work 167

III RUSTBELT MEETS RELAXED MEMORY 171

14 Challenge: RustBelt and Relaxed Memory 173
14.1 Task 1: Re-prove the Safety of Rust Libraries under RMC . . . 174
14.2 Task 2: Re-prove the Safety of the λ_{Rust} Type System under RMC . . . 176
14.3 Contributions of RustBelt Relaxed . . . 176

15 The Lifetime Logic of SC RustBelt 179
15.1 Borrowing in Rust . . . 179
15.2 The Lifetime Logic Primer, in SC . . . 181

16 Lifetime Logic Meets Relaxed Memory 187
16.1 More Rules for the Lifetime Logic . . . 187
16.2 Other Forms of Borrows . . . 191
16.3 Adaption of the Lifetime Logic's Model in iRC11 . . . 195

17 GPS Single-Location Protocols 201
17.1 Surface-level GPS Protocols in iRC11 . . . 201
17.2 Middleware GPS Protocols in iRC11 . . . 218
17.3 The Model of GPS Protocols . . . 221

18 Verification of `RwLock` 225
18.1 RMC Implementation of a Reader-Writer Lock . . . 225
18.2 The Semantic Model of the Reader-Writer Lock Type . . . 229
18.3 Proof Sketches of the Library's Operations . . . 233

19 Verification of `Arc` 237
19.1 The Core `Arc` library . . . 237
19.2 Verification of Core `Arc` with Cancelable GPS Protocols . . . 239
19.3 Verification of `Arc`'s Full APIs . . . 245
19.4 Insufficient Synchronization in `get_mut` . . . 249

20 Related Work 251

IV COMPASS 253

21 Background: Strong Specifications with Logical Atomicity 257
21.1 Sequential Specifications for Queues . . . 257
21.2 SC Specifications with Logical Atomicity . . . 258
21.3 Logically Atomic Specifications in RMC with Views . . . 260

22 Strong Compass Specifications with Richer Partial Orders 263
22.1 Graph-Based Specs to Encode Partial Orders . . . 263
22.2 Weaker Specs by Abandoning Abstract States . . . 268
22.3 Implementing Compass Specs in iRC11 . . . 269

23 Verifications of Stacks and Queues 273
23.1 Queue Specs and Verification of the Michael-Scott Queue . . . 273
23.2 Stack Specs and Verification of the Treiber Stack . . . 287

24 Exchangers and the Elimination Stack 291
24.1 The Elimination Stack . . . 291
24.2 A Strong Spec for Exchangers . . . 292
24.3 Verifying the Elimination Stack . . . 295

25 Related Work 297

26 Conclusion 301

Bibliography 305

List of Figures

1.1 Dependency graph of this dissertation's chapters and the concepts that they introduce. . . 7

2.1 Message-Passing examples in C11/RC11. 16
2.2 Candidate executions of several MP examples. 20
2.3 Illustrations of derived relations. 22
2.4 A racy execution of a racy MP program. 23
2.5 Several *forbidden* (inconsistent) executions in C11/RC11. 24
2.6 Load-buffering (LB) and Out-of-thin-air (OOTA) behaviors. 25

3.1 View-based explanation of MP behaviors. 29
3.2 Computations of post thread-views for read and write operations. 36
3.3 View-based machine semantics. 37
3.4 Data-race free (DRF) pre-conditions. 40
3.5 Data-race free (DRF) post-conditions. 42

4.1 The relaxed λ_{Rust} language syntax. 48
4.2 Some syntactic sugars for λ_{Rust}. 49
4.3 CPS notations for λ_{Rust}. 50
4.4 Relaxed λ_{Rust} expression semantics. 52
4.5 The combined 1-thread semantics of ORC11 machine semantics and λ_{Rust} expression semantics. 54
4.6 Threadpool semantics. 55

6.1 An excerpt of Iris grammar. 64
6.2 Basic rules of several Iris connectives. 65
6.3 Basic rules of Iris ghost ownership and basic updates. 66
6.4 Some rules for Iris invariants and fancy updates. 68
6.5 Some common rules for Iris weakest pre-conditions and Hoare triples. 71
6.6 Some rules for Iris weakest pre-conditions and invariants. 72
6.7 Some properties of timeless propositions and persistent propositions. 73
6.8 Several rules for the $\textsc{Auth}(M)$ RA. 74

7.1 Pure primitive WPs in the RMC base logic 80
7.2 Main properties of the base logic's local assertions 83
7.3 The base logic's primitive Hoare rules for fences 85
7.4 The base logic's primitive Hoare rules for non-atomic reads and writes 86
7.5 The base logic's primitive Hoare rules for atomic reads and writes 87
7.6 The base logic's primitive Hoare rule for CASes 88
7.7 The base logic's primitive WP rule for CASes 91
7.8 Several agreements between the global ghost state and local assertions 95
7.9 Several update rules for the global ghost state and local assertions 95

8.1 iRC11 rules for fence modalities 103
8.2 iRC11 rules for objective propositions and the objective modality 106
8.3 iRC11 rules for view-explicit modalities 108

8.4 iRC11 rules for the subjective modality . . . 110

9.1 Rules for iRC11 non-atomic points-to . . . 112

10.1 Basic properties of assertions related to the atomic points-to . . . 117
10.2 Conversions between the non-atomic and atomic points-to assertion . . . 119
10.3 iRC11 read rules with the atomic points-to assertion . . . 121
10.4 iRC11 write rules with the atomic points-to assertion . . . 123
10.5 An iRC11 CAS rule with the atomic points-to assertion . . . 125
10.6 An iRC11 CAS rule with the atomic points-to in single-writer mode . . . 126
10.7 Several properties of ghost abstractions for the atomic RA . . . 128

11.1 iRC11 rules for objective invariants . . . 135
11.2 iRC11 rules for cancelable invariants . . . 137
11.3 Stronger iRC11 rules for cancelable invariants . . . 139
11.4 Properties of the RA FRACVIEWR for cancelable invariants . . . 142
11.5 The interface of non-atomic invariants . . . 144

12.1 Message-Passing with Loops . . . 146
12.2 Hoare proof outlines for **mp** . . . 148
12.3 Message-Passing with Reclamation . . . 150
12.4 Hoare proof outlines for **mp_reclaim** . . . 151
12.5 Derived iRC11 atomic access rules with the view-join modality . . . 153
12.6 A Spawn-and-Join library . . . 154
12.7 Hoare proof outlines for SPAWN-SPEC . . . 156
12.8 A simple release-acquire implementation for Treiber stacks . . . 157
12.9 Bag or per-element specifications for Treiber stacks . . . 159
12.10 Hoare proof outlines for **try_push_swap** . . . 163
12.11 Hoare proof outlines for **try_pop** . . . 164

14.1 Key rules for cancelable invariants in Iris-SC . . . 175

15.1 Selected rules of SC RustBelt's lifetime logic . . . 182
15.2 The life cycle of borrows and lifetimes . . . 182
15.3 MP verified with the lifetime logic in Iris-SC. . . . 184

16.1 More selected rules for lifetimes and full borrows, ported to λ_{Rust} + ORC11 . . . 189
16.2 Selected rules for other borrow alternatives, sound in λ_{Rust} + ORC11 . . . 193

17.1 Rules for GPS Persistent Concurrent Protocols . . . 204
17.2 CAS Rules for GPS Persistent Concurrent Protocols . . . 206
17.3 Rules for auxillary assertions of GPS Single-Writer Protocols . . . 210
17.4 Selected rules for Cancelable Single-Writer GPS Protocols . . . 213
17.5 Selected basic rules for Atomic-Borrows-based GPS Protocols . . . 215
17.6 Selected read and write rules for Atomic-Borrows-based GPS Protocols . . . 217
17.7 A CAS rule for Atomic-Borrows-based GPS Protocols . . . 218
17.8 Selected rules for assertions of middleware GPS protocols . . . 220

18.1 A λ_{Rust} version of Rust's RMC `RwLock<T>` . . . 228

19.1 Implementation of Core `Arc` . . . 239

19.2 Counting permissions for Core `Arc` . . . 241
19.3 A truncated history of the `Arc` counter . . . 247

21.1 Specifications of Queue operations, from sequential, to SC concurrency and strong RMC . 259
21.2 A Message-Passing (MP) client with Queues . . . 261

22.1 Compass Specs for Queues . . . 264
22.2 A proof sketch of Message Passing with queues . . . 267

23.1 Full $\texttt{LAT}_{\texttt{hb}}^{\text{abs}}$ specs for queue . . . 274
23.2 A release-acquire Michael-Scott queue . . . 275
23.3 An RMC Herlihy-Wing Queue . . . 277
23.4 $\texttt{LAT}_{\texttt{hb}}$ specs for stack . . . 288
23.5 $\texttt{LAT}_{\texttt{hb}}$ Stack Consistency . . . 289

24.1 $\texttt{LAT}_{\texttt{hb}}$ specs for exchangers (excerpt, simplified). . . . 293

List of Tables

16.1 Comparison of borrow types 192

18.1 A summary of Rust types for `RwLock<T>` and its lock guards 226

19.1 An excerpt of Rust's `Arc<T>` and `Weak<T>` APIs 245
19.2 Rust's implementation (excerpt) of `Arc::get_mut` and `Arc::drop` 246

Glossary

SC Sequential Consistency

RMC Relaxed Memory Consistency/Concurrency

CSL Concurrent Separation Logic

C11 C/C++ 2011 Standards

RC11 Repaired C11

ORC11 Operational Repaired C11

1

Introduction

Reasoning about concurrency is hard, due to the explosion of possible interactions between threads running in parallel. In the traditional concurrency model of *sequential consistency*[1], every thread takes turns to execute its atomic instructions, and the behavior of a concurrent program is defined as all interleavings of all threads' atomic instructions. As such, if one needs to verify some property of the program, one would need to check that property for every possible interleaving of the atomic instructions performed by the threads. This is low level and hard to scale: if we want to compose our *verified* libraries, then we would have to look at the compositions of their interleavings, and we would have to make sure that the properties they have been verified against are compatible with interleaving composition. In order to scale verification to more intricate programming language features and algorithms, we need more *abstract* and *modular* reasoning principles.

[1] Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs" [Lam79].

CONCURRENT SEPARATION LOGICS[2] (hereafter, CSLs) provide a feasible approach to abstract and modular control of *interference*: instead of thinking in terms of interleavings, we can reason about each thread more modularly by thinking in terms of the *resources* that the thread *owns*. The resources owned by each thread are "separated" from those of other threads, and encode the thread's *permissions* on the shared memory's fragments that it owns. As a result, they can restrict how other threads may interfere with the current thread's execution. This "separation" idea has led to long research lines on highly expressive logics or logic frameworks[3] that have been applied to various sophisticated concurrency verification problems. Among these problems includes reasoning about realistic *relaxed memory concurrency*[4]—the main focus of this dissertation.

[2] O'Hearn, "Resources, concurrency, and local reasoning" [O'H07]; Brookes, "A Semantics for Concurrent Separation Logic" [Bro07].

[3] Just to list a few: [VP07; FFS07; Fen09; Fu+10; DY+10; JB12; SB14; RPDG14; RP+16; Nan+14; SWT18; Kro+20; TDB13; Jun+15; Jun+18b; Cha+21; FKB21; G+22].

[4] [VN13; TVD14; DV16; DV17; Kai+17; Sve+18; He+18; Dan+20a; MJP20].

RELAXED MEMORY CONCURRENCY. Sequential consistency (hereafter, SC)—the interleaving model of concurrency in which threads take turns accessing the global state, and all threads share the same view of that state—does not reflect what is going on in modern multicore programming languages. In reality, multicore hardware employ rich hierarchies of *caches* to improve memory access performance, with which a CPU's write may not immediately reach the main memory, or may not be immediately visible to all other cores, or may not be visible to all other cores at the same time. To further improve performance, both hardware

and compilers can analyze dependencies of memory accesses to apply optimizations: if the effects of two memory accesses are independent, they can be executed independently. In short, from the perspective of programmers, memory accesses instructions can be executed *out-of-order* in modern programming languages.

To match this modern reality, we need models of so-called *relaxed memory concurrency* (hereafter, RMC) at the programming-language level that provide an abstraction over different hardware architectures and compilers. However, due to the complexity of hardware behaviors and desirable optimizations, the formal semantics of RMC models (at both hardware level and language level) still require extensive ongoing research.[5] Nevertheless, the goal of this dissertation is *not* to find the right model that captures all relaxed memory features. Here, I take as assumption a language-level memory model whose features have stabilized over years of research, and present how to build RMC separation logics that can scale up to very substantial verification efforts.

[5] [Bat+11; Kan+17; Pul+18; Flu+17; Lah+17; Pul+19; CV19; PLV19; Lee+20; Cho+21b; Sim+20; Cho+22; Lee+23].

1.1 Reasoning about Relaxed Memory Concurrency

This dissertation focuses on the relaxed memory model of C/C++, which was first proposed in the C++11 standard and was formalized by Batty et al.,[6] and is now broadly adopted[7] by the RMC models of Rust, Java, OCaml, JavaScript, and WebAssembly.[8] The C/C++ RMC model (hereafter, C11) supports a variety of different *consistency levels* for shared-memory accesses, which intuitively dictate how much reordering can be applied to the accesses. For programmers who demand the simpler SC concurrency model where there is strong *synchronization* between threads (so that they have the same view of shared memory), SC accesses are available.[9] This strength, however, comes at the cost of disabling reordering optimizations and inserting expensive memory fences into the compiled code. The weaker consistency levels of *release/acquire* and *relaxed* allow one to trade off synchronization strength in return for more efficient compiled code. These different consistency levels are widely employed in performance-critical concurrency libraries such as locks, reference-counting, stacks, queues, read-copy-update (RCU), and so on.

[6] Batty et al., "Mathematizing C++ concurrency" [Bat+11].

[7] potentially partially, with minor modifications or simplifications.

[8] [BP19; WRP19; DSM18; Wat+20].

[9] The word "consistency", used *e.g.*, in sequential consistency or relaxed memory consistency, can be understood as the consistency among the views of the threads (or processors) on shared memory.

Compared to SC, reasoning about RMC is significantly more complicated: relaxed-memory programs have many more behaviors depending on which consistency levels are employed. In fact, some useful reasoning principles in SC logics are no longer sound for reasoning about relaxed behaviors. Furthermore, such behaviors are defined in C11 not in the familiar style of interleavings, but by an *axiomatic* semantics, in which the allowed behaviors of a program are defined by enumerating candidate executions (represented as "event graphs") and then restricting attention to the executions that obey various coherence axioms. Vafeiadis et al. overcome these challenges and provided the first abstract and modular reasoning principles for C11 in form of various RMC separation logics.[10]

[10] [VN13; DV16; DV17; TVD14].

However, in building these logics, Vafeiadis et al. were not able to use the standard model of Hoare-style program specifications from

prior CSLs because notions like "the machine states before and after executing a command c" do not have a clear meaning in C11's axiomatic semantics. Instead, they had to come up with new, non-standard models of separation logic in terms of predicates on event graphs. Unfortunately, the complexity of these new models has made them challenging to adapt and extend to more complex settings, for example in verifying Rust's type system. Furthermore, although the soundness of these logics has been verified formally in Coq, there has thus far been no tool support to perform machine-checked verifications of RMC programs or libraries in these logics.

To scale the reasoning principles of concurrent separation logic to realistic languages like C/C++ or Rust, which encompass many interweaving complex features, we need a few ingredients: (1) strong but abstract reasoning principles so that we can avoid the too tedious details of the underlying concurrency model; (2) modular reasoning so that we can compose smaller verification results into larger ones; (3) reasoning extensibility so that we can derive new reasoning principles for both complex language features and algorithms without rebuilding our logic from scratch; and (4) machine-checked verifications so that we do miss potential bugs in our proofs—both in soundness proofs of our logics and in program verifications. Only recently was it possible to acquire these ingredients at once with the CSL framework Iris,[11] which comes with strong tactic support in Coq.[12] Using Iris, Jung et al.[13] have verified the soundness of the Rust's type system, and thus have demonstrated the scalability of CSLs to complex languages such as Rust, even though only for the SC memory model. Meanwhile, my collaborators and I had previously re-proven the soundness of Vafeiadis et al.'s RSL and GPS logics in Iris, and demonstrated the possibility of building extensible RMC separation logics, even though only for a small fragment of the C11 model.[14]

I, together with my collaborators, have developed strong, abstract, modular, extensible, and machine-checked RMC separation logics in Iris that scale to substantial verification efforts, for an also substantial fragment of C11 whose features have stabilized over years of research, namely the RC11 (Repaired C11) model.[15] In this dissertation, I present the abstractions needed to build such logics. I report two main contributions that rely on those logics:

1. RustBelt Relaxed:[16] the soundness proof of the Rust's type system in RMC, where relaxed memory effects are encapsulated behind the safe interface of libraries and thus are not visible to clients; and
2. Compass:[17] the compositional specification and verification of relaxed memory libraries, where relaxed memory effects are exposed to clients of such libraries.

[11] Jung et al., "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning" [Jun+15]; Jung et al., "Higher-order ghost state" [Jun+16]; Krebbers et al., "The Essence of Higher-Order Concurrent Separation Logic" [Kre+17]; Jung et al., "Iris from the ground up: A modular foundation for higher-order concurrent separation logic" [Jun+18b].

[12] Krebbers et al., "Interactive Proofs in Higher-Order Concurrent Separation Logic" [KTB17]; Krebbers et al., "MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic" [Kre+18].

[13] Jung et al., "RustBelt: Securing the Foundations of the Rust Programming Language" [Jun+18a].

[14] Kaiser et al., "Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris" [Kai+17].

[15] Lahav et al., "Repairing sequential consistency in C/C++11" [Lah+17].

[16] Dang et al., "RustBelt Meets Relaxed Memory" [Dan+20a].

[17] Dang et al., "Compass: strong and compositional library specifications in relaxed memory separation logic" [Dan+22].

1.2 RustBelt Relaxed: Verifying the Soundness of Rust's Type System in RMC

Rust[18] is a young and evolving programming language that aims to

[18] Klabnik and Nichols, *The Rust Programming Language* [KN18].

bring safety to systems programming. Specifically, Rust provides low-level control over data layout and resource management à la modern C++, while at the same time offering strong high-level guarantees (such as type and memory safety) that are traditionally associated with safe languages like Java. In fact, Rust takes a step further, statically preventing more forms of anomalous behavior, such as data races and iterator invalidation, that safe languages typically fail to rule out. Rust strikes its delicate balance between safety and control using a *substructural* type system, in which types not only classify data but also represent *ownership* of resources, such as the right to read, write, or reclaim a piece of memory. By tracking ownership in the types, Rust is able to prohibit dangerous combinations of mutation and aliasing, a well-known source of programming pitfalls and security vulnerabilities in C/C++ and Java.

Nevertheless, Rust's ownership-based type system is not always expressive enough to type-check very delicate programming idioms, *e.g.*, some pointer-based data structures, synchronization abstractions, garbage collection mechanisms. To allow for these mechanisms, Rust supports extension to the type system via *libraries* whose implementations internally utilize *unsafe features* (*e.g.*, unchecked type casts, array accesses without bounds checks, or accesses of "raw" pointers who are untracked by the type system). Given that these libraries are not checked by the type system, it is now the responsibility of libraries developers to make sure that these extensions are actually *safe*, in the sense that they have properly encapsulated the uses of unsafe features within their "safe APIs". Unfortunately, as the language is evolving and libraries are being updated or created, it is not clear what such encapsulation formally means.

[19] Jung et al., "RustBelt: Securing the Foundations of the Rust Programming Language" [Jun+18a].

RustBelt[19] is the first work on the formal foundations of the Rust programming language, in which it covers not only the soundness of the ownership-based type system, but also the safe encapsulation by Rust's extensions via libraries. RustBelt managed to formalize such interactions between the type system and the extensions in the presence of complex language features like recursive types and higher-order state. Furthermore, all proofs were machine-checked in Coq. Unfortunately, while ground-breaking, RustBelt assumes the SC memory model. Therefore, even though RustBelt's results increase the confidence in the safety of Rust's type system and libraries, the results cannot yet be applied to actual Rust code, which relies on the C11 memory model.

To circumvent this problem, we developed RustBelt Relaxed (or $\mathsf{RB}_{\mathtt{rlx}}$, for short), the first formal validation of the soundness of Rust under RMC. Although based closely on the original RustBelt, $\mathsf{RB}_{\mathtt{rlx}}$ takes a significant step forward by accounting for the safety of the more weakly consistent memory operations that real concurrent Rust libraries actually use. For the most part, we were able to verify Rust's uses of relaxed-memory operations as is. Only in the implementation of one Rust library (`Arc`) did we need to strengthen the consistency level of two memory reads (from relaxed to acquire) in order to make our verification go through. And in one of these cases, our attempt to verify the original (more relaxed) access led us to expose it as the source of a previously undetected data race in the library. Our fix for this race has since been merged into the

Rust codebase.[20]

[20] Jourdan, *Insufficient synchronization in Arc::get_mut* [Jou18].

SYNCHRONIZED GHOST STATE. The main technical challenge of porting RustBelt to RMC is relevant not just to Rust but to relaxed-memory verification in general: namely, that **existing work on separation logic does not provide an adequate foundation for reasoning about resource reclamation under relaxed memory**. Resource reclamation under relaxed memory intertwines *resource accounting* and *physical synchronization*: one needs to make sure that resources provided to every client thread must be returned completely and with proper synchronization. As a result, more care is needed when designing and proving relaxed memory reasoning rules. Fortunately, in $\mathsf{RB}_{\mathtt{rlx}}$ we show that changes in the rules needed to support reclamation are minimal and can be handled fairly routinely, thanks to a novel notion of *synchronized ghost state*: ghost state that is tied to physical synchronization so that it can be used for safe, well-synchronized resource accounting.

1.3 Compass: Strong and Compositional Specifications of Relaxed-Memory Libraries

Existing RMC separation logics have been applied to verify tricky RMC algorithms such as locks, stacks, queues, read-copy-update,[21] and reference counting,[22] as well as the $\mathsf{RB}_{\mathtt{rlx}}$ work. However, these works (except Cosmo[23]—see more below) only verify implementations against some "reasonable" specifications that are sufficient for their respective purposes, but do not necessarily capture their full functional correctness. For example, as we will see, even with unsafe features, the RMC libraries verified in $\mathsf{RB}_{\mathtt{rlx}}$ only need specifications strong enough to verify the soundness of Rust's type system, which focuses on safety and does not expose relaxed behaviors to users. As another example, the queue specification in GPS[24] only captures the fact that a dequeue is synchronized with the enqueue that it is matched with, but *not* the standard first-in-first-out (FIFO) property of queues. Stronger functional correctness CSL specifications (from now on, *specs* for short) for RMC libraries thus are needed, especially for clients that build new libraries out of smaller ones and rely on certain relaxed behaviors of the constituent libraries to verify their library's implementation.

[21] Tassarotti et al., "Verifying read-copy-update in a logic for weak memory" [TDV15].

[22] Doko and Vafeiadis, "Tackling Real-Life Relaxed Concurrency with FSL++" [DV17].

[23] Mével et al., "Cosmo: a concurrent separation logic for multicore OCaml" [MJP20].

[24] Turon et al., "GPS: navigating weak memory with ghosts, protocols, and separation" [TVD14]; Kaiser et al., "Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris" [Kai+17].

However, unlike in the SC setting, in RMC verification research there is no canonical way to specify full functional correctness of a library that may expose relaxed behaviors. While *linearizability*[25] is the de facto standard correctness condition for concurrent libraries, it does not extend to many highly concurrent libraries, including those in RMC: these libraries tend to have less synchronization or control, and it may be that a *linearization* is extremely difficult to construct (*e.g.*, Herlihy-Wing queue) or that the library has no useful sequential behaviors (*e.g.*, exchangers[26]). Therefore, various linearizability-like criteria have been proposed as alternatives,[27] especially for relaxed memory.[28] These works essentially share one basic idea in relaxing linearizability: instead of requiring a *total order* on a library's operations, one only requires that

[25] Herlihy and Wing, "Linearizability: A Correctness Condition for Concurrent Objects" [HW90].

[26] [SLS05; HRV15].

[27] [Hen+13; JR14; Der+14; Haa+16; Nei94; AKY10; Bur+14; CRR15].

[28] [Bur+12; BDG13; Jag+13; Doh+18; Don+18; Raa+19; EE19; Kri+20].

operations respect some *partial orders*. These works, however, have little support for modular client reasoning. Therefore, we want to improve the proposed relaxations of linearizability with *Hoare-style* specs to support better modular reasoning about clients who rely on strong correctness guarantees of RMC libraries.

Accordingly, our starting point is *logical atomicity*,[29] a key proof technique to achieve strong specs *and* modular client reasoning in (SC) CSLs. Logically atomic specs are similar to Hoare-triple based specs, but they allow *atomic access* to the exact, up-to-date *abstract state* of the data structure. As such, they provide the abstraction that an operation takes effect atomically on the data structure's abstract state, so that clients can build a concurrent protocol to govern how the data structure is used (how the state can evolve). If the client wants to compose multiple data structures, they can further build a protocol for multiple abstract states, all the while enjoying the benefits of separation logics.

[29] Rocha Pinto et al., "TaDA: A Logic for Time and Data Abstraction" [RPDG14]; Svendsen and Birkedal, "Impredicative Concurrent Abstract Predicates" [SB14]; Jung et al., "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning" [Jun+15]; Jung et al., "The future is ours: prophecy variables in separation logic" [Jun+20].

Logical atomicity has been applied mostly in the SC setting, and only recently did Mével and Jourdan[30] demonstrate its use to give stronger CSL specs for RMC libraries. Unsurprisingly, the application of the technique needs to account for relaxed behaviors: Mével and Jourdan needed to combine logical atomicity with the tracking of some *synchronization* information among library operations, reminiscent of the partial orders from the relaxations of linearizability. But they only needed limited synchronization tracking, because their logic, Cosmo,[31] is sound only for the Multicore OCaml memory model,[32] and they only gave one spec for a concurrent queue and verified one client.

[30] Mével and Jourdan, "Formal verification of a concurrent bounded queue in a weak memory model" [MJ21].

[31] Mével et al., "Cosmo: a concurrent separation logic for multicore OCaml" [MJP20].

[32] Dolan et al., "Bounding data races in space and time" [DSM18].

Consequently, the Cosmo-style specs does not scale to libraries or clients that rely on *interacting* relaxed behaviors. More specifically, while Cosmo specs expose *internal* (to the implementation) synchronizations among operations, they do not take into account how additional *external* synchronizations created by clients or other libraries can affect the behaviors of the library in question.

Logical Atomicity and Richer Partial Orders. We generalize Mével and Jourdan's approach by combining **logical atomicity** with **richer partial orders** inspired by the relaxations of linearizability, so that we can give stronger specs for more weakly consistent libraries, in the more relaxed memory model RC11. But, given the plethora of partial orders from those relaxations of linearizability, which one should we use? We believe the *event-graph* based criteria proposed by Raad et al.[33] ("Yacovet") are the most general, because in that framework a verifier can give a library stronger or weaker specs by choosing the partial orders they prefer and by stating suitable library-specific *consistency conditions* on the partial orders. Therefore, we decided to encode Yacovet criteria in our separation logic and enhance them further with logical atomicity. As such, we can give strong and compositional Hoare-style specs for RMC libraries, with better support for modular client reasoning, in a new framework called Compass. We demonstrate the strength, satisfiability, and support for client reasoning of our specs with multiple mechanized libraries and client verifications.

[33] Raad et al., "On library correctness under weak memory consistency: specifying and verifying concurrent libraries under declarative consistency models" [Raa+19].

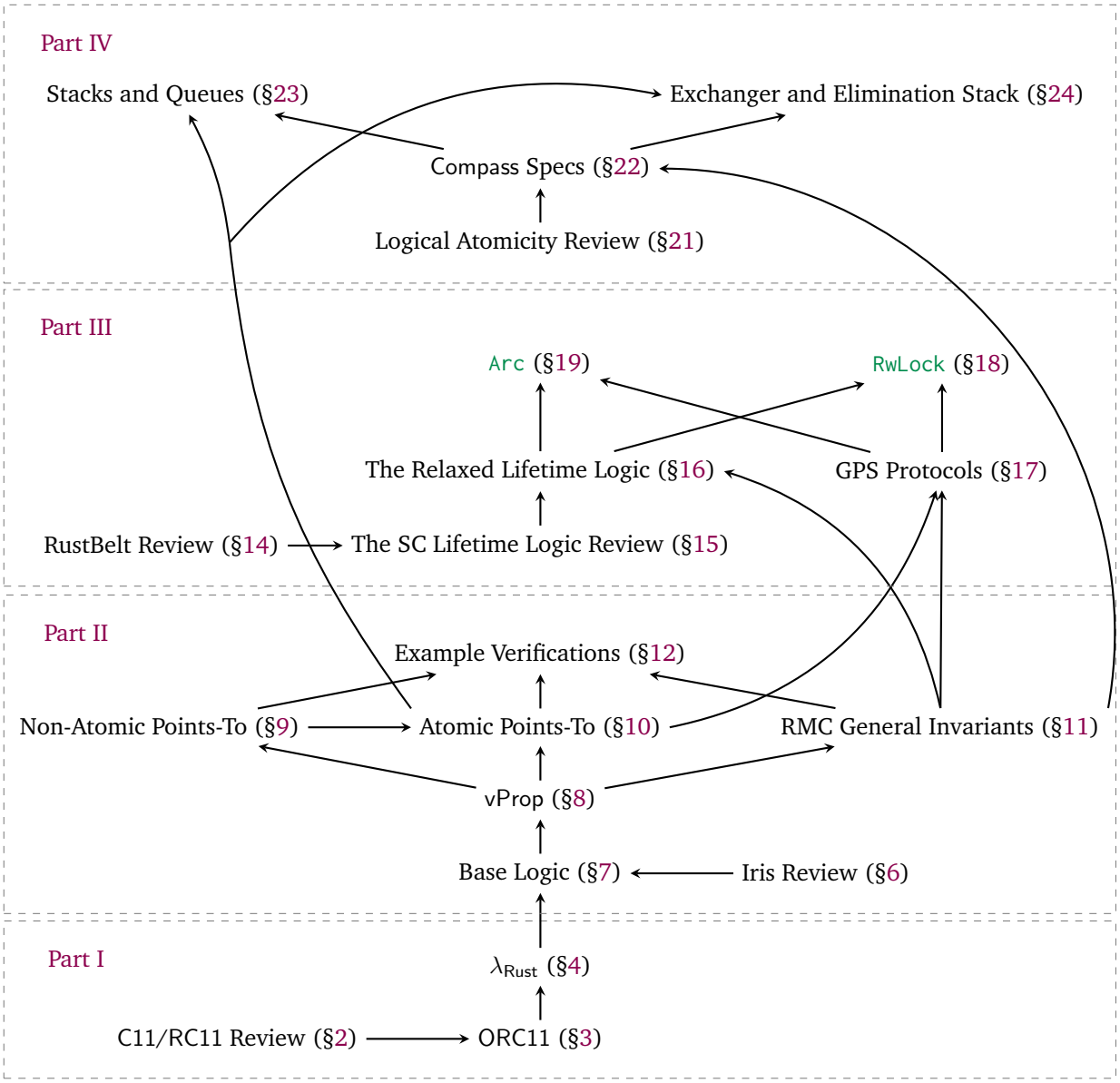


FIGURE 1.1: Dependency graph of this dissertation's chapters and the concepts that they introduce.

1.4 Structure

This dissertation is composed of three parts: Part II presents the basic layers needed to build RMC separation logics with Iris, while Part III and Part IV discuss how such logics can be extended and/or applied for RustBelt Relaxed and Compass, respectively. Part III and Part IV are independent from each other, but both rely on materials presented in Part II. Each part will discuss the context, the challenges, the solutions, and the results separately, as well as related and future work in detail. The conclusion (Chapter 26) provides a high level summary and potential future research directions. Note that Figure 1.1 (page 7) provides the dependency graph for the main chapters in this dissertation.

Part I presents ORC11, an *operational* variant of RC11 that is needed to instantiate Iris. It provides a brief background review on relaxed memory models, which readers who are familiar with the topics can skip. The most important feature of ORC11 is its *race detector*—an operational account for *data races*, which need meticulous care and significantly complicate the soundness proof of iRC11.[34]

[34] It indeed delayed the publication of the $\mathsf{RB}_{\mathtt{r1x}}$ work by a year.

Part II discusses the features and the construction of iRC11, our extensible RMC separation logic for ORC11. It first provides a review of the Iris framework. The main chapters of Part II flesh out the abstraction layers needed to build the various core reasoning principles of iRC11: its *modalities*, and its *non-atomic* and *atomic* points-to assertions, and its forms of invariants, including *cancelable invariants* that employ *synchronized ghost state*. The atomic points-to assertion is a novel contribution of this dissertation that has not been published elsewhere. The extensibility of the construction will be demonstrated by the fact that iRC11 not only can incorporate *all* reasoning principles from all other RMC separation logics, but also can extend and combine them with iRC11's own novel reasoning principles.

Part III discusses the proofs of the RustBelt Relaxed work. It first provides an overview of Rust and RustBelt, and briefly explains the soundness proof the Rust's type system, which crucially depends on the *lifetime logic*. The remaining chapters of Part III elaborate on how iRC11's *synchronized ghost state* and *cancelable invariants* are used in re-proving the lifetime logic and in re-verifying the concurrent standard libraries of Rust that use relaxed memory operations (*e.g.*, `Mutex`, `RwLock`, or `Arc`). The library verifications depend crucially on a combination of cancelable invariants and *GPS single-location protocols*.[35] A bit of history on how the bug in `Arc` was found will be provided.[36]

[35] Turon et al., "GPS: navigating weak memory with ghosts, protocols, and separation" [TVD14]; Kaiser et al., "Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris" [Kai+17].

[36] Jourdan, *Insufficient synchronization in Arc::get_mut* [Jou18].

Part IV presents the Compass specification framework. It starts by reviewing logical atomicity in both SC and RMC settings, as well as the event-graph based Yacovet specs. It then presents how to encode Yacovet specs in iRC11 with logical atomicity. The remaining chapters present the library verifications and client verifications of various RMC data structures, relying on a general notion of *multi-location invariants* in combination with the atomic points-to assertions. I will also touch on the topic of *helping* (cooperation) with logical atomicity, and its role in the specs of exchangers. Some of the specifications and verifications are

the first-ever performed in the relaxed memory setting.

1.5 Publications and Collaborations

This dissertation contains the work of the following two papers:

- [Dan+20a]: Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, Derek Dreyer. "RustBelt Meets Relaxed Memory", appeared in POPL 2020.
- [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", appeared in PLDI 2022.

While reusing much of the text from these two papers, the dissertation provides substantially more in-depth details—many of which have not been presented before—in a coherent structure. The following contents are new and have not been discussed elsewhere:

- §3.4: the details of ORC11's race detector;
- §7: the detailed model of the iRC11 base logic;
- §8: the models of various iRC11's modalities;
- §9: the model of the *non-atomic* points-to assertion, which depends tightly on ORC11's race detector;
- §10: the model of the *atomic* points-to assertion;
- §11: the detailed interfaces and models of iRC11 *objective invariants* and *cancelable invariants*;
- §12: several example verifications demonstrating many *intermediate* abstractions provided by iRC11;
- §16: more details on how the lifetime logic was ported to iRC11;
- §17: the detailed model of GPS *single-location protocols*, built atop atomic points-to;
- §19: the detailed verification of Rust's standard library `Arc`;
- §22: the detailed interpretations of Compass specs in iRC11 with logical atomicity;
- §23: the verifications of stacks and queues against Compass specs;
- §24: a more complete spec of the exchanger with *helping*.

Some of the ideas in this work were originally developed in iGPS ([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", appeared in ECOOP 2017) of which I was a co-author. Although iGPS is not a part of this dissertation, I contributed to those ideas and have ported them fully into iRC11.

COLLABORATIONS. The two papers mentioned above, on which this dissertation is based, are the results of delightful collaborations. Although I led the efforts in both works, they would not have been achievable without the team efforts of many fellow researchers.

For $\mathsf{RB}_{\mathtt{rlx}}$, the ORC11's race detector and the model of GPS protocols were inspired by those developed for iGPS, which in turn was the result of collaborations with Jan-Oliver (Janno) Kaiser. The flaw of the initial ORC11's race detector was found by Derek Dreyer, and after I fixed the design, it was Janno that led the (on-paper) correspondence proof between RC11 and ORC11. I proved most of the soundness of the iRC11 logic, but I collaborated with Jacques-Henri Jourdan to construct the models of several iRC11 modalities. It was Jacques-Henri who used iRC11 to re-prove the soundness of the lifetime logic. I re-verified the Rust concurrent libraries by substantially extending the original proofs in SC RustBelt.[37] It was also Jacques-Henri's original suggestion to re-prove the model of GPS protocols directly on top of iRC11, instead of on top of the base logic from Iris' instantiation as in iGPS. I only completed that task 2 years later.

[37] Jung et al., "RustBelt: Securing the Foundations of the Rust Programming Language" [Jun+18a].

For Compass, I encoded the Yacovet specs in iRC11 with logical atomicity, and verified library implementations against those specs. I collaborated with Jaehwang Jung and Jaemin Choi to refine those specs to cater to the *linearizablity-style* specs, but those specs are not included in this dissertation. Together with all other co-authors, we performed the client verifications that used the specs reported in [Dan+22].

COQ ARTIFACTS. Unless noted explicitly, all definitions and proofs in this dissertation are formalized in Coq. The follow repositories contain the respective Coq developments and instructions for how to build and use them.

- ORC11: https://gitlab.mpi-sws.org/iris/gpfsl/-/tree/master/orc11
- iRC11: https://gitlab.mpi-sws.org/iris/gpfsl/-/tree/master/gpfsl
- $\mathsf{RB}_{\mathtt{rlx}}$: https://gitlab.mpi-sws.org/iris/lambda-rust/-/tree/masters/weak_mem
- Compass: https://gitlab.mpi-sws.org/iris/gpfsl/-/tree/master/gpfsl-examples

Part I

OPERATIONAL SEMANTICS FOR RELAXED MEMORY

This thesis discusses the features and the construction of iRC11, a concurrent separation logic that is sound for the relaxed memory model RC11.[38] We do not assume prior knowledge either on relaxed memory models or concurrent separation logics. Therefore, in this part, we discuss the semantics of relaxed memory concurrency. We start with Chapter 2 to review relaxed memory models defined in *axiomatic* style, specifically for the C11[39] and RC11 models. Readers familiar with RMC can freely skip this review, unless they are interested in the specific details of the RC11 model. Then, in Chapter 3, we present our first contribution: ORC11, an *operational* version of RC11 that is geared to complement the λ_{Rust} language used in RustBelt,[40] which is presented in Chapter 4. Developing an operational semantics for RC11 is a necessary prerequisite for instantiating Iris. The bottom half of Figure 1.1 visualizes the dependency among these chapters.

[38] Lahav et al., "Repairing sequential consistency in C/C++11" [Lah+17].

[39] Batty et al., "Mathematizing C++ concurrency" [Bat+11].

[40] Jung et al., "RustBelt: Securing the Foundations of the Rust Programming Language" [Jun+18a].

2

Background: Relaxed Memory Models

The goal of hardware and language relaxed memory models is to give an abstraction for the possibility (or impossibility) of *out-of-order* behaviors for relaxed memory accesses, which are induced by hardware and/or compiler optimizations. The models can be defined in form of either *operational* or *axiomatic* semantics. RMC operational semantics typically involve some kind of buffers (*e.g.*, write buffers in x86-TSO)[1] to delay the effects of memory accesses and thus make them appear out-of-order. Axiomatic semantics, on the other hand, define a set of constraints (*axioms*) on several *partial orders* among memory accesses in a *candidate execution*—accesses not so tightly ordered can thus be executed out-of-order. In this chapter, we review the axiomatic semantics of C11[2] and RC11.[3] More specifically, we review the intuitive semantics of C11 in §2.1, then a formal excerpt of RC11's partial orders and axioms in §2.2. In §3, we present ORC11, our operational version of RC11.

[1] Sewell et al., "x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors" [Sew+10].

[2] Batty et al., "Mathematizing C++ concurrency" [Bat+11].

[3] Lahav et al., "Repairing sequential consistency in C/C++11" [Lah+17].

2.1 C11, Intuitively

The C11 memory model offers several different modes of memory accesses, including non-atomic (**`na`**), relaxed (**`rlx`**), release (**`rel`**), acquire (**`acq`**), and sequentially consistent (**`sc`**). Non-atomic accesses are "normal" data accesses, meaning that it is the programmer's responsibility to ensure that they are properly synchronized through other means. If they are not properly synchronized—*i.e.*, there is a data race involving non-atomics—then C11 says the whole program has *undefined behavior*, or UB for short. The remaining modes, collectively called *atomic* accesses, are allowed to be racy and are indeed used to establish synchronization among non-atomic accesses.

Example 2.1 (Message-Passing)**.** To explain what synchronization actually means, we explore the *Message-Passing* examples in Figure 2.1. In Example 2.1(a), we initialize two memory locations ℓ_x and ℓ_y to 0 non-atomically, then spawn two threads π (on the left) and ρ (on the right). Thread π intends to pass a "message" to ρ. The message, 42, is stored in ℓ_x (line $\pi 1$). π then sets the boolean flag ℓ_y to 1 (line $\pi 2$), to signal to ρ that the message is ready to be received. Once ρ sees the flag set (line $\rho 1$), it attempts to read the message from ℓ_x (line $\rho 2$). However, both the intended value of 42 as well as the initial value of 0 could be

$\ell_x :=_{\texttt{na}} 0; \ell_y :=_{\texttt{na}} 0;$

$\pi 1$:	$\ell_x :=_{\texttt{rlx}} 42;$	$\rho 1$:	**if** $^{*\texttt{rlx}}\ell_y \mathrel{!}= 0$ **then**
$\pi 2$:	$\ell_y :=_{\texttt{rlx}} 1;$	$\rho 2$:	$^{*\texttt{rlx}}\ell_x;$ // 0 or 42

((a)) MP with relaxed accesses.

$\ell_x :=_{\texttt{na}} 0; \ell_y :=_{\texttt{na}} 0;$

$\pi 1$:	$\ell_x :=_{\texttt{na}} 42;$	$\rho 1$:	**if** $^{*\texttt{sc}}\ell_y \mathrel{!}= 0$ **then**
$\pi 2$:	$\ell_y :=_{\texttt{sc}} 1;$	$\rho 2$:	$^{*\texttt{na}}\ell_x;$ // 42

((b)) MP with SC accesses.

$\ell_x :=_{\texttt{na}} 0; \ell_y :=_{\texttt{na}} 0;$

$\pi 1$:	$\ell_x :=_{\texttt{na}} 42;$	$\rho 1$:	**if** $^{*\texttt{acq}}\ell_y \mathrel{!}= 0$ **then**
$\pi 2$:	$\ell_y :=_{\texttt{rel}} 1;$	$\rho 2$:	$^{*\texttt{na}}\ell_x;$ // 42

((c)) MP with release-acquire accesses.

$\ell_x :=_{\texttt{na}} 0; \ell_y :=_{\texttt{na}} 0;$

$\pi 1$:	$\ell_x :=_{\texttt{na}} 42;$	$\rho 1$:	**if** $^{*\texttt{rlx}}\ell_y \mathrel{!}= 0$ **then**
$\pi 2$:	**fence**$_{\texttt{rel}}$;	$\rho 2$:	**fence**$_{\texttt{acq}}$;
$\pi 3$:	$\ell_y :=_{\texttt{rlx}} 1;$	$\rho 3$:	$^{*\texttt{na}}\ell_x;$ // 42

((d)) MP with relaxed accesses and fences.

FIGURE 2.1: Message-Passing examples in C11/RC11.

read. That is, even though ρ has read 1 from ℓ_y, it is not guaranteed to read 42 from ℓ_x. This is because the relaxed accesses of ℓ_y are not enough to establish synchronization between π and ρ.

In C11, threads are not synchronized by default: they each have their own perspective on the values in shared memory, and thus may observe memory events in different order. In Example 2.1(a), thread ρ may see that $\pi 2$ (π's write to ℓ_y) is executed out-of-order, before $\pi 1$ (π's write to ℓ_x), and therefore ρ reads 0 from ℓ_x in line $\rho 2$. What is happening under the hood is that hardware and/or compilers may deduce that π's writes are of *independent* memory locations, and thus may reorder them.[4]

[4]Note that from thread π's point of view, such reordering does not really matter as it cannot distinguish the effects, which, on the contrary, are distinguishable to the concurrently running thread ρ.

C11, however, also provides certain ways of performing accesses such that all threads can agree that one access is ordered before the other. In particular, the remaining examples in Figure 2.1 present several ways to create the *happens-before* relation between π's write to ℓ_x ($\pi 1$) and ρ's read from ℓ_x ($\rho 2$). We say to "establish synchronization" is to guarantee somehow that two memory events of interest are in the happens-before relation. Relaxed accesses are the weakest atomic accesses in C11 and do *not* guarantee happens-before. Thus, in Example 2.1(a), the relaxed accesses on ℓ_y do not establish synchronization between the accesses on ℓ_x.

SC ACCESSES (**sc**) are the strongest option to establish synchronization, and we use them in Example 2.1(b) for the accesses of ℓ_y.[5] If ρ's read of ℓ_y (line $\rho 1$) is not zero, then it reads from π's write of 1 to ℓ_y (line $\pi 2$). By C11's semantics of SC accesses, $\pi 2$ happens before $\rho 1$. Furthermore, SC accesses *prevent* all reorderings of other *intra-thread* accesses around them—*i.e.*, $\pi 1$ cannot be reordered to after $\pi 2$, and $\rho 2$ cannot be reordered to before $\rho 1$. As a result, we know that $\pi 1$ happens before $\rho 2$—or in other words, that ρ's read of ℓ_x is *synchronized with* π's write to it. Since the write of 42 is the most recent write to ℓ_x, we know that thread ρ must read 42 in $\rho 2$.

[5]According to C11, SC accesses can have subtle behaviors when mixed with other kinds of accesses. We refer interested readers to the RC11 paper ([Lah+17]) for more details. In this dissertation, we do not focus on SC accesses. We only mention them here for the purpose of demonstration.

RELEASE-ACQUIRE ACCESSES. Instead of using the costly SC accesses, we can use the release-acquire idiom to establish synchronization, as in Example 2.1(c). Here, π uses a release (**rel**) write in $\pi 2$, and ρ uses an

acquire (**acq**) read in $\rho 1$. If $\rho 1$ reads 1 from $\pi 2$, C11's release-acquire semantics on the location ℓ_x says that $\pi 2$ happens before $\rho 1$. Furthermore, a release write prevents reordering other intra-thread reads and writes that appear *before* it to *after* it, so, again, $\pi 1$ cannot be reordered to after $\pi 2$. Conversely, an acquire read prevents reordering other intra-thread reads and writes that appear *after* it to *before* it, so $\rho 2$ cannot be reordered to before $\rho 1$. Consequently, we still have $\pi 1$ happens before $\rho 2$. Note that release and acquire accesses are less costly to implement than SC accesses because they allow more reordering around them. Nevertheless, they are quite sufficient to establish synchronization in many RMC algorithms.[6]

[6] In x86-TSO ([Sew+10]), release and acquire accesses are the default and weakest atomic accesses.

RELEASE-ACQUIRE FENCES. We can also achieve release-acquire synchronization using relaxed accesses with *fences*, as in Example 2.1(d). Here, thread π performs a release fence ($\texttt{fence}_{\texttt{rel}}$) after the write to ℓ_x, and then relaxedly (**rlx**) writes to ℓ_y. Meanwhile, ρ performs an acquire fence ($\texttt{fence}_{\texttt{acq}}$) once it relaxedly reads 1 from ℓ_y. Note that there is no happens-before relation between the relaxed accesses of ℓ_y, but C11 guarantees happens-before between the accesses of ℓ_x (lines $\pi 1$ and $\rho 3$) through chains of the form "release fence $\rightarrow$ relaxed write $\rightarrow$ relaxed read $\rightarrow$ acquire fence". That is, synchronization is guaranteed between the events *before* the release fence and the events *after* the acquire fence *if* the two fences are connected by the relaxed write and read.

In terms of reordering, a release fence prevents reorderings of other accesses before it (to after it) and relaxed writes after it (to before it), while an acquire fence prevents reorderings of other accesses after it (to before it) and relaxed reads before it (to after it). Combining those restrictions with the fact that $\rho 1$ reads from $\pi 3$, we have that $\pi 1$ happens before $\rho 3$.

DATA RACES. Note that in Example 2.1(a), where we do not have sufficient synchronization between the accesses to ℓ_x, the worst thing can happen is that ρ would read unwanted values. However, if we were to replace the **rlx** accesses of ℓ_x with non-atomic accesses (**na**), it would constitute a data race and the program would exhibit undefined behavior. In the remaining examples in Figure 2.1, we always have sufficient synchronization (and hence no races) between the accesses of ℓ_x, so we can use non-atomic accesses for those.

2.2 RC11, Formally

The axiomatic semantics of C11/RC11 relaxed memory models are defined in two steps:

- first, we generate a set of *candidate executions* for the program of interest, in form of *graphs* whose vertices are memory *events* generated by the program's memory accesses and whose edges are several *partial orders* among the events;
- then, the behaviors of the program are those candidate executions that satisfy the model's *consistency axioms*.

In the following, we provide an excerpt of the RC11 formalization—following Lahav et al.[7] closely with minor presentation deviations—that are relevant to the features used in this dissertation. Interested readers can consult the original paper. Note that the formalizations in this chapter are also not included in our Coq developments. Again, they can be found in the RC11 paper's artifacts.

[7] Lahav et al., "Repairing sequential consistency in C/C++11" [Lah+17].

2.2.1 *Basic Definitions*

A relaxed memory model only concerns about the possible orders between memory accesses, and thus can be separated from the language syntax. Therefore, we can delay our language syntax much later (Chapter 4).For the bare minimum, we assume the abstract types *Loc* for memory locations and *Val* for values stored in memory, with meta-variables $\ell \in \mathit{Loc}$ and $v \in \mathit{Val}$, respectively.

First, we need the type of memory access consistency mode:

Definition 2.2 (Memory Access Consistency Mode)**.**

$$o \in \mathit{AccessMode} ::= \mathbf{sc} \mid \mathbf{acq} \mid \mathbf{rel} \mid \mathbf{relacq} \mid \mathbf{rlx} \mid \mathbf{na}.$$

AccessMode's Lattice $\boxed{o_1 \sqsubseteq o_2}$

$$\mathbf{na} \sqsubseteq o \qquad \mathbf{rlx} \sqsubseteq \mathbf{acq} \qquad \mathbf{rlx} \sqsubseteq \mathbf{rel}$$

$$\mathbf{rel} \sqsubseteq \mathbf{relacq} \qquad \mathbf{acq} \sqsubseteq \mathbf{relacq} \qquad o \sqsubseteq \mathbf{sc}$$

Definition 2.3 (Memory Access Event)**.** Each memory access generates an event of type *MemEvent*, with the meta-variable ε.

$$\varepsilon \in \mathit{MemEvent} ::= \mathtt{R}^o(\ell, v) \mid \mathtt{W}^o(\ell, v) \mid \mathtt{U}^{o_r, o_w}(\ell, v_r, v_w) \mid \mathtt{F}^o.$$

Specifically:

- $\mathtt{R}^o(\ell, v)$: a Read of v from ℓ, with access mode $o \in \{\mathbf{na}, \mathbf{rlx}, \mathbf{acq}, \mathbf{sc}\}$.
- $\mathtt{W}^o(\ell, v)$: a Write of v to ℓ, with access mode $o \in \{\mathbf{na}, \mathbf{rlx}, \mathbf{rel}, \mathbf{sc}\}$.
- $\mathtt{U}^{o_r, o_w}(\ell, v_r, v_w)$: a read-modify-write (Update) to ℓ, with read value v_r and write value v_w, and read access mode $o_r \in \{\mathbf{rlx}, \mathbf{acq}, \mathbf{sc}\}$, and write access mode $o_w \in \{\mathbf{rlx}, \mathbf{rel}, \mathbf{sc}\}$.[8]
- $\mathtt{F}^o$: a memory Fence, with $o \in \{\mathbf{acq}, \mathbf{rel}, \mathbf{relacq}, \mathbf{sc}\}$.

[8] Alternatively, RC11 models an Update event as a Read event immediately followed by a Write event. Here we follow C11. It is only a matter of presentation.

Definition 2.4 (Memory Event Projections)**.** For a memory event ε, the projections loc, mod, val_r, and val_w respectively give ε's location, access mode, read value and write value when applicable. More specifically, locis only applicable for R, W, and U events; val_r is applicable for R and U events; and val_w is applicable for W and U events.

For U events, mod is defined as follows:

- $\mathtt{U}^{\mathbf{rlx},\mathbf{rlx}}(\_).\mathsf{mod} ::= \mathbf{rlx}$
- $\mathtt{U}^{\mathbf{acq},\mathbf{rlx}}(\_).\mathsf{mod} ::= \mathbf{acq}$
- $\mathtt{U}^{\mathbf{rlx},\mathbf{rel}}(\_).\mathsf{mod} ::= \mathbf{rel}$
- $\mathtt{U}^{\mathbf{acq},\mathbf{rel}}(\_).\mathsf{mod} ::= \mathbf{relacq}$

- $\mathtt{U}^{\mathsf{sc},\mathsf{sc}}(\_).\mathsf{mod} ::= \mathbf{sc}$

Notation 2.5 (Update Event Access Mode)**.** Consequently, we also use the following shorthand notations for Update events:

- $\mathtt{U}^{\mathtt{rlx}}(\_) ::= \mathtt{U}^{\mathtt{rlx},\mathtt{rlx}}(\_)$
- $\mathtt{U}^{\mathtt{relacq}}(\_) ::= \mathtt{U}^{\mathtt{acq},\mathtt{rel}}(\_)$
- $\mathtt{U}^{\mathtt{rel}}(\_) ::= \mathtt{U}^{\mathtt{rlx},\mathtt{rel}}(\_)$
- $\mathtt{U}^{\mathtt{sc}}(\_) ::= \mathtt{U}^{\mathtt{sc},\mathtt{sc}}(\_)$
- $\mathtt{U}^{\mathtt{acq}}(\_) ::= \mathtt{U}^{\mathtt{rlx},\mathtt{acq}}(\_)$

Notation 2.6 (Memory Event Sets)**.** The notations R, W, U, and F respectively denote sets of Read, Write, Update, and Fence events.

We may also combine event sets, *e.g.*, $\mathtt{RW} ::= \mathtt{R} \cup \mathtt{W}$. We use subscript and superscript respectively to filter the sets by accessed location and access mode, *e.g.*, $\mathtt{W}_{\ell}^{\sqsupseteq \mathtt{rel}} ::= \{\varepsilon \in \mathtt{W} \mid \varepsilon.\mathsf{loc} = \ell \wedge \varepsilon.\mathsf{mod} \sqsupseteq \mathbf{rel}\}$.

Notation 2.7 (Memory Event Relations)**.** For a binary relation on events $R \in \mathit{MemEvent} \times \mathit{MemEvent}$, $R^?$, R^+, and R^* respectively denote its reflexive, transitive, and reflexive-transitive closures. $\mathtt{dom}(R)$ and $\mathtt{codom}(R)$ denote the domain and co-domain of R, respectively.

The notation $R_1 \,;\, R_2$ denotes the left composition of two relations R_1 and R_2. We assume that ; binds stronger than $\cup$ and $\setminus$. The notation $[A]$ stands for the identity relation on the set A. Consequently, $[A] \,;\, R$ can be understood as filtering R on the left with A, while $R \,;\, [B]$ filters R on the right with B. That is, $[A] \,;\, R = \{(\varepsilon_a, \varepsilon_b) \in R \mid \varepsilon_a \in A\}$, and $R \,;\, [B] = \{(\varepsilon_a, \varepsilon_b) \in R \mid \varepsilon_b \in B\}$. Finally, $[A] \,;\, R \,;\, [B] = R \cap (A \times B)$.

Given a function f, $=_f$ and $\neq_f$ denote the binary relations of pairs that are f-equal and f-non-equal, respectively:

$$=_f ::= \{(\varepsilon_a, \varepsilon_b) \mid f(\varepsilon_a) = f(\varepsilon_b)\}$$
$$\neq_f ::= \{(\varepsilon_a, \varepsilon_b) \mid f(\varepsilon_a) \neq f(\varepsilon_b)\}$$

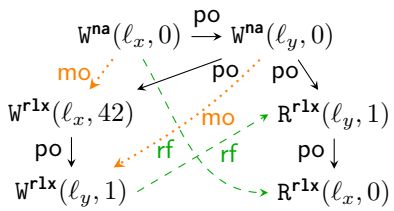
Meanwhile, given a relation R', $R|_{R'}$ denotes the filtering of R with respect to R', *i.e.*, $R|_{R'} ::= R \cap R'$. For example, $R|_{=_{\mathsf{loc}}}$ and $R|_{\neq_{\mathsf{loc}}}$ denote the relation R restricted to same and different locations, respectively.

2.2.2 *Execution Graphs*

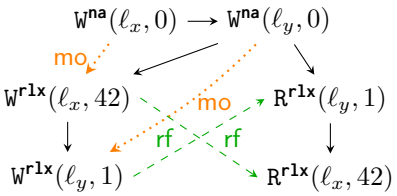
Definition 2.8 (Execution Graph)**.** An execution graph G is a tuple $(\mathtt{E}, \mathsf{po}, \mathsf{rf}, \mathsf{mo})$:

- E is the set of memory events (*MemEvent*) in G.
- The *program order* po is a *strict*[9] partial order that orders each thread's event by the program's control flow. For simplicity, RC11 assumes that for each location ℓ, E contains a Write event $\varepsilon_0^\ell ::= \mathtt{W}^{\mathtt{na}}(\ell, 0)$ as the initialization for ℓ. po is then required to order initialization events before all other events, *i.e.*, $\mathtt{E}_0 \times (\mathtt{E} \setminus \mathtt{E}_0) \subseteq \mathsf{po}$ where $\mathtt{E}_0 ::= \{\varepsilon_0^\ell \in \mathtt{E}\}$ is the set of E's initialization events.
- The *reads-from* relation rf relates a write with a read that reads from it, *i.e.*,

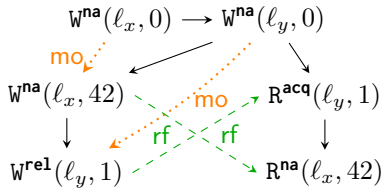
[9] It is irreflexive, *i.e.*, $(\varepsilon, \varepsilon) \notin \mathsf{po}$.



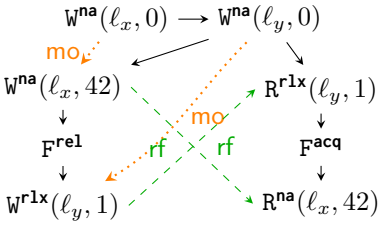
((a)) An execution of Example 2.1(a).



((b)) Another execution of Example 2.1(a).



((c)) An execution of Example 2.1(c).



((d)) An execution of Example 2.1(d).

FIGURE 2.2: Candidate executions of several MP examples.

(i) $\mathsf{rf} \subseteq [\mathtt{WU}] ; =_{\mathsf{loc}} ; [\mathtt{RU}]$;[10] and

(ii) rf respects written and read values: $\varepsilon_w.\mathsf{val}_w = \varepsilon_r.\mathsf{val}_r$ for all $(\varepsilon_w, \varepsilon_r) \in \mathsf{rf}$; and

(iii) rf is injective: if $(\varepsilon_w^1, \varepsilon_r) \in \mathsf{rf}$ and $(\varepsilon_w^2, \varepsilon_r) \in \mathsf{rf}$ then $\varepsilon_w^1 = \varepsilon_w^2$.

- The *modification order* mo is a strict partial order that gives a *strict total order* on the write events of each location. That is, mo is a disjoint union of the relations $\{\mathsf{mo}_\ell\}_{\ell \in Loc}$ where mo_ℓ is a strict total order on $(\mathtt{WU})_\ell$.

The components are also used as projections, *e.g.*, $G.\mathsf{mo}$. In cases where G is clear in the context, we may also drop the "$G.$" part and just use mo.

Definition 2.9 (Candidate Execution)**.** Execution graphs of a program $\mathcal{P}$ encode *prefixes* of traces of events generated by the program's memory accesses and fences. A execution G is a *candidate execution* if it represents a *full* trace generated by the whole program $\mathcal{P}$.

Example 2.10 (Candidate Executions for MP)**.** Figure 2.2 gives a few candidate executions for several MP examples in Figure 2.1. We use filled arrows, dotted arrows, and dashed arrows—with the same colors—for po, mo, and rf edges, respectively, between events. To avoid cluttering, we sometimes elide edge labels and instead use the arrow style to make the edge's type evident.

Definition 2.11 (Complete Execution)**.** An execution G is *complete* if every read reads some written value, *i.e.*, $G.\mathtt{R} \subseteq \mathtt{codom}(G.\mathsf{rf})$. A candidate execution is always complete, but the reverse is not always true.

[10] Recall that an Update event is considered both a read and a write.

Definition 2.12 (Derived Relations)**.** RC11 defines the following derived partial orders on execution graphs.

$$\mathsf{rb} ::= (\mathsf{rf}^{-1} ; \mathsf{mo}) \setminus [\mathtt{E}] \qquad \textit{(reads before)}$$

$$\mathsf{eco} ::= (\mathsf{rf} \cup \mathsf{mo} \cup \mathsf{rb})^{+} \qquad \textit{(extended coherence order)}$$

$$\mathsf{rs} ::= [\mathtt{WU}] ; \mathsf{po}|^{?}_{=\mathsf{loc}} ; [\mathtt{WU}^{\sqsupseteq \mathtt{rlx}}] ; (\mathsf{rf} ; [\mathtt{U}])^{*} \qquad \textit{(release sequence)}$$

$$\mathsf{sw} ::= [\mathtt{E}^{\sqsupseteq \mathtt{rel}}] ; ([\mathtt{F}] ; \mathsf{po})^{?} ; \mathsf{rs} ; \mathsf{rf} ; [\mathtt{RU}^{\sqsupseteq \mathtt{rlx}}] ; (\mathsf{po} ; [\mathtt{F}])^{?} ; [\mathtt{E}^{\sqsupseteq \mathtt{acq}}] \qquad \textit{(synchronized-with)}$$

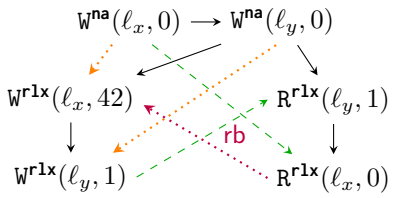
$$\mathsf{hb} ::= (\mathsf{po} \cup \mathsf{sw})^{+} \qquad \textit{(happens-before)}$$

$$\mathsf{psc} ::= \ldots \text{(elided)} \qquad \textit{(partial SC)}$$

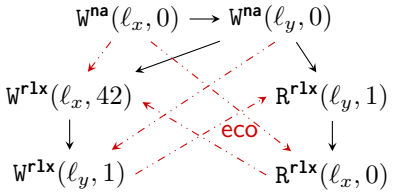
- The reads-before relation rb relates a Read event ε_r and a Write event ε_w, where ε_r reads from (rf) a write that is mo-before ε_w. The "$\setminus [\mathtt{E}]$" part is to exclude the case where an Update event reads from itself.
- The extended coherence order eco is the transitive closure of rf, mo, and rb, and is defined by RC11 to remedy C11's behaviors for SC accesses and fences.[11]
- The release sequence rs of a Write event ε_w contains (i) all later same-thread, same-location ($\mathsf{po}|_{=\mathsf{loc}}$-later) atomic writes ($\mathtt{WU}^{\sqsupseteq \mathtt{rlx}}$) including the write ε_w itself—hence the reflexive closure ($?$) of $\mathsf{po}|_{=\mathsf{loc}}$, as well as (ii) all Updates that recursively read from such writes.
- The synchronized-with relation sw defines *inter*-thread synchronization. A release event $\varepsilon_a \in \mathtt{E}^{\sqsupseteq \mathtt{rel}}$ is synchronized with an acquire event $\varepsilon_b \in \mathtt{E}^{\sqsupseteq \mathtt{acq}}$, if ε_b (or, in case ε_b is a Fence event, some atomic Read event that is po-before ε_b) reads from the release sequence of ε_a (or in case ε_a is a Fence event, some atomic Write event that is po-after ε_a). Note that the relation rs ; rf is between a Write event and a Read event. The relations $([\mathtt{F}] ; \mathsf{po})^{?}$ and $(\mathsf{po} ; [\mathtt{F}])^{?}$ allow us to extend rs ; rf to fences that come po-before and po-after the Write and the Read events in rs ; rf, respectively.
- Most importantly, the happens-before relation hb formally defines what *global* synchronization means, as the transitive closure of the *inter*-thread synchronization sw relation and the *intra*-thread program order po.
- Finally, the partial SC relation psc is defined by RC11 to rectify SC behaviors, using a diligent combination of mo, rf, rb, eco, and hb. The exact definition, however, is not in the focus of this dissertation and therefore elided.

[11] Lahav et al., "Repairing sequential consistency in C/C++11" [Lah+17].

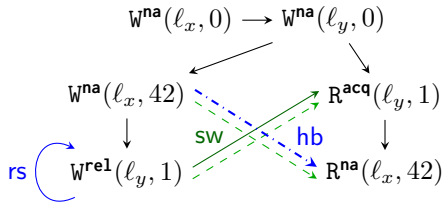
Example 2.13 (Illustrations of Derived Relations)**.** Figure 2.3 demonstrates the derived relations on several execution graphs. Figure 2.3(e) especially demonstrates a fairly complex instance of the release sequence rs relation with 4 threads, of which the middle 2 threads use Updates (atomic read-modify-write instructions).



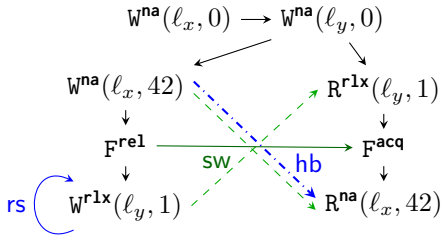
((a)) rb on the execution in Figure 2.2(a).



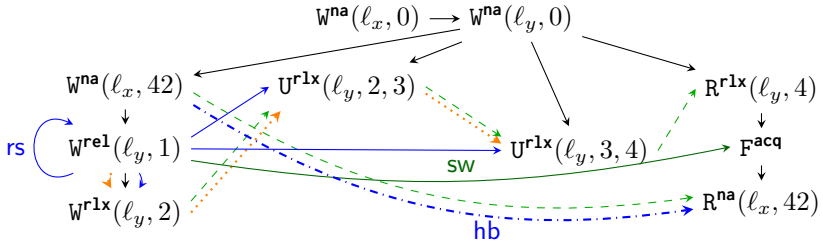
((b)) eco (transitive closure elided) derived from Figure 2.3(a).



((c)) rs, sw, and a part of hb on the execution in Figure 2.2(c).



((d)) rs, sw, and a part of hb on the execution in Figure 2.2(d).



((e)) More complex rs, sw, and hb on an execution with Updates.

FIGURE 2.3: Illustrations of derived relations.

We use dotted arrows, dash-dot-dotted arrows, filled arrows, filled arrows, and dash-dotted arrows, respectively for rb, eco, rs, sw, and hb edges.

2.2.3 *Consistency*

Definition 2.14 (RC11-consistency)**.** An execution G is RC11-*consistent* if it is complete (Definition 2.11) and

- hb ; $\text{eco}^?$ is irreflexive; and (RC11-COHERENCE)
- psc is acyclic; and (RC11-SC)
- po $\cup$ rf is acyclic. (RC11-NO-OOTA)

RC11-COHERENCE is the main axiom that give sane behaviors to most memory operations—see Proposition 2.19 below. RC11-SC is the main contribution of the RC11 work to give better semantics for SC accesses and fences, which, again, is not in the focus of this dissertation and is only stated here for completeness. The RC11-NO-OOTA condition is a simple fix to forbid *load-buffering* (LB) behaviors, and therefore forbids the *out-of-thin-air* problem[12]—see Remark 2.21 below.

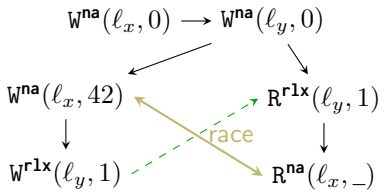
[12]Boehm and Demsky, "Outlawing ghosts: avoiding out-of-thin-air results" [BD14].

$$\ell_x :=_{\mathtt{na}} 0; \ell_y :=_{\mathtt{na}} 0;$$

$$\pi 1:\ \ell_x :=_{\mathtt{na}} 42; \quad \Big\| \quad \rho 1:\ \mathtt{if}\ {}^{*\mathtt{rlx}}\ell_y \mathrel{!}= 0\ \mathtt{then}$$

$$\pi 2:\ \ell_y :=_{\mathtt{rlx}} 1; \quad \Big\| \quad \rho 2:\ \quad {}^{*\mathtt{na}}\ell_x;\ \text{// racy}$$

((a)) A racy MP program.



((b)) No hb is established between the accesses of ℓ_x.

FIGURE 2.4: A racy execution of a racy MP program.

2.2.4 *Data Races*

Definition 2.15 (Races)**.** Two events ε_a and ε_b are *conflicting* in an execution G if they are on the same location and one of them is a write, *i.e.*, $\varepsilon_a \neq \varepsilon_b$ and $\varepsilon_a.\mathsf{loc} = \varepsilon_b.\mathsf{loc}$ and $\{\varepsilon_a, \varepsilon_b\} \cap G.(\mathtt{WU}) \neq \varnothing$.

The pair $(\varepsilon_a, \varepsilon_b)$ is called a *race* in G, denoted $(\varepsilon_a, \varepsilon_b) \in G.\mathsf{race}$, if they are conflicting in G and neither happens before the other, *i.e.*, $(\varepsilon_a, \varepsilon_b) \notin \mathsf{hb} \cup \mathsf{hb}^{-1}$.

Definition 2.16 (Racy Executions)**.** An execution G is called *racy* if there is some racy event pair in G such that one of them is a non-atomic access, *i.e.*, $\exists (\varepsilon_a, \varepsilon_b) \in G.\,\mathsf{race} \wedge \{\varepsilon_a, \varepsilon_b\} \cap \mathtt{E}^{\mathtt{na}} \neq \varnothing$.

Example 2.17 (Racy Execution of MP)**.** A racy MP program and one of its racy executions is given in Figure 2.4. The race is between the non-atomic accesses of ℓ_x, where no hb edge is established between the accesses, because we use only relaxed accesses for ℓ_y.

2.2.5 *Program Behaviors*

Definition 2.18 (RC11 Program Behavior)**.** A program $\mathcal{P}$ has *undefined behavior* (UB) under RC11 if it has some racy RC11-consistent execution. Otherwise, its behaviors are defined by the set of RC11-consistent *full* executions of $\mathcal{P}$.

Proposition 2.19 (RC11 and C11 Coherence)**.** RC11-COHERENCE *is equivalent to the conjunction of the following* C11 *axioms:*[13]

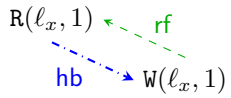
- hb *is irreflexive.* (C11-HB)
- rf ; hb *is irreflexive.* (C11-NO-FUTURE-READ)
- mo ; rf ; hb *is irreflexive.* (C11-CoRW)
- mo ; hb *is irreflexive.* (C11-CoWW)
- mo ; hb ; rf^{-1} *is irreflexive.* (C11-CoWR)
- mo ; rf ; hb ; rf^{-1} *is irreflexive.* (C11-CoRR)

[13] Lahav et al., "Repairing sequential consistency in C/C++11" [Lah+17], §3.4, Proposition 1.

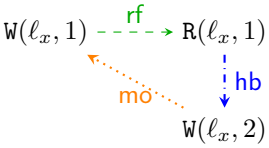
Example 2.20 (C11 Coherence)**.** C11 coherence axioms are demonstrated by several forbidden (non-consistent) behaviors[14] in Figure 2.5.

- C11-HB ensures that hb is a strict partial order.
- C11-NO-FUTURE-READ (Figure 2.5(a)) says that a read may not happen before the write that it reads from.

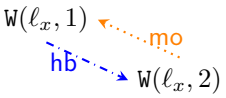
[14] Batty et al., "Mathematizing C++ concurrency" [Bat+11], §2.7.



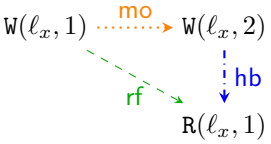
((a)) Violation of C11-NO-FUTURE-READ.



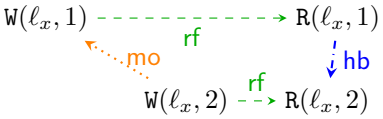
((b)) Violation of C11-CoRW.



((c)) Violation of C11-CoWW.



((d)) Violation of C11-CoWR.



((e)) Violation of C11-CoRR.

FIGURE 2.5: Several *forbidden* (inconsistent) executions in C11/RC11.

- C11-CoRW (Figure 2.5(b)) requires that a read may not happen before a write that mo-before the write it reads from.
- C11-CoWW (Figure 2.5(c)) requires that mo and hb may not disagree.
- C11-CoWR (Figure 2.5(d)) requires that a read may not read from a write that is already hidden by (mo-before) another write that happens-before it.
- C11-CoRR (Figure 2.5(e)) requires that two reads connected by hb may not read from writes with the inverse order in mo.

Remark 2.21 (LB and OOTA)**.** The C11 memory model allows the so-called Load-Buffering (LB) behavior, while the RC11 memory model simply forbids it with RC11-NO-OOTA. Figure 2.6(a) gives an example program with an execution demonstrating its LB behavior in Figure 2.6(c). Here, one can think that the reads (loads) are buffered until the writes are completed, and then they can both read 1. The execution in Figure 2.6(c) is consistent in C11, and so such behavior is allowed in C11.

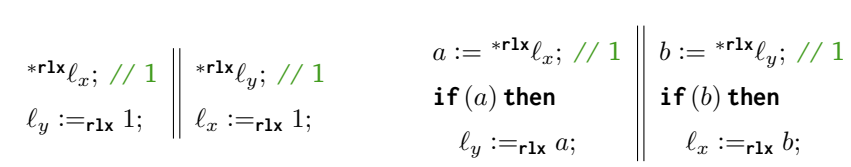
The problem with LB is that, the same execution in Figure 2.6(c) justifies an *undesirable* behavior of the program in Figure 2.6(b), where the reads read 1, even though 1 does not appear in the program: it appears out of thin air (OOTA)! OOTA behaviors are forbidden by the *informal* C11 standard, and are not exhibited in any implementation. However, it is formally non-trivial to distinguish LB, which is desirable, from OOTA, which is not. Several solutions are already proposed to distinguish them,[15] but they result in more involved semantics. Furthermore, the LB behavior itself is rather non-local and makes it hard to build high-level, logic-based reasoning—see [Sve+18] for an attempt.

RC11 resolves to a simpler solution: forbidding LB behaviors altogether, by requiring $\mathsf{po} \cup \mathsf{rf}$ to be acyclic (RC11-NO-OOTA). Similar to existing logics,[16] we also adopt this solution in ORC11 (which is an operational version of RC11), as it simplifies the construction of our separation logic. Recent work by Ou and Demsky[17] suggests that the performance

[15] Kang et al., "A promising semantics for relaxed-memory concurrency" [Kan+17]; Chakraborty and Vafeiadis, "Grounding thin-air reads with event structures" [CV19].

[16] [VN13; DV16; DV17].

[17] Ou and Demsky, "Towards understanding the costs of avoiding out-of-thin-air results" [OD18].



((a)) A program with LB behaviors.

((b)) A program with OOTA behaviors in C11.

((c)) An execution with LB behaviors.

FIGURE 2.6: Load-buffering (LB) and Out-of-thin-air (OOTA) behaviors.

overhead of working with RC11 vs. C11 may not be so significant in practice. Even more recently, Lee et al.[18] suggests forbidding LB behaviors in the source language model (*e.g.*, C and C++) but still allowing LB in the intermediate representation (IR) semantics. This would result in a compromising memory model where logics like iRC11 can be used for verifications the source level, and where all optimizations for non-atomics can still be performed at the IR level, and therefore imposing no overhead on non-atomics.

[18] Lee et al., "Putting Weak Memory in Order via a Promising Intermediate Representation" [Lee+23].

Remark 2.22 (Consume Accesses and Locks)**.** Unlike C11, RC11 does not consider *consume* accesses, which is a premature feature not implemented by major compilers, nor *locks*, which can be implemented with release-acquire accesses.

CHAPTER SUMMARY. This chapter reviews the high-level intuition for the behaviors of C11 atomic accesses, as well as an excerpt of the RC11 formalization in form of axiomatic semantics. The distinctive feature of the RMC axiomatic semantics is the use of axioms to constrains partial orders among memory events. While this style results in very concise definitions,[19] it may take time to get used to. Nevertheless, the main inconvenience of axiomatic semantics is that the behaviors are encoded in the axioms that are stated rather *globally* on relations that span multiple events across multiple threads, making it difficult to prove soundness of *thread-local*, Hoare-style CSLs directly on top of those semantics.[20] In the next chapter, we present ORC11, an operational version of RC11 that is more convenient to build our separation logic iRC11 in Iris.

[19] In contrast, the formulation of ORC11 in Chapter 3 is much more verbose.

[20] Yet it is still achievable, by annotating resources on incoming and outcoming edges of an event node, as Vafeiadis et al. demonstrated with RSL and FSL ([VN13; DV16]).

3

ORC11: *Operational Repaired C11*

Following iGPS,[1] we need an operational semantics for relaxed memory so that it can be instantiated in the Iris framework. We extend iGPS's operational semantics for release-acquire and non-atomics (RA+NA) to include relaxed accesses and fences. The result is ORC11—Operational Repaired C11.[2]

Features-wise, ORC11 is closely related to the axiomatic semantics of RC11.[3] Most importantly, it forbids load-buffering (LB) behaviors, *i.e.*, $\mathsf{po} \cup \mathsf{rf}$ is acyclic. Construction-wise, ORC11 follows the *view*-based approach to operational semantics for relaxed memory.[4] More concretely, it follows the promising semantics formalization[5] but *without* promises, and thus forbids LB. The promising semantics, however, does not model non-atomics.[6] Meanwhile, ORC11 needs to employ a race detector to formalize races on non-atomics which, as we will see, in the presence of relaxed accesses, are more tricky to get right than iGPS's race detector.

Consequently, ORC11 is defined by two sub-semantics: the view-based *machine* semantics that focuses on relaxed behaviors (§3.3) and the *race-detector* semantics that focuses on UB-triggering races (§3.4). In §3.6, we sketch a paper proof of correspondence between ORC11 and RC11.

The *expression* semantics, which defines the reductions of language expressions, can fortunately be mostly separated from the relaxed memory model that is ORC11. Chapter 4 will present the relaxed λ_{Rust} language which combines the expression reductions together with ORC11.[7]

But first, let us give a high-level, intuitive explanation of RMC using views.

[1] Kaiser et al., "Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris" [Kai+17].

[2] The definitions in this chapter are formalized in Coq in the repository https://gitlab.mpi-sws.org/iris/gpfsl/-/tree/master/orc11.

[3] Lahav et al., "Repairing sequential consistency in C/C++11" [Lah+17].

[4] Steinke and Nutt, "A unified theory of shared memory consistency" [SN04]; Lahav et al., "Taming release-acquire consistency" [LGV16]; Podkopaev et al., "Operational Aspects of C/C++ Concurrency" [PSN16].

[5] Kang et al., "A promising semantics for relaxed-memory concurrency" [Kan+17].

[6] It models *plain* accesses, who can race without triggering undefined behaviors.

[7] And thus the semantics of the relaxed λ_{Rust} language is a combination of three sub-semantics: the expression semantics, the view-based machine semantics, and the race-detector semantics.

3.1 Understanding Relaxed Memory with Views

The view-based approach to operational semantics for relaxed memory allows for a more *thread-local* characterization of relaxed effects. In particular, each thread in the program has its own *local view* which represents its subjective *observations* on the *globally-shared* memory.

For example, a thread π's local view may record (but not limited to) the writes to memory that the thread has observed, *e.g.*, those writes that *happen before* the current program counter $\mathtt{PC}_\pi$ of the thread π. More concretely, if we follow the language of iGPS[8] and track writes to memory in views, we can defined a view as a map from memory locations

[8] Kaiser et al., "Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris" [Kai+17].

to timestamps: *View* ::= *Loc* $\xrightarrow{\text{fin}}$ *Time*, where the timestamps are indices into an ordering of the writes to a location.[9] With $V(\ell) = t$, we say that the view V has *seen* or observed the write to ℓ identified by the timestamp t. When thread π writes to a location ℓ in shared memory, a new write event ε_w is added to the shared memory with some fresh timestamp t_w, and thread π updates its local view (its observations) V_π accordingly to include t_w for ℓ.

[9] Timestamps are typically just natural numbers, but can be more complex depending on the memory model.

However, it is not necessary that another thread ρ observes that write ε_w by thread π immediately. In the terminology of views, we say that thread ρ's local view V_ρ does not include the timestamp t_w for ℓ. In order to observe the write ε_w, thread ρ needs to perform physical synchronization with thread π, so that thread π's local view V_π (which includes ε_w) is incorporated or *joined* into V_ρ. Then, V_π is *included* in V_ρ: $V_\pi \sqsubseteq V_\rho$. The *view inclusion* relation therefore approximates synchronization, or more formally, the happens-before (hb) relation. Consequently, as threads execute and their local views grow over time, they occasionally synchronize with one another by *sending* their local views to other threads.

Example 3.1 (Racy MP with Views)**.** Consider again the racy MP example in Figure 2.4 (Example 2.17), where we use relaxed accesses for ℓ_y and therefore we are not guaranteed a happens-before relation between the conflicting non-atomic accesses to ℓ_x by thread π and thread ρ. In the language of views, this racy behavior can be explained as follows, using Figure 3.1(a).

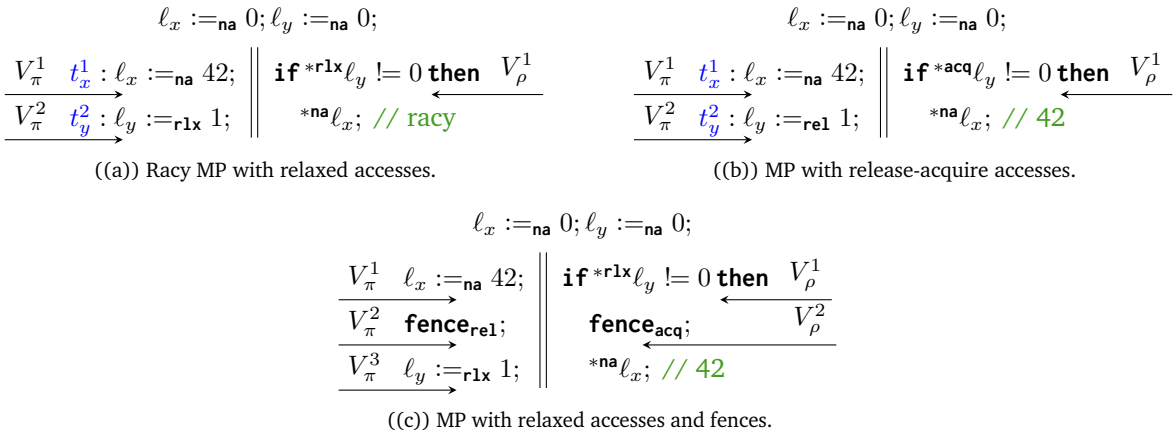
- After thread π writes 42 to ℓ_x, its local view is V_π^1, as illustrated in Figure 3.1(a) with an arrow pointing to after the write. As an approximation of hb, V_π^1 also tracks the po relation in π, and thus it includes the *freshly created* timestamp t_x^1 for the write 42 to ℓ_x, *i.e.*, $V_\pi^1(\ell_x) = t_x^1$.

- Similarly, after thread π writes 1 to ℓ_y, its local view V_π^2 includes the timestamp t_y: $V_\pi^2(\ell_y) = t_y^2$. More importantly, $V_\pi^1 \sqsubseteq V_\pi^2$: a thread-local view only grows, so as to maintain that happens-before (hb) contains program order (po).

- Unfortunately, because we use a relaxed access in writing 1 to ℓ_y, thread π does *not* release its local view (neither V_π^1 nor V_π^2) with that write.

- So even if thread ρ reads that write of 1 to ℓ_y, it does *not* acquire the timestamp t_x^1 (for the write of 42 to ℓ_x) into its local view after the read V_ρ^1.

- Consequently, when reading ℓ_x non-atomically in the next line, thread ρ's current local view V_ρ^1 is not guaranteed to include t_x^1. In the operational semantics, performing a non-atomic operation without having observed *all* writes to the same location in the local view constitutes a race.



((a)) Racy MP with relaxed accesses.

((b)) MP with release-acquire accesses.

((c)) MP with relaxed accesses and fences.

FIGURE 3.1: View-based explanation of MP behaviors.

Put it differently, that thread ρ's view before the non-atomic read of ℓ_x does not include t_x^1 *approximates* the fact that the non-atomic write to ℓ_x does not happen before the non-atomic read, hence a race.

Example 3.2 (Release-Acquire MP with Views)**.** Similarly, the release-acquire synchronization in Figure 2.1(c) (Example 2.1) can also be explained with views, using Figure 3.1(b). Because we instead use a release write of 1 to ℓ_y, thread π *releases* its local view V_π^2 through the write (which also include the write itself). When thread ρ reads that write using an acquire read, it *acquires* that view into its local view V_ρ^1. Accordingly, $V_\pi^2 \sqsubseteq V_\rho^1$, so thread ρ has observed all writes to ℓ_x, and can safely read ℓ_x non-atomically. Furthermore, thread ρ reads from ℓ_x's latest write, which is 42.

In other words, $V_\pi^2 \sqsubseteq V_\rho^1$ encodes the synchronized-with (sw) relation between the release-acquire pair, and transitively the happens-before relation (hb). Effectively, thread π's write of 42 to ℓ_x happens before thread ρ's read from ℓ_x, the race is excluded, and the expected behavior results.

Example 3.3 (Fence-MP with Views)**.** The view-based explanation for the MP example with fences in Figure 2.1(d) (Example 2.1) is a bit more interesting, using Figure 3.1(c).

- Here, thread π's relaxed write to ℓ_y, like in Example 3.1, does not release the *current* local view V_π^3 at the point of the write. However, unlike in Example 3.1, π's release fence which comes before guarantees that the write to ℓ_y does release π's local view *before* the fence, *i.e.*, V_π^1.
- On the other side, ρ's read of ℓ_y, if reads 1, will acquire V_π^1, but does not immediately *join* V_π^1 into its local view V_ρ^1 right after the read, *i.e.*, $V_\pi^1 \not\sqsubseteq V_\rho^1$. Instead, later, thread ρ's acquire fence will perform that join, so that after the acquire fence, $V_\pi^1 \sqsubseteq V_\rho^2$. Consequently, we again have the hb relation between the non-atomic write and read of ℓ_x.

In other words, to explain fences behaviors in terms of views, we requires more views than just the *current* thread-local view: a release fence *stores*

the current view (by the time of the fence), so that it can be released through some *later* relaxed write, while an acquire fence *restores* (into the current view) some view that have been acquired by some *earlier* relaxed read. This is in agreement with the definition of the synchronized-with (sw) relation (Definition 2.12).

SUMMARY. Views are an approximation of the happens-before (hb) relation that is more thread-local and can help simplify the soundness proof of RMC separation logics. However, we need more intricate uses of views to handle fences (§3.3), which need multiple views, and to handle data races (§3.4), which, due to their subtle interactions with relaxed (**`rlx`**) accesses, require views to have a more complex structure than just a map from locations to timestamps.

3.2 Basic Machine State Definitions

We define the basic definitions of ORC11's machine state, whose most important components are the *globally-shared* memory and the *thread-local* views. First, we note some extra features that affect the formal definitions of ORC11.

Pointer Arithmetic The λ_{Rust} language (Chapter 4) adopts the CompCert model for locations,[10] where allocations and deallocations are done in *blocks*, and a location consists of a block index i and an offset n into that block, and pointer arithmetics can only be performed within the same block. Consequently, we need to model explicit allocations and deallocations of blocks.

[10] Leroy et al., *The CompCert Memory Model, Version 2* [Ler+12].

Uninitialized Memory λ_{Rust} also allows memory to be uninitialized, with the only safe operations are reading and writing to uninitialized memory—other uses of values read from uninitialized memory are undefined behavior. We follow λ_{Rust}, which in turn follows Lee et al.[11] to use a *poison value* ☣ for uninitialized memory.

[11] Lee et al., "Taming undefined behavior in LLVM" [Lee+17].

Data Races To handle the interactions between races and **`rlx`** accesses, ORC11 views cannot just simply track write events (like in iGPS and what we have seen in §3.1). Instead, ORC11 views need to track both read and write events.

Definition 3.4 (ORC11 Basic Types)**.**

$$
\begin{aligned}
\pi, \rho \in \mathit{Thread} &::= \mathbb{N}^+ \\
\ell \in \mathit{Loc} &::= (i, n) \qquad i \in \mathbb{N}^+, n \in \mathbb{Z} \\
v \in \mathit{Val} &::= \text{☣} \mid \dots \\
\omega \in \mathit{MemVal} &::= \dagger \mid \text{☣}^\dagger \mid v \in \mathit{Val} \\
t \in \mathit{Time} &::= \mathbb{N}^+ \\
\alpha \in \mathit{ActIds} &::= 2^{\mathbb{N}^+}
\end{aligned}
$$

- A thread-id π or ρ is a positive number.

- A location ℓ is a pair of block index i (which is a positive number) and an offset n.
- The value type *Val* can still be abstract, but should include the poison value ☣.
- The *memory value* type *MemVal* is the type for values stored in locations the global memory, which can be in *Val* or be the two additional values † and ✞ to respectively mark the *allocated* and *deallocated* states of a location.
- A timestamp t is a positive number.
- A set of *actions* α is a set of positive numbers, which will be used to track sets of reads and writes.

Definition 3.5 (ORC11 Memory Access Event)**.**

$$\begin{aligned}\varepsilon \in \mathit{MemEvent} ::= \;& \mid \mathtt{R}^o(\ell, v) \mid \mathtt{W}^o(\ell, v) \mid \mathtt{U}^{o_r,o_w}(\ell, v_r, v_w) \mid \mathtt{F}^o \\ & \mid \mathtt{A}(\ell, n \in \mathbb{N}^+) \mid \mathtt{D}(\ell, n \in \mathbb{N}^+).\end{aligned}$$

We extend memory events (Definition 2.3) to include two new event types: the *allocation* event type `A` and *deallocation* event type `D`. Both event types carry the *base* location ℓ of a block, and the size n of that block.

Definition 3.6 (Views)**.**

$$V \in \mathit{View} ::= \mathit{Loc} \xrightarrow{\text{fin}} \{\mathsf{w} : \mathit{Time}, \mathsf{aw} : \mathit{ActIds}, \mathsf{nr} : \mathit{ActIds}, \mathsf{ar} : \mathit{ActIds}\}$$

A (simple) view is a finite, partial map from locations to tuples of one timestamp and three sets of actions. For a view V and a location ℓ,

- $V(\ell).\mathsf{w}$ is the timestamp of the latest write to ℓ that V has seen.
- $V(\ell).\mathsf{aw}$ is the set of *atomic* writes to ℓ that V has seen.
- $V(\ell).\mathsf{nr}$ is the set of *non-atomic* reads from ℓ that V has seen.
- $V(\ell).\mathsf{ar}$ is the set of *atomic* reads from ℓ that V has seen.

Definition 3.7 (Views' Join Semi-Lattice)**.** The bottom element of views is the empty map $\varnothing$. The inclusion relation and the join operation for views are defined as follows.

$$\begin{aligned}V_1 \sqsubseteq V_2 ::= \;& \forall \ell.\, V_1(\ell).\mathsf{w} \le V_2(\ell).\mathsf{w} \land V_1(\ell).\mathsf{aw} \subseteq V_2(\ell).\mathsf{aw} \\ & \land V_1(\ell).\mathsf{nr} \subseteq V_2(\ell).\mathsf{nr} \land V_1(\ell).\mathsf{ar} \subseteq V_2(\ell).\mathsf{ar} \\ V_1 \sqcup V_2 ::= \;& \lambda \ell.\, \big\{\, \mathsf{w} := \mathtt{max}(\{V_1(\ell).\mathsf{w}, V_2(\ell).\mathsf{w}\}); \\ & \quad \mathsf{aw} := V_1(\ell).\mathsf{aw} \cup V_2(\ell).\mathsf{aw}; \\ & \quad \mathsf{nr} := V_1(\ell).\mathsf{nr} \cup V_2(\ell).\mathsf{nr}; \\ & \quad \mathsf{ar} := V_1(\ell).\mathsf{ar} \cup V_2(\ell).\mathsf{ar} \,\big\}\end{aligned}$$

Note that $V_1 \sqcup V_2$ is only defined for locations that are in the domains of either V_1 or V_2, *i.e.*, $\mathtt{dom}(V_1 \sqcup V_2) = \mathtt{dom}(V_1) \cup \mathtt{dom}(V_2)$. If some location

ℓ is not in one view, then the value in the other view takes over for the join.

For view inclusion, we consider by default:

$$V_1(\ell) \sqsubseteq V_2(\ell) \text{ if } \ell \notin \mathtt{dom}(V_1)$$
$$V_1(\ell) \not\sqsubseteq V_2(\ell) \text{ if } \ell \in \mathtt{dom}(V_1) \wedge \ell \notin \mathtt{dom}(V_2)$$

Definition 3.8 (Thread-Views)**.**

$$\mathcal{V} \in \mathit{ThreadView} ::= \{\mathsf{rel} : \mathit{Loc} \xrightarrow{\text{fin}} \mathit{View}, \mathsf{frel} : \mathit{View}, \mathsf{cur} : \mathit{View}, \mathsf{acq} : \mathit{View}\}$$

[12]This is inspired by thread-views of the promising semantics ([Kan+17]).

A thread-view[12] is used to track the observations of a thread, and has four components. For a thread π to have the thread-view $\mathcal{V}$ at its current program counter $\mathtt{PC}_\pi$,

- $\mathcal{V}.\mathsf{cur}$ is the actual, *current* view of π, which includes all reads and writes that happen before the current counter $\mathtt{PC}_\pi$.
- $\mathcal{V}.\mathsf{acq}$ is the *acquire* view of π. It tracks the observations acquired by π's earlier relaxed reads, and will be restored into π's current view *after* the next acquire or SC fence. In other words, it tracks all reads and writes that happen before π's next acquire or SC fence.
- $\mathcal{V}.\mathsf{frel}$ is the *release-fence* view of π. It tracks all reads and writes that happen before π's most recent release or SC fence, and it can be released by π's later relaxed writes.
- $\mathcal{V}.\mathsf{rel}$ is a finite, partial function that tracks *per-location release* views for π. For a location ℓ, $\mathcal{V}.\mathsf{rel}(\ell)$ is the release view of π's most recent release write to ℓ, and can be released by π's later relaxed writes to the same location ℓ. This view is needed to model the release sequence (rs, Definition 2.12) for ℓ.

Property 3.9 (Thread-Views Wellformedness)**.** The following properties must hold for a thread-view $\mathcal{V}$:

- $\mathtt{dom}(\mathcal{V}.\mathsf{rel}) \subseteq \mathtt{dom}(\mathcal{V}.\mathsf{cur})$ (TVIEW-DOM)
- $\forall \ell.\, \mathcal{V}.\mathsf{rel}(\ell) \sqsubseteq \mathcal{V}.\mathsf{cur}$ (TVIEW-REL)
- $\mathcal{V}.\mathsf{frel} \sqsubseteq \mathcal{V}.\mathsf{cur}$ (TVIEW-FREL)
- $\mathcal{V}.\mathsf{cur} \sqsubseteq \mathcal{V}.\mathsf{acq}$ (TVIEW-CUR)

Definition 3.10 (Thread-Views' Join Semi-Lattice)**.** The bottom element of thread-views, also denoted by $\varnothing$, is the tuple of an empty release map and empty (bottom) views. The inclusion relation and the join operation for thread-views are defined as follows.

$$\begin{aligned}
\mathcal{V}_1 \sqsubseteq \mathcal{V}_2 ::= {} & (\forall \ell.\, \mathcal{V}_1.\mathsf{rel}(\ell) \sqsubseteq \mathcal{V}_2.\mathsf{rel}(\ell)) \wedge \mathcal{V}_1.\mathsf{frel} \sqsubseteq \mathcal{V}_2.\mathsf{frel} \\
& \wedge \mathcal{V}_1.\mathsf{cur} \sqsubseteq \mathcal{V}_2.\mathsf{cur} \wedge \mathcal{V}_1.\mathsf{acq} \sqsubseteq \mathcal{V}_2.\mathsf{acq} \\
\mathcal{V}_1 \sqcup \mathcal{V}_2 ::= {} & \{\, \mathsf{rel} := \lambda \ell.\, \mathcal{V}_1.\mathsf{rel}(\ell) \sqcup \mathcal{V}_2.\mathsf{rel}(\ell); \\
& \mathsf{frel} := \mathcal{V}_1.\mathsf{frel} \sqcup \mathcal{V}_2.\mathsf{frel}; \\
& \mathsf{cur} := \mathcal{V}_1.\mathsf{cur} \sqcup \mathcal{V}_2.\mathsf{cur}; \\
& \mathsf{acq} := \mathcal{V}_1.\mathsf{acq} \sqcup \mathcal{V}_2.\mathsf{acq} \,\}
\end{aligned}$$

Definition 3.11 (Global Memory)**.**

$$\mathcal{M} \in \mathit{MsgPool} ::= \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Time} \xrightarrow{\text{fin}} \left\{ \mathsf{val} : \mathit{MemVal}, \mathsf{view} : \mathit{View}^? \right\}$$
$$m \in \mathit{ExtMsg} ::= \left\{ \mathsf{ts} : \mathit{Time}, \mathsf{val} : \mathit{MemVal}, \mathsf{view} : \mathit{View}^? \right\}$$

The global memory, or the *message pool* $\mathcal{M}$ contains all write messages to all locations. It is a finite, partial map from locations to timestamps to a pair of a written value and an *optional* view ($\mathit{View}^?$).

For a location, $\mathcal{M}(\ell)$ contains all write messages to ℓ, ordered by timestamps. The timestamp order of $\mathcal{M}(\ell)$ encodes the per-location modification order mo_ℓ (Definition 2.8) for ℓ.

For some timestamp t, the pair $\mathcal{M}(\ell)(t)$ carries the information about a write to ℓ identified by the timestamp t. If the write is an non-atomic write, then $\mathcal{M}(\ell)(t).\mathsf{view} = \mathsf{None}$. Otherwise, if the write is an atomic write, then $\mathcal{M}(\ell)(t).\mathsf{view} = \mathsf{Some}(V)$ for some view V that is called the (released) view of the write. As a shorthand notation for the option type, we write $\bot$ for None, and simply write V for $\mathsf{Some}(V)$.[13]

[13] Note that instead of using the option type, we can also require that the view V of a non-atomic write to be empty ($\varnothing$).

The message type *ExtMsg* combines the timestamp with the value and the optional view into a single message. As such, the message pool can be seen as a map from locations to messages: $\mathit{MsgPool} \approx \mathit{Loc} \xrightarrow{\text{fin}} \mathit{ExtMsg}$.

Property 3.12 (View Closedness)**.** $\boxed{V \in \mathcal{M}}$ $\boxed{\mathcal{V} \in \mathcal{M}}$

A view V is said to be closed in $\mathcal{M}$ if V only contains write messages in $\mathcal{M}$, *i.e.*, $\forall \ell, V(\ell).\,\mathsf{w} \in \mathcal{M}(\ell)$. The definition is lifted point-wise for thread-views.

Property 3.13 (Global Memory Wellformedness)**.** A global memory $\mathcal{M}$ is wellformed if the following hold.

$$\forall \ell, m.\, m \in \mathcal{M}(\ell) \land m.\mathsf{view} \neq \bot \Rightarrow m.\mathsf{view}(\ell') \in \mathcal{M} \quad \text{(WF-MEM-CLOSED)}$$
$$\forall \ell, m.\, m \in \mathcal{M}(\ell) \land m.\mathsf{view} \neq \bot \Rightarrow m.\mathsf{view}(\ell).\mathsf{w} = m.\mathsf{ts} \quad \text{(WF-MSG-VIEW)}$$
$$\forall \ell, t.\, \mathcal{M}(\ell)(t).\mathsf{val} = \dagger \Rightarrow t = \mathtt{min}(\mathtt{dom}(\mathcal{M}(\ell))) \quad \text{(WF-MEM-ALLOC)}$$
$$\forall \ell, t.\, \mathcal{M}(\ell)(t).\mathsf{val} = \Dagger \Rightarrow t = \mathtt{max}(\mathtt{dom}(\mathcal{M}(\ell))) \quad \text{(WF-MEM-DEALLOC)}$$

WF-MEM-CLOSED requires that $\mathcal{M}$ is closed in itself, *i.e.*, any view of any write messages in $\mathcal{M}$ only refers to messages also in $\mathcal{M}$. WF-MSG-VIEW requires that the view of a write message contains exactly the timestamp of that message. WF-MEM-ALLOC (resp. WF-MEM-DEALLOC) require that if a write is an allocation (resp. deallocation) then it must be the minimum (resp. maximum) write event for that location.

Definition 3.14 (Global Machine State)**.**

$$\mathcal{N} \in \mathit{RaceView} ::= \mathit{View}$$
$$\varsigma \in \mathit{GlobalState} ::= \mathit{MsgPool} \times \mathit{RaceView}$$

The global machine state ς is a pair $(\mathcal{M}, \mathcal{N})$ of the global memory $\mathcal{M}$ and a simple view $\mathcal{N}$ that is the state of the race detector (§3.4).

Property 3.15 (Global Machine State Wellformedness)**.** A global state $(\mathcal{M}, \mathcal{N})$ is wellformed if

- $\mathcal{M}$ is wellformed (Property 3.13); and
- $\mathcal{N}$ is closed in $\mathcal{M}$; and
- $\mathcal{N}$ observe the deallocations in $\mathcal{M}$, *i.e.*, $\forall \ell, t.\, \mathcal{M}(\ell)(t).\mathsf{val} = \maltese \Rightarrow t \leq \mathcal{N}(\ell).\mathsf{w}$.

3.3 View-based RMC Semantics

We define the view-based semantics of ORC11, which describes the interactions between thread-views $\mathcal{V}$'s and the global memory $\mathcal{M}$. First we need a few auxiliary definitions.

Definition 3.16 (Memory Value Injection)**.** $\boxed{\omega \equiv v}$

The injection of memory values (*MemVal*) into values (*Val*) is defined by the following rules.

$$\frac{}{v \equiv v}\ \textsc{MVal-Val} \qquad \frac{}{\dagger \equiv \text{☣}}\ \textsc{MVal-AVal}$$

That is, if the memory value is a value v, then it is returned as is. If the memory value is the allocated value $\dagger$, then poison ☣ is returned. There is no injection of the deallocated value $\maltese$ into values.

Definition 3.17 (Unallocated Locations)**.** $\boxed{\ell \in \text{unalloc}(\mathcal{M})}$

A location ℓ is called *unallocated* in $\mathcal{M}$ if it has not been allocated or it has been deallocated in $\mathcal{M}$.

$$\frac{\ell \notin \mathtt{dom}(\mathcal{M})}{\ell \in \text{unalloc}(\mathcal{M})} \qquad \frac{\exists t.\, \mathcal{M}(\ell)(t) = (\maltese, \_)}{\ell \in \text{unalloc}(\mathcal{M})}$$

Notation 3.18 (Function Computations)**.** We use the notation $(x)\,?\,y : z$ to denote the expression that if x is true, then y is returned, otherwise z is returned.

For a finite, partial function f, the notation $f[x \leftarrow y]$ denotes the same function f but with the key x updated to the value y.

For a record r, the notation $\{r\,[x := y]\}$ (including braces) denotes the same record r but with the key x updated to the value y.

Remark 3.19 (Conditions on Allocations and Deallocations)**.** C11 only specifies that the lifetime of an object is from its allocation to deallocation, but does not specify a synchronization condition or possible races between allocation/deallocation and normal accesses. Here, we employ the following conditions that are widely thought to be reasonable.

• *The allocation of a block must happen-before all accesses to it.* (Alloc-Safe)

• *The deallocation of a block must happen-after all accesses to it.* (Dealloc-Safe)

We consider violations of these conditions data races and, thus, undefined behavior. Correspondingly, we will treat the semantics of allocations and deallocations as that of non-atomic writes.

We now define two functions (written in form of relations) to compute the resulting thread-view of a thread after a read or a write, in Figure 3.2.

Definition 3.20 (Post-Read Thread-Views). $\mathcal{V} \xrightarrow{\mathtt{R}:o,\ell,t,V_r,r} \mathcal{V}'$

OM-POST-READ-TVIEW (Figure 3.2) computes the thread-view *after* a read $\mathcal{V}' = (V_{\text{rel}}, V_{\text{frel}}, V'_{\text{cur}}, V'_{\text{acq}})$ from the thread-view *before* the read $\mathcal{V} = (V_{\text{rel}}, V_{\text{frel}}, V_{\text{cur}}, V_{\text{acq}})$, using the read's access mode o and location ℓ, the timestamp t and the view $V_r \in \mathit{View}^+$ of the write message that the read reads from, and a fresh action id r ($\{r\} \in \mathit{ActIds}$) that identifies the read.[14] The computation is as follows.

[14] Note that if $o = \mathtt{na}$, then $V_r = \bot$ (None).

- $V_{\text{cur}}(\ell).\mathsf{w} \leq t$: the read only reads a write event (identified by t) that is not mo-earlier than the current view. Intuitively, this restriction helps establish axioms like C11-CoWR and C11-CoRR.
- $V_r(\ell) \leq t$: the operational semantics maintains an invariant that the timestamp t of the write to ℓ is the maximum timestamp in the write's view V_r.
- The view V tracks the identifying information of the read and the write that the read reads from. In particular, $V.\mathsf{w}$ is the timestamp t of the write. The read id r is added to the non-atomic read component $V.\mathsf{nr}$ if this is a non-atomic read ($o = \mathtt{na}$), otherwise r is added to the atomic read component $V.\mathsf{ar}$.
- A read only changes the current and acquire (simple) views of $\mathcal{V}$.
- The view V is joined into both the new current and acquire views V'_{cur} and V'_{acq}, so that both views observe at least the read and the write.
- If this is an atomic read ($\mathtt{rlx} \sqsubseteq o$), then the view V_r of the write is also joined into the acquire view V'_{acq}. This encodes the delayed synchronization of relaxed reads, where the view V_r sent over the write is temporarily stored in the acquire view V'_{acq} and will only later be restored into the current view with an acquire fence (recall Example 3.3).
- If this is at least an acquire read ($\mathtt{acq} \sqsubseteq o$), then the view V_r of the write is immediately joined into the current view V'_{cur} (recall Example 3.2).
- The computation maintains wellformedness of thread-views (Property 3.9).

Definition 3.21 (Post-Write Thread-Views). $\mathcal{V} \xrightarrow{\mathtt{W}:o,\ell,t,V_r^?,V_w^?} \mathcal{V}'$

OM-POST-WRITE-TVIEW (Figure 3.2) computes the thread-view *after* a write $\mathcal{V}' = (V'_{\text{rel}}, V_{\text{frel}}, V'_{\text{cur}}, V'_{\text{acq}})$ from the thread-view *before* the write $\mathcal{V} = (V_{\text{rel}}, V_{\text{frel}}, V_{\text{cur}}, V_{\text{acq}})$, using the write's access mode o and location ℓ, and a fresh timestamp t to identify the write, and the view $V_r \in \mathit{View}^+$ that the write reads from in case it is an Update (U). Additional, it computes the view V_w of the write itself. The computation is as follows.

OM-POST-READ-TVIEW

$$\frac{\begin{array}{c} V_{\mathrm{cur}}(\ell).\mathsf{w} \le t \qquad V_r(\ell) \le t \\ V = [\ell \leftarrow \{\mathsf{w} := t; \mathsf{aw} := \varnothing; \mathsf{nr} := (o = \mathbf{na})\ ?\ \{r\} : \varnothing; \mathsf{ar} := (o \sqsubseteq \mathbf{rlx})\ ?\ \{r\} : \varnothing\}] \\ V'_{\mathrm{cur}} = (\mathbf{acq} \sqsubseteq o)\ ?\ V_{\mathrm{cur}} \sqcup V \sqcup V_r : V_{\mathrm{cur}} \sqcup V \\ V'_{\mathrm{acq}} = (\mathbf{rlx} \sqsubseteq o)\ ?\ V_{\mathrm{acq}} \sqcup V \sqcup V_r : V_{\mathrm{acq}} \sqcup V \end{array}}{(V_{\mathrm{rel}}, V_{\mathrm{frel}}, V_{\mathrm{cur}}, V_{\mathrm{acq}}) \xrightarrow{\mathtt{R}:o,\ell,t,V_r,r} (V_{\mathrm{rel}}, V_{\mathrm{frel}}, V'_{\mathrm{cur}}, V'_{\mathrm{acq}})}$$

OM-POST-WRITE-TVIEW

$$\frac{\begin{array}{c} V_{\mathrm{cur}}(\ell).\mathsf{w} < t \\ V = [\ell \leftarrow \{\mathsf{w} := t; \mathsf{aw} := (\mathbf{rlx} \sqsubseteq o)\ ?\ \{t\} : \varnothing; \mathsf{nr} := \varnothing; \mathsf{ar} := \varnothing\}] \\ V'_{\mathrm{cur}} = V_{\mathrm{cur}} \sqcup V \qquad V'_{\mathrm{acq}} = V_{\mathrm{acq}} \sqcup V \\ V' = V_{\mathrm{rel}}(\ell) \sqcup (\mathbf{rel} \sqsubseteq o)\ ?\ V'_{\mathrm{cur}} : V \qquad V'_{\mathrm{rel}} = V_{\mathrm{rel}}[\ell \leftarrow V'] \\ V_w = (\mathbf{rlx} \sqsubseteq o)\ ?\ V' \sqcup V_{\mathrm{frel}} \sqcup V_r : \bot \end{array}}{(V_{\mathrm{rel}}, V_{\mathrm{frel}}, V_{\mathrm{cur}}, V_{\mathrm{acq}}) \xrightarrow{\mathtt{W}:o,\ell,t,V_r,V_w} (V'_{\mathrm{rel}}, V_{\mathrm{frel}}, V'_{\mathrm{cur}}, V'_{\mathrm{acq}})}$$

FIGURE 3.2: Computations of post thread-views for read and write operations.

- $V_{\mathrm{cur}}(\ell).\mathsf{w} < t$: the fresh timestamp t picked for the write must be mo-later than the current view of ℓ.
- The view V tracks the identifying information of the write. In particular, $V.\mathsf{w}$ is the timestamp t of the write. t is also added to the atomic write component $V.\mathsf{aw}$ if this is an atomic write.
- The write updates the current and acquire (simple) views, and the component ℓ of the per-location release view V_{rel} of $\mathcal{V}$.
- The view V is joined into both the new current and acquire views V'_{cur} and V'_{acq}, so that both views observe at least the write.
- The view V' is the new release write for the location ℓ, and is updated into to $V'_{\mathrm{rel}}(\ell)$ ($V'_{\mathrm{rel}} = V_{\mathrm{rel}}[\ell \leftarrow V']$). In particular, if this is at least a release write, then ℓ's new release view V' also includes the *new* current view V'_{cur}. (Otherwise, if the write is at most a relaxed write, then the release views remain unchanged.) This means that the write releases its current view immediately (recall also Example 3.2), and this release write starts a new release sequence (Definition 2.12) for ℓ, so that po-later relaxed writes to the same ℓ will indeed release V'_{cur}.
- The view V_w of the write itself is $\bot$ if this is a non-atomic write. Otherwise, it includes at least (i) the new release view $V'_{\mathrm{rel}}(\ell)$ for ℓ, and (ii) the view V_{frel} of the most recent same-thread release fence, and (iii) the view V_r of another write that this write reads from in case it is an Update. All of these views establish this write's effects as a part of a release sequence for ℓ (see also Example 2.13).
- The computation maintains wellformedness of the thread-views (Property 3.9).

Finally, we can define the view-based semantics of ORC11.

$$\textsc{OM-READ}\quad \frac{\ell \notin \text{unalloc}(\mathcal{M}) \qquad \mathcal{M}(\ell)(t) = (\omega, V_r) \qquad \omega \equiv v \qquad \mathcal{V} \xrightarrow{\texttt{R}:o,\ell,t,V_r,r} \mathcal{V}'}{\mathcal{M} \mid \mathcal{V} \xrightarrow{\texttt{R}^o(\ell,v),r,[]} \mathcal{M} \mid \mathcal{V}'}$$

$$\textsc{OM-WRITE}\quad \frac{\ell \notin \text{unalloc}(\mathcal{M}) \qquad t \notin \mathcal{M}(\ell) \qquad \mathcal{M}' = \mathcal{M}[\ell \leftarrow \mathcal{M}(\ell)[t \leftarrow (v, V_w)]] \qquad \mathcal{V} \xrightarrow{\texttt{W}:o,\ell,t,\bot,V_w} \mathcal{V}'}{\mathcal{M} \mid \mathcal{V} \xrightarrow{\texttt{W}^o(\ell,v),\bot,[(t,v,V_w)]} \mathcal{M}' \mid \mathcal{V}'}$$

$$\textsc{OM-UPDATE}\quad \frac{\ell \notin \text{unalloc}(\mathcal{M}) \qquad \mathcal{M}(\ell)(t_r) = (v_r, V_r) \qquad t_w = t_r + 1 \qquad t_w \notin \mathcal{M}(\ell) \qquad \mathcal{M}' = \mathcal{M}[\ell \leftarrow \mathcal{M}(\ell)[t_w \leftarrow (v_w, V_w)]] \qquad \mathcal{V} \xrightarrow{\texttt{R}:o_r,\ell,t_r,V_r,r} \xrightarrow{\texttt{W}:o_w,\ell,t_w,V_r,V_w} \mathcal{V}'}{\mathcal{M} \mid \mathcal{V} \xrightarrow{\texttt{U}^{o_r,o_w}(\ell,v_r,v_w),r,[(t_w,v_w,V_w)]} \mathcal{M}' \mid \mathcal{V}'}$$

$$\textsc{OM-ACQ-FENCE}\quad \mathcal{M} \mid \mathcal{V} \xrightarrow{\texttt{F}^{\texttt{acq}},\bot,[]} \mathcal{M} \mid (\mathcal{V}.\mathsf{rel}, \mathcal{V}.\mathsf{frel}, \mathcal{V}.\mathsf{acq}, \mathcal{V}.\mathsf{acq})$$

$$\textsc{OM-REL-FENCE}\quad \mathcal{M} \mid \mathcal{V} \xrightarrow{\texttt{F}^{\texttt{rel}},\bot,[]} \mathcal{M} \mid (\left[\ell \leftarrow \mathcal{V}.\mathsf{cur} \mid \ell \in \mathtt{dom}(\mathcal{V}.\mathsf{rel})\right], \mathcal{V}.\mathsf{cur}, \mathcal{V}.\mathsf{cur}, \mathcal{V}.\mathsf{acq})$$

$$\textsc{OM-ALLOC}\quad \frac{\ell = (i, n') \qquad \{i\} \times \mathbb{N} \,\#\, \mathtt{dom}(\mathcal{M}) \qquad \mathcal{M}' = \mathcal{M}[\ell + m \leftarrow [t_m \leftarrow (\dagger, \bot)] \mid m \in [0, n)] \qquad \mathcal{V} \xrightarrow{\texttt{W:na},\ell+0,t_0,\bot,\bot} \cdots \xrightarrow{\texttt{W:na},\ell+m,t_m,\bot,\bot} \cdots \xrightarrow{\texttt{W:na},\ell+(n-1),t_{n-1},\bot,\bot} \mathcal{V}' \qquad ms = [(t_m, \dagger, \bot) \mid m \in [0, n)]}{\mathcal{M} \mid \mathcal{V} \xrightarrow{\texttt{A}(\ell,n),\bot,ms} \mathcal{M}' \mid \mathcal{V}'}$$

$$\textsc{OM-FREE}\quad \frac{\ell = (i, n') \qquad \mathtt{dom}(\mathcal{M}) \cap \{i\} \times \mathbb{N} = \{i\} \times ([\geq n', <n' + n]) \qquad \forall m \in [0, n).\, \ell + m \notin \text{unalloc}(\mathcal{M}) \qquad \forall m \in [0, n), t \in \mathtt{dom}(\mathcal{M}(\ell + m)).\, t < t_m \qquad \mathcal{M}' = \mathcal{M}[\ell + m \leftarrow [t_m \leftarrow (\maltese, \bot)] \mid m \in [0, n)] \qquad \mathcal{V} \xrightarrow{\texttt{W:na},\ell+0,t_0,\bot,\bot} \cdots \xrightarrow{\texttt{W:na},\ell+m,t_m,\bot,\bot} \cdots \xrightarrow{\texttt{W:na},\ell+(n-1),t_{n-1},\bot,\bot} \mathcal{V}' \qquad ms = [(t_m, \dagger, \bot) \mid m \in [0, n)]}{\mathcal{M} \mid \mathcal{V} \xrightarrow{\texttt{D}(\ell,n),\bot,ms} \mathcal{M}' \mid \mathcal{V}'}$$

FIGURE 3.3: View-based machine semantics.

Definition 3.22 (ORC11 View-based Reductions). $\boxed{\mathcal{M} \mid \mathcal{V} \to \mathcal{M}' \mid \mathcal{V}'}$

The relation $\mathcal{M} \mid \mathcal{V} \xrightarrow{\varepsilon,r^?,ms} \mathcal{M}' \mid \mathcal{V}'$ relates a pair of global memory $\mathcal{M}$ and a local thread-view $\mathcal{V}$ before a machine step that generates a memory event ε to a corresponding pair $\mathcal{M}'$ and $\mathcal{V}'$ after the step. r is an optional action id associated with the event if it is a read, and ms is a list of write messages generated by the event if it is a write. The rules of the view-based reductions are given in Figure 3.3.

- OM-READ says that a read $\texttt{R}^o(\ell, v)$ does not change the global memory $\mathcal{M}$, and is only possible if ℓ is alive in $\mathcal{M}$,[15] whose memory value ω is injected into the read value v (so that a read of an uninitialized location will return the poison value $\maltese$).[16] The thread-view is updated from $\mathcal{V}$ to $\mathcal{V}'$ using the timestamp t and the view V_r of

[15] see Definition 3.17.

[16] see Definition 3.16.

the write, and the fresh action id r for the read, following Definition 3.20.

- OM-WRITE says that a write $\mathtt{W}^o(\ell, v)$ is only possible if ℓ is alive in $\mathcal{M}$, and $\mathcal{M}$ is updated to $\mathcal{M}'$ with a new message (t, v, V_w) for ℓ, where t is a fresh timestamp in $\mathcal{M}$ for ℓ, and V_w is computed following Definition 3.21, which also defines how $\mathcal{V}'$ is computed. Note that there is no other constraint on the timestamp t, *e.g.*, it does not need to be the next largest timestamp for ℓ in $\mathcal{M}$. This allows "holes" in the set of used timestamps, so that writes to ℓ by other threads may come in later in ORC11 machine execution order, but actually ends up mo-earlier than the writes made by the thread in question. But do note also that Definition 3.21 requires that the timestamp t is at least mo-later than the writes for ℓ seen by the current thread-view, so as to guarantee that mo_ℓ agrees with the current thread's po.

- OM-UPDATE combines the effects of both OM-READ and OM-WRITE, saying that an update $\mathtt{U}^{o_r,o_w}(\ell, v_r, v_w)$ reads from an existing write message (t_r, v_r, V_r) and updates the memory $\mathcal{M}$ with a new write message (t_w, v_w, V_w). The new write message's timestamp t_w must be fresh for ℓ in $\mathcal{M}$, and must be next to the read message's timestamp t_r: $t_w = t_r + 1$, so as to exclude holes between the two messages in mo_ℓ, and thus to disallow other threads' concurrent writes to come in between this update and the write that it reads from. This guarantees the *uniqueness* of a successful update event U who represents the effects of RMW instructions: if multiple RMW instructions are racing on reading the same value, then only one of them will successfully perform a write. Finally, the new thread-view $\mathcal{V}'$ is computed from $\mathcal{V}$ using r, V_r, and V_w, following Definition 3.20 and then Definition 3.21.

- OM-ACQ-FENCE simply joins the thread-view's acquire component $\mathcal{V}$.acq into the new current component $\mathcal{V}'$.cur, restoring views acquired through earlier relaxed reads and thus establishing synchronizations (recall Example 3.3).

- OM-REL-FENCE stores the thread-view's current component $\mathcal{V}$.cur into the new release components $\mathcal{V}'$.rel (the per-location release views) and $\mathcal{V}'$.frel (the release-fence view).

- OM-ALLOC says that an allocation $\mathtt{A}(\ell, n)$ of a fresh block (whose base is ℓ) inserts n write messages ms into the global memory $\mathcal{M}$, each for a location in the block. The new write messages ms all have the allocated memory value †. The new thread-view $\mathcal{V}'$ is computed by applying Definition 3.21 for n consecutive non-atomic writes.

- Finally, OM-FREE says that a deallocation $\mathtt{D}(\ell, n)$ requires that ℓ is indeed the base location of a block whose size is n, and all locations in the block are alive ($\forall m \in [0, n). \ell + m \notin \mathrm{unalloc}(\mathcal{M})$). The

deallocation inserts n write messages ms into the global memory $\mathcal{M}$, all with the deallocated memory value $\lightning$, and the maximal timestamps ($\forall m \in [0, n), t \in \mathtt{dom}(\mathcal{M}(\ell + m)).\, t < t_m$).

Property 3.23 (Wellformedness of View-based Reductions)**.** The pair $\mathcal{M} \mid \mathcal{V}$ is wellformed if $\mathcal{V}$ is wellformed (Property 3.9) and $\mathcal{M}$ is wellformed (Property 3.13) and $\mathcal{V}$ is closed in $\mathcal{M}$ (Property 3.12).

Lemma 3.24 (Invariant of ORC11 View-based Reductions)**.** *The* ORC11 *view-based reductions (Definition 3.22) maintain wellformedness (Property 3.23), and maintain that the thread-views only grow ($\mathcal{V} \sqsubseteq \mathcal{V}'$).*

3.4 The Data-Race Detector

The goal of the race detector, as the name suggest, is to raise undefined behavior (UB) if the program is racy in C11/RC11 (Definition 2.16 and 2.18). That is, if a program may have a RC11-consistent execution graph that is racy, then the program must also have a ORC11 execution where the data-race detector (defined in this section) raises UB.

In this work, we model UB as *stuckness*: we say that the execution gets stuck if there is no further reduction possible when the reducing expression has not reaches a value. (If the reducing expression is already a value, then the execution has safely terminated.)[17] We will not see the expression reductions until §4.2, so in the following, we simply consider stuckness as "there is no further reduction possible for the current machine state".

[17] There may be several ways for an execution to run into UBs, *i.e.*, to get stuck (*e.g.*, performing computations on poison ☣, see §4.2), so it may be beneficial to distinguish the different reasons for the different UB types, rather than collapsing all of them into a single stuck state. This can be done by introducing error machine states. However, in this work, we do not need such details, and therefore decide to simply use stuckness.

The aim of the race detector is to make sure locally that racy accesses where at least one is non-atomic must get stuck. For this, the race detector relies on the global machine state, which includes the global memory $\mathcal{M}$ and the race detector's state $\mathcal{N} \in \mathit{View}$, in combination of the executing thread π's thread-view $\mathcal{V}$. In practice, only the current component $\mathcal{V}.\mathsf{cur} \in \mathit{View}$ will be used, because that view encodes what have happened before the thread π's program counter $\mathtt{PC}_\pi$, and recall that races are due to the lack of hb edges between conflicting accesses.

Recall from Definition 3.6 that both $\mathcal{N}$ and $\mathcal{V}.\mathsf{cur}$ tracks, for each location ℓ, the most recent write timestamp and sets of action ids for atomic writes, non-atomic reads, and atomic reads. The differences are that (1) $\mathcal{N}$ tracks all actions that have been performed globally by *all* threads, while $\mathcal{V}.\mathsf{cur}$ only tracks locally what π has observed, and (2) $\mathcal{N}.\mathsf{w}(\ell)$ only tracks the globally most recent *non-atomic* write for ℓ, not the most recent write for ℓ.

The race detector checks for *data-race freedom* (DRF) for each memory access on an ℓ that π is going to perform. If it is not data-race free, then the execution gets stuck. Otherwise, the race detector state $\mathcal{N}$ is updated correspondingly to track the newly performed access. The race detector is therefore defined using two definitions: the DRF pre-condition (Definition 3.25) which defines the pre-condition of a data-race free access, and the DRF post-condition (Definition 3.26) which computes the post state $\mathcal{N}'$ for the race detector.

DRF-READ-NA
$$\frac{\mathcal{N}(\ell).\mathsf{w} \le \mathcal{V}.\mathsf{cur}(\ell).\mathsf{w} \qquad \forall t \in \mathtt{dom}(\mathcal{M}(\ell)).\, t \le \mathcal{V}.\mathsf{cur}(\ell).\mathsf{w} \qquad \mathcal{N}(\ell).\mathsf{aw} \sqsubseteq \mathcal{V}.\mathsf{cur}(\ell).\mathsf{aw}}{\mathcal{M},\mathcal{N},\mathcal{V} \vdash \mathsf{RaceFree}(\mathtt{R}^{\mathsf{na}}(\ell, v))}$$

DRF-WRITE-NA
$$\frac{\mathcal{N}(\ell) \sqsubseteq \mathcal{V}.\mathsf{cur}(\ell) \qquad \forall t \in \mathtt{dom}(\mathcal{M}(\ell)).\, t \le \mathcal{V}.\mathsf{cur}(\ell).\mathsf{w}}{\mathcal{M},\mathcal{N},\mathcal{V} \vdash \mathsf{RaceFree}(\mathtt{W}^{\mathsf{na}}(\ell, v))}$$

DRF-READ-AT
$$\frac{\mathtt{rlx} \sqsubseteq o \qquad \mathcal{N}(\ell).\mathsf{w} \le \mathcal{V}.\mathsf{cur}(\ell).\mathsf{w}}{\mathcal{M},\mathcal{N},\mathcal{V} \vdash \mathsf{RaceFree}(\mathtt{R}^{o}(\ell, v))}$$

DRF-WRITE-AT
$$\frac{\mathtt{rlx} \sqsubseteq o \qquad \mathcal{N}(\ell).\mathsf{w} \le \mathcal{V}.\mathsf{cur}(\ell).\mathsf{w} \qquad \mathcal{N}(\ell).\mathsf{nr} \sqsubseteq \mathcal{V}.\mathsf{cur}(\ell).\mathsf{nr}}{\mathcal{M},\mathcal{N},\mathcal{V} \vdash \mathsf{RaceFree}(\mathtt{W}^{o}(\ell, v))}$$

DRF-UPDATE
$$\frac{\mathcal{M},\mathcal{N},\mathcal{V} \vdash \mathsf{RaceFree}(\mathtt{R}^{o_r}(\ell, v_r)) \qquad \mathcal{M},\mathcal{N},\mathcal{V} \vdash \mathsf{RaceFree}(\mathtt{W}^{o_w}(\ell, v_w))}{\mathcal{M},\mathcal{N},\mathcal{V} \vdash \mathsf{RaceFree}(\mathtt{U}^{o_r,o_w}(\ell, v_r, v_w))}$$

DRF-ALLOC
$$\mathcal{M},\mathcal{N},\mathcal{V} \vdash \mathsf{RaceFree}(\mathtt{A}(\ell, n))$$

DRF-DEALLOC
$$\frac{\forall i \in [<n], t' \in \mathtt{dom}(\mathcal{M}(\ell+i)).\, t' \le \mathcal{V}.\mathsf{cur}(\ell).\mathsf{w} \qquad \forall i \in [<n].\, \mathcal{N}(\ell+i) \sqsubseteq \mathcal{V}.\mathsf{cur}(\ell+i)}{\mathcal{M},\mathcal{N},\mathcal{V} \vdash \mathsf{RaceFree}(\mathtt{D}(\ell, n))}$$

DRF-FENCE
$$\mathcal{M},\mathcal{N},\mathcal{V} \vdash \mathsf{RaceFree}(\mathtt{F}^{o})$$

FIGURE 3.4: Data-race free (DRF) pre-conditions.

Definition 3.25 (DRF Pre-conditions)**.** $\boxed{\mathcal{M},\mathcal{N},\mathcal{V} \vdash \mathsf{RaceFree}(\varepsilon)}$

$\mathcal{M},\mathcal{N},\mathcal{V} \vdash \mathsf{RaceFree}(\varepsilon)$ says that the memory access ε is data-race free when executed with the global state $(\mathcal{M},\mathcal{N})$ by a thread whose thread-view is $\mathcal{V}$. The rules are given in Figure 3.4.

- DRF-READ-NA says that a non-atomic read from ℓ is data-race free if the thread has observed *all writes* to ℓ, atomic and non-atomic, tracked globally by $\mathcal{M}$ and $\mathcal{N}$. Note that the conditions concerning $\mathcal{V}.\mathsf{cur}(\ell).\mathsf{w}$ by themselves are not sufficient: they only maintain that $\mathcal{V}$ has observed the mo_ℓ-maximum non-atomic write, which is only sufficient to guarantee observations of all non-atomic writes which must happen sequentially. The condition $\mathcal{N}(\ell).\mathsf{aw} \sqsubseteq \mathcal{V}.\mathsf{cur}(\ell).\mathsf{aw}$ guarantees the observations of all atomic writes: the atomic writes can be safely concurrent with one another, so a set of timestamps are needed instead of just a simple timestamp. Note that a non-atomic read can be safely executed concurrently with other reads, atomic or non-atomic.

- DRF-WRITE-NA says that a non-atomic write to ℓ is data-race free if the thread has observed *all memory accesses* to ℓ.

- DRF-READ-AT says that an atomic read from ℓ is data-race free if the thread has observed the latest non-atomic write to ℓ. Note that an atomic read can be safely executed concurrently with other reads, atomic or non-atomic, and atomic writes.
- DRF-WRITE-AT says that an atomic write to ℓ is data-race free if the thread has observed *all non-atomic accesses*, reads or writes.
- DRF-UPDATE is simply a combination of DRF-READ-AT and DRF-WRITE-AT. Note that an Update (U) does not support non-atomic (**na**) accesses.
- DRF-ALLOC says that the allocation of a fresh block is always data-race free. Note that we do not model out-of-memory errors.
- DRF-DEALLOC is simply an iteration of DRF-WRITE-NA for the whole block.
- DRF-FENCE says that fences are never racy.

Definition 3.26 (DRF Post-conditions). [18]

$$\boxed{\mathcal{N} \overset{\varepsilon, r^?, ms}{\Longrightarrow} \mathcal{N}'}$$

The relation $\mathcal{N} \overset{\varepsilon, r^?, ms}{\Longrightarrow} \mathcal{N}'$ defines how a race-free event ε for ℓ updates the global race detector state from $\mathcal{N}$ to $\mathcal{N}'$, using the optional action id r if ε is a read, and the list of write messages ms if ε is a write. The rules are given in Figure 3.5. Note that we use the record update notation defined in Notation 3.18.

- DRF-POST-READ-NA requires that the action id r is picked fresh globally to identify this read, and the race detector's component for tracking ℓ's non-atomic reads ($\mathcal{N}'(\ell)$.nr) is extended with r.
- DRF-POST-WRITE-NA says that a non-atomic write with the message m simply extends the race detector's component for tracking ℓ's non-atomic writes ($\mathcal{N}'(\ell)$.w) with the write timestamp m.ts.
- DRF-POST-READ-AT says that the effect of an atomic read on $\mathcal{N}$ is similar to that of a non-atomic one (DRF-POST-READ-NA), but instead changes the component for tracking ℓ's atomic reads.
- DRF-POST-WRITE-AT says that the effect of an atomic write on $\mathcal{N}$ is similar to that of a non-atomic one (DRF-POST-WRITE-NA), but instead changes the component for tracking ℓ's atomic writes.
- DRF-POST-UPDATE combines the effects of DRF-POST-READ-AT and DRF-POST-WRITE-AT.
- DRF-POST-ALLOC says that the race detector state is extended with simple observations on the non-atomic write timestamps m_i.ts for the whole newly allocated block.
- DRF-POST-DEALLOC is an iteration of DRF-POST-WRITE-NA for the whole block that is deallocated.

[18] The racing-ghost notation is due to Jan-Oliver Kaiser.

$$\textsc{DRF-Post-read-na}\quad \frac{r \notin \mathcal{N}(\ell).\mathsf{nr} \qquad \mathcal{N}' = \mathcal{N}[\ell \leftarrow \{\mathcal{N}(\ell)\,[\mathsf{nr} := \mathcal{N}(\ell).\mathsf{nr} \cup \{r\}]\}]}{\mathcal{N} \xrightarrow{\mathtt{R}^{\mathsf{na}}(\ell,v),r,[]} \mathcal{N}'}$$

$$\textsc{DRF-Post-write-na}\quad \frac{\mathcal{N}' = \mathcal{N}[\ell \leftarrow \{\mathcal{N}(\ell)\,[\mathsf{w} := m.\mathsf{ts}]\}]}{\mathcal{N} \xrightarrow{\mathtt{W}^{\mathsf{na}}(\ell,v),\bot,[m]} \mathcal{N}'}$$

$$\textsc{DRF-Post-read-at}\quad \frac{\mathbf{rlx} \sqsubseteq o \qquad r \notin \mathcal{N}(\ell).\mathsf{ar} \qquad \mathcal{N}' = \mathcal{N}[\ell \leftarrow \{\mathcal{N}(\ell)\,[\mathsf{ar} := \mathcal{N}(\ell).\mathsf{ar} \cup \{r\}]\}]}{\mathcal{N} \xrightarrow{\mathtt{R}^{o}(\ell,v),r,[]} \mathcal{N}'}$$

$$\textsc{DRF-Post-write-at}\quad \frac{\mathbf{rlx} \sqsubseteq o \qquad \mathcal{N}' = \mathcal{N}[\ell \leftarrow \{\mathcal{N}(\ell)\,[\mathsf{aw} := \mathcal{N}(\ell).\mathsf{aw} \cup \{m.\mathsf{ts}\}]\}]}{\mathcal{N} \xrightarrow{\mathtt{W}^{o}(\ell,v),\bot,[m]} \mathcal{N}'}$$

$$\textsc{DRF-Post-update}\quad \frac{r \notin \mathcal{N}(\ell).\mathsf{ar} \qquad \mathcal{N}' = \mathcal{N}[\ell \leftarrow \{\mathcal{N}(\ell)\,[\mathsf{ar} := \mathcal{N}(\ell).\mathsf{ar} \cup \{r\}\,;\mathsf{aw} := \mathcal{N}(\ell).\mathsf{aw} \cup \{m.\mathsf{ts}\}]\}]}{\mathcal{N} \xrightarrow{\mathtt{U}^{o_r,o_w}(\ell,v_r,v_w),r,[m]} \mathcal{N}'}$$

$$\textsc{DRF-Post-alloc}\quad \frac{\mathcal{N}' = \mathcal{N}[\ell + i \leftarrow \{\mathsf{w} := m_i.\mathsf{ts}, \mathsf{aw} := \varnothing, \mathsf{nr} := \varnothing, \mathsf{ar} := \varnothing\} \mid i \in [<n]]}{\mathcal{N} \xrightarrow{\mathtt{A}(\ell,n),\bot,[m_0 \ldots m_{n-1}]} \mathcal{N}'}$$

$$\textsc{DRF-Post-dealloc}\quad \frac{\mathcal{N}' = \mathcal{N}[\ell + i \leftarrow \{\mathcal{N}(\ell+i)\,[\mathsf{w} := m_i.\mathsf{ts}]\} \mid i \in [<n]]}{\mathcal{N} \xrightarrow{\mathtt{D}(\ell,n),\bot,[m_0 \ldots m_{n-1}]} \mathcal{N}'}$$

$$\textsc{DRF-Post-fence}\quad \mathcal{N} \xrightarrow{\mathtt{F}^{o},\bot,[]} \mathcal{N}$$

FIGURE 3.5: Data-race free (DRF) post-conditions.

- DRF-POST-FENCE says that fences do not affect the race detector state.

Lemma 3.27. *The DRF post-conditions (Definition 3.26) only grow the data-race view,* i.e.*, $\mathcal{N} \sqsubseteq \mathcal{N}'$. When combined with the view-based reductions (Definition 3.22), the DRF post-conditions also maintain wellformedness of the global machine state (Property 3.15).*

3.5 Comparison with iGPS Race Detector

ORC11 is the first operational semantics that incorporates a race detector for non-atomic accesses into a language with release-acquire accesses, relaxed accesses, and fences. ORC11's race detector extends the race detector Kaiser et al.[19] developed for iGPS, in order to address the extra effects of relaxed accesses. To explain the necessity of this extension, we

[19] Kaiser et al., "Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris" [Kai+17].

first discuss why the approach of Kaiser et al. does not scale to relaxed accesses.

The iGPS race detector, introduced by Kaiser et al. for the release-acquire/non-atomic (RA+NA) fragment of C11, is somewhat unusual in that it does not in fact detect all races in *every* execution. Instead, although iGPS forbids write-before-read races—that is, races where a write is interleaved before a racing read—it *allows* read-before-write races—where a read is interleaved before a racing write.

Example 3.28 (iGPS Race Detector Asymmetry)**.** To illustrate this asymmetry, consider the following example code:

$$\ell_x := 0; \\ {}^*\ell_x \mid\mid \ell_x := 37$$

In this program there are two possible interleavings, both of which are considered racy by C11. The iGPS race detector detects a race in the interleaving where the read from ℓ_x is executed after the write to ℓ_x, but it does not detect a race in the interleaving where ℓ_x is read first.

The upside of iGPS's approach is that reads do not need to be tracked by the race detector, which reduces the amount of state in the operational semantics. The downside, of course, is that some races are not detected—a seemingly rather severe problem for a race detector! The reason this is not a problem in iGPS is that Hoare triples imply absence of races for *all* executions of a program. In order to be able to claim that the iGPS *logic* ensures absence of data races according to C11, it thus suffices for the race detector in the operational semantics to detect a race on *some* execution of every program that is racy according to C11. And indeed it does: for programs with only release-acquire and non-atomic accesses (the domain of iGPS), for any execution with a read-before-write race, there is always a differently interleaved execution with a write-before-read race, which iGPS's race detector will detect.

In the presence of relaxed accesses, however, the iGPS race detector is no longer sufficient, because the property mentioned above is no longer true. That is, it is possible to construct programs that have executions in which the read-before-write races happen, but there is no interleaving where the write will be executed before the read.

Example 3.29 (No Write-before-Read Races)**.** Consider the following program:

$$\ell_x := 0; \ell_y := 0; \\ {}^*\ell_x; \ell_y :=_{\texttt{rlx}} 1 \mid\mid \texttt{while}\,({}^{*\texttt{rlx}}\ell_y == 0); \ell_x := 37$$

Here, the non-atomic read in the left thread is guaranteed to be executed before the non-atomic write to ℓ_x in the right thread, and there is no interleaving where the reverse can happen. The iGPS race detector would *not* declare this program racy, but the two accesses to ℓ_x are not related by happens-before and are thus considered a race by C11.

To account for such programs, ORC11's race detector extends iGPS's, which already tracked non-atomic writes, to track *all* memory access events, including atomic writes, and atomic and non-atomic reads in the local thread-views (see Definition 3.6). These events then must be sent across threads to perform synchronization and to ensure data-race freedom. In Example 3.29 above, the non-atomic read of ℓ_x by the left thread, when executed, will add a fresh read event r_{na} into $\mathcal{N}$'s global set of non-atomic reads for ℓ_x ($\mathcal{N}(\ell_x).\mathsf{nr}$). The non-atomic write of ℓ_x by the right thread is guaranteed to be executed after the read by the left thread. However, when the write is executed, the race detector requires that the right thread must have observed in its local current view *all* read events (DRF-WRITE-NA), including r_{na}, in order to be deemed non-racy. Since the right thread did not synchronize with the left thread to obtain r_{na} in its local view, its write to ℓ_x will be declared racy by the ORC11 race detector.

3.6 The Correspondence between RC11 and ORC11

To show a correspondence between ORC11 and RC11, we exploit the fact that ORC11 is very close to the "promise-free" fragment of Kang et al.[20] extended with non-atomics and a race detector. Kang et al. already proved a formal correspondence between their promise-free fragment and C11.[21] Building on their result, we show an *on-paper* correspondence proof between ORC11 and RC11. This proof has been provided in the technical appendix of the $\mathsf{RB}_{\mathtt{rlx}}$ paper,[22] and we only sketch its summary here.

It is woth noting that our language is actually the combination of the memory model of ORC11 and the λ_{Rust} expression language (Chapter 4). Together, ORC11 + λ_{Rust} is more conservative and declares more programs as having undefined behaviors than RC11. For example, λ_{Rust} can trigger UBs if we compare two values that are incomparable (Definition 4.9). Furthermore, the race detector in ORC11 is stronger, *i.e.*, detects more races, than RC11. In particular, ORC11 does not permit reducing an RMW operation with the **acq** access mode in the presence of an unsynchronized non-atomic read *even though* the RMW itself synchronizes with the non-atomic read. In contrast, the self-synchronizing nature of RMW leads to RC11 accepting this particular behavior as non-racy.

Consequently, we cannot show an equivalence in behaviors between ORC11 + λ_{Rust} and RC11. Instead, we also combine RC11 with λ_{Rust}, and relates ORC11 + λ_{Rust} with RC11 + λ_{Rust}. To simplify the proof, we allow RC11 to take λ_{Rust}'s expression reduction steps that are disallowed with ORC11. In particular, the declarative semantics in RC11 + λ_{Rust} may compare arbitrary values with each other, whereas ORC11 + λ_{Rust} will get stuck in some of these cases.

Ultimately, we want to show that if a program has defined behaviors in ORC11 + λ_{Rust},[23] then it must have defined behaviors in RC11 + λ_{Rust}.[24] Conversely, we show that any (UB-triggering) racy RC11-consistent execution[25] can be replayed as a racy execution in ORC11.

[20] Kang et al., "A promising semantics for relaxed-memory concurrency" [Kan+17].

[21] [Kan+17], Appendix B: Proof of Theorem 5.

[22] Dang et al., *Technical Appendix: RustBelt Meets Relaxed Memory* [Dan+20b], §2.

[23] which is a proof dischargeable using our separation logics

[24] see Definition 2.18

[25] see Definition 2.14 and Definition 2.16

Following Kang et al., we decompose the correspondence proof into 2 steps. First, we prove that any racy RC11 execution of the program can be replayed as a racy execution in the *Operational Graph Semantics* (OGS). Second, we prove that the racy OGS execution can be replayed as a racy execution in ORC11. The OGS is designed to be an intermediate mixture of RC11 and ORC11: its state is a RC11-consistent execution graph G, and every memory access operationally extends G without violating OGS's own race detector. OGS's race detector is stated in terms of execution graphs, but encodes exactly the rules of ORC11's race detector.[26] OGS is then extended with the same λ_{Rust}'s expression reduction steps as that of RC11 (which can compare arbitrary values).

We then show that (1a) any non-racy, consistent execution of RC11 + λ_{Rust} can be replayed as a non-racy *or* racy consistent execution of OGS + λ_{Rust};[27] and (1b) any program that has racy executions in RC11 + λ_{Rust} also has racy executions in OGS + λ_{Rust}.[28] Then, to relate OGS + λ_{Rust} and ORC11 + λ_{Rust}, we relate OGS's state with the timestamp assignments in ORC11's thread-view and race detector states. We then show that (2a) any non-racy, consistent execution of OGS + λ_{Rust} can be replayed as a non-stuck *or* stuck execution of ORC11 + λ_{Rust};[29] and (2b) any racy execution of OGS + λ_{Rust} can get stuck in ORC11 + λ_{Rust}.[30] Finally, combining all of those results, we show that any program that has racy executions in RC11 + λ_{Rust} can get stuck in ORC11 + λ_{Rust}.[31]

[26] see Figure 3.4

[27] [Dan+20b], Lemma 3.

[28] [Dan+20b], Lemma 4.

[29] [Dan+20b], Lemma 6.

[30] [Dan+20b], Lemma 7.

[31] [Dan+20b], Theorem 1.

CHAPTER SUMMARY. This chapter explains the view-based semantics and the race-detector semantics of ORC11, and illustrates how it is related to RC11. In the next chapter, we present the λ_{Rust} language and explain how to combine it with ORC11 to achieve our target language that will be used to instantiate Iris.

4

The Relaxed λ_{Rust} Language

The *relaxed* λ_{Rust} language retrofits the original RustBelt's λ_{Rust}[1] on top of the relaxed memory semantics of ORC11. In this chapter, we briefly review λ_{Rust} and formally define how to "plug" it in with ORC11 to obtain our target language.

[1] Jung et al., "RustBelt: Securing the Foundations of the Rust Programming Language" [Jun+18a].

It worths noting up front that we (nor the origin RustBelt's authors) do not plan to tackle the Herculean task of giving a formal definition of the *complete* Rust language: the core λ_{Rust} calculus only captures central features of the Rust language, and the original semantics assume a SC memory model,[2] and we extend the operational semantics to cover relaxed memory accesses. Nevertheless, the reasoning principles we develop in this dissertation are not restricted to λ_{Rust} or Rust, and can be applied to other languages that employ the RC11 memory model (or stronger memory models).

[2] with a race detector that intuitively implements reader-writer locks on non-atomics locations.

4.1 Language Syntax

Definition 4.1 (λ_{Rust} Grammar)**.** The grammar is given in Figure 4.1. λ_{Rust} is a lambda calculus with:

- values that can be poison (☣),[3] a block-based location, an integer (meta-variable $z \in \mathbb{Z}$), or a recursive function (meta-variable f) that has a list of binders ($\overline{x}$) for the arguments.
- expressions (meta-variable e) that can be a value; or a variable (x); or a path operator ($e.e$) where the second operand (called the offset) must evaluate to an integer offset of the first operand; or a binary operator; or function application ($e(\overline{e})$) where the arguments are a list expressions; or a (switch) case block (**case** e **of** $\overline{e}$) that allows branching into a list of expressions; or a fork operator that supports forking new (detached) threads; or a memory instruction which can be a read, a write, a compare-and-swap (**CAS**), or a fence instruction with different consistency modes;[4] or an explicit allocation or deallocation instruction.

[3] see §3.2.

[4] see Definition 2.2.

Definition 4.2 (λ_{Rust} Left-to-Right Evaluation Contexts)**.** The reduction strategy of λ_{Rust}'s expressions is encoded using *evaluation contexts* $K \in$ *ECtx*. The approach of evaluation contexts decomposes an expression e into an evaluation context K and an expression e' that can perform a

$$
\begin{aligned}
v \in \mathit{Val} ::= {} & \lightning \mid \ell \mid z \mid \mathbf{rec}\, f(\overline{x}) := e \\
e \in \mathit{Expr} ::= {} & \mid v \mid x \mid e.e \\
& \mid e + e \mid e - e \mid e \le e \mid e == e \\
& \mid e(\overline{e}) \\
& \mid \mathbf{case}\, e\, \mathbf{of}\, \overline{e} \\
& \mid \mathbf{fork}\{e\} \\
& \mid {}^{*o}e \mid e_1 :=_o e_2 \mid \mathbf{CAS}^{o_f,o_r,o_w}(e_0, e_1, e_2) \mid \mathbf{fence}_o \\
& \mid \mathbf{alloc}(e) \mid \mathbf{free}(e_1, e_2) \\
K \in \mathit{ECtx} ::= {} & \mid \bullet \\
& \mid K.e \mid v.K \\
& \mid K + e \mid v + K \mid K - e \mid v - K \mid K \le e \mid v \le K \mid K == e \mid v == K \\
& \mid K(\overline{e}) \mid v(\overline{v} +\!\!+ [K] +\!\!+ \overline{e}) \\
& \mid \mathbf{case}\, K\, \mathbf{of}\, \overline{e} \\
& \mid {}^{*o}K \mid K :=_o e \mid v :=_o K \\
& \mid \mathbf{CAS}^{o_f,o_r,o_w}(K, e_1, e_2) \mid \mathbf{CAS}^{o_f,o_r,o_w}(v_0, K, e_2) \mid \mathbf{CAS}^{o_f,o_r,o_w}(v_0, v_1, K) \\
& \mid \mathbf{alloc}(K) \mid \mathbf{free}(K, e_2) \mid \mathbf{free}(e_1, K)
\end{aligned}
$$

FIGURE 4.1: The relaxed λ_{Rust} language syntax.

primitive reduction and so will be evaluated next, satisfying $e = K[e']$. The empty context $\bullet$ is called the "hole" where the next-to-be-evaluated expression is filled in.

The evaluation strategy is left-to-right call-by-value, and is given in Figure 4.1. Let us consider an example evaluation of an assignment $e_1 :=_o e_2$:

- The expression is first decomposed into $K_1 = \bullet :=_o e_2$ and e_1, which allows e_1 to be evaluated first.
- After e_1 is evaluated to a value v_1, the expression is $K_1[v_1] = v_1 :=_o e_2$, which can be decomposed into $K_2 = v_1 :=_o \bullet$, which now allows e_2 to be evaluated.
- Once e_2 is evaluated to a value v_2, the expression is $K_2[v_2] = v_1 :=_o v_2$, which is decomposed into $\bullet$ and $v_1 :=_o v_2$.
- The primitive reduction of assignments then can kick in and complete the evaluation.

Notation 4.3 (λ_{Rust} Syntactic Sugars)**.** Several syntactic sugars are taken as-is from the original RustBelt, given in Figure 4.2. Specifically:

- Non-recursive functions ($\lambda[\overline{x}].\, e$) simply ignore the recursive function argument. **let** bindings are used to declare local variables in λ_{Rust}, which are pure and do not occupy memory (they are not mutable nor addressable). They are simply evaluated and then substituted into the remaining expression, hence the definition using functions. Sequential composition is defined using **let** bindings.

$$
\begin{aligned}
\lambda[\overline{x}].\, e &::= \textbf{rec}\ \_([\overline{x}]) := e \\
\textbf{let}\ x = e\ \textbf{in}\ e' &::= (\lambda[x].\, e')([e]) \\
e'; e &::= \textbf{let}\ \_ = e'\ \textbf{in}\ e \\
\\
\textbf{false} &::= 0 \\
\textbf{true} &::= 1 \\
\textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2 &::= \textbf{case}\ e_0\ \textbf{of}\ [e_2, e_1] \\
\\
{}^*e &::= {}^{*\textsf{na}}e \\
e_1 := e_2 &::= e_1 :=_{\textsf{na}} e_2 \\
\\
\textbf{new} &::= \lambda[\mathtt{size}].\, \textbf{if}\ \mathtt{size} == 0\ \textbf{then}\ (42, 1337)\ \textbf{else}\ \textbf{alloc}(\mathtt{size}) \\
\textbf{delete} &::= \lambda[\mathtt{size}, \mathtt{ptr}].\, \textbf{if}\ \mathtt{size} == 0\ \textbf{then}\ \text{☣}\ \textbf{else}\ \textbf{free}(\mathtt{size}, \mathtt{ptr}) \\
\\
\textbf{memcpy} &::= \textbf{rec}\ \mathtt{memcpy}([\mathtt{dst}, \mathtt{len}, \mathtt{src}]) := \\
&\qquad \textbf{if}\ \mathtt{len} \le 0\ \textbf{then}\ \text{☣}\ \textbf{else} \\
&\qquad \mathtt{dst}.0 := \mathtt{src}.0; \\
&\qquad \mathtt{memcpy}([\mathtt{dst}.1, \mathtt{len} - 1, \mathtt{src}.1]) \\
e_1 :=_n {}^*e_2 &::= \textbf{memcpy}(e_1, n, e_2) \\
\\
e :\overset{\textsf{inj}\, i}{=\!=} () &::= e.0 := i \\
e_1 :\overset{\textsf{inj}\, i}{=\!=} e_2 &::= e_1.0 := i; e_1.1 := e_2 \\
e_1 :\overset{\textsf{inj}\, i}{=\!=}_n {}^*e_2 &::= e_1.0 := i; e_1.1 :=_n {}^*e_2 \\
\\
\textbf{skip} &::= \textbf{let}\ x = \text{☣}\ \textbf{in}\ \text{☣} \\
\textbf{newlft} &::= \text{☣} \\
\textbf{endlft} &::= \textbf{skip}
\end{aligned}
$$

FIGURE 4.2: Some syntactic sugars for λ_{Rust}.

- We use 0 and 1 as boolean values **false** and **true**, respectively. This allows us to define **if**-branching using **case**: if e_0 is **false**, then the expression with index 0 in the list $[e_2, e_1]$, which is e_2 is picked to be evaluated next; and if e_0 is **true**, then the expression with index 1, which is e_1, is picked.
- We suppress the access modes for reads and writes if they are **na**.
- **new** and **delete**, unlike **alloc** and **free**, never get stuck when the provided block size is 0. **new** simply just returns some location in that case.
- **memcpy** copies `len` cells from `src` to `dst` using the path operator. The "assign with length" notation ($e_1 :=_n {}^*e_2$) uses **memcpy**.
- There is no language primitive to define compound data structures. Instead, they can be implemented in memory using pointer arith-

$$\begin{aligned}
\textbf{letcont}\, k(\overline{x}) := e\, \textbf{in}\, e' &::= \textbf{let}\, k = (\textbf{rec}\, k(\overline{x}) := e)\, \textbf{in}\, e' \\
\textbf{jump}\, k(\overline{e}) &::= k(\overline{e}) \\
\textbf{funrec}\, f(\overline{x})\, \textbf{ret}\, k := e &::= \textbf{rec}\, f([k] +\!\!+ \overline{x}) := e \\
\textbf{call}\, f(\overline{e})\, \textbf{ret}\, k &::= f([k] +\!\!+ \overline{e})
\end{aligned}$$

FIGURE 4.3: CPS notations for λ_{Rust}.

metic. In particular, $e_1 :\overset{\text{inj}\, i}{=\!=} e_2$ simulates *tagged union* (sum types): the location at offset $e.0$ stores the active tag i (the case of the sum) while the location at offset $e.1$ stores the actual value evaluated from e_2. $e_1 :\overset{\text{inj}\, i}{=\!=}_n {}^* e_2$ is the notation for writing a tagged union whose values span multiple locations (memory cells).

- Finally, **`newlft`** and **`endlft`** are "ghost instructions" that have no interesting operational behaviors, and are only needed for logical soundness of the RustBelt type system (Part III).

Notation 4.4 (Continuation-Passing Style)**.** Programs in λ_{Rust} can be presented in continuation-passing style (CPS). This allows for encoding more complex control-flow constructs like labeled `break` or early **`return`**. This also makes λ_{Rust} closer to Rust's *Mid-level Intermediate Representation* (MIR)[5] than to surface Rust. The notations are given in Figure 4.3.

[5] Matsakis, *Introducing MIR* [Mat16].

- Continuations (meta-variable k) can be defined with $\textbf{letcont}\, k(\overline{x}) := e\, \textbf{in}\, e'$ where $\overline{x}$ is a list of binders that will be instantiated with the arguments $\overline{e}$ when a continuation is called with $\textbf{jump}\, k(\overline{e})$.
- CPS functions can be declared with $\textbf{funrec}\, f(\overline{x})\, \textbf{ret}\, k := e$ where f is the binder for a recursive function, $\overline{x}$ the list of binders for the arguments, and k the binder for the return continuation that will be called when the function returns. The return continuation takes only one argument for the return value. Accordingly, CPS functions can be called using $\textbf{call}\, f(\overline{e})\, \textbf{ret}\, k$ where $\overline{e}$ are the list of arguments and k the return continuation argument.

Note 4.5. Most syntactic sugars (Notation 4.3) and the CPS notations (Notation 4.4) are needed for the type system in RustBelt (Part III).

4.2 Language Expression Reductions

The complete semantics is defined by three sub semantics: the view-based machine semantics (§3.3), the race-detector semantics (§3.4), and the expressions semantics defined in this section. The complete semantics will be given in §4.3.

We first define some more auxiliary definitions.

Definition 4.6 (Readable Memory Value)**.** $\boxed{\omega \in \text{Readable}(\ell, \mathcal{M}, \mathcal{V})}$

The predicate $\text{Readable}(\ell, \mathcal{M}, \mathcal{V})$ defines the set of memory values *readable* from ℓ for the global memory $\mathcal{M}$ by a thread π whose thread-view is

currently $\mathcal{V}$.

$$\omega \in \text{Readable}(\ell, \mathcal{M}, \mathcal{V}) ::= \exists t.\, \mathcal{M}(\ell)(t) = (\omega, \_) \land t \leq \mathcal{V}.\mathsf{cur}(\ell)$$

That is, the thread π can only read a memory value ω that is in the memory for ℓ and is not mo-earlier than π's current view $\mathcal{V}$.cur for ℓ. This is the same condition as OM-READ (Definition 3.22), but additionally concerns values.

POINTER COMPARISON is a problem on its own,[6] especially for dead pointers. In this work, for simplicity, we follow the original RustBelt work and assume the most conservative choice that avoids UB:[7] unallocated pointers can non-deterministically be compared equal even though their representations are not.

[6] Kang et al., "A formal C memory model supporting integer-pointer casts" [Kan+15]; Memarian et al., "Into the depths of C: elaborating the de facto standards" [Mem+16]; Lee et al., "Reconciling high-level optimizations and low-level code in LLVM" [Lee+18]; Besson et al., "CompCertS: A Memory-Aware Verified C Compiler Using a Pointer as Integer Semantics" [BBW19]; Memarian et al., "Exploring C semantics and pointer provenance" [Mem+19]; Lepigre et al., "VIP: verifying real-world C idioms with integer-pointer casts" [Lep+22].

[7] UB is not a choice because pointer comparison is possible in safe Rust.

Definition 4.7 (Value Equality). $\boxed{\mathcal{M} \vdash v_1 = v_2}$

The results of *equality* comparison in $\mathcal{M}$ are defined by the following rules. (Recall that z is the meta-variable for integers.)

$$\mathcal{M} \vdash z = z \qquad \mathcal{M} \vdash \ell = \ell \qquad \frac{\ell_1 \in \text{unalloc}(\mathcal{M}) \lor \ell_2 \in \text{unalloc}(\mathcal{M})}{\mathcal{M} \vdash \ell_1 = \ell_2}$$

Definition 4.8 (Value Inequality). $\boxed{\vdash v_1 \neq v_2}$

The results of *inequality* comparison are defined by the following rules.

$$\frac{z_1 \neq z_2}{\vdash z_1 \neq z_2} \qquad \frac{\ell_1 \neq \ell_2}{\vdash \ell_1 \neq \ell_2} \qquad \vdash \ell \neq 0 \qquad \vdash 0 \neq \ell$$

That is, two values compare in-equal if their representations are different, and locations are never null (0). Note that this means that equality and inequality are not mutually exclusive for unallocated pointers.

Definition 4.9 (*Val* (non-UB) Comparability). $\boxed{\vdash v_1 =^? v_2}$

Two values are *comparable*, and thus may be compared equal and/or in-equal, if they satisfy the following rules.

$$\vdash z_1 =^? z_2 \qquad \vdash \ell_1 =^? \ell_2 \qquad \vdash \ell =^? 0 \qquad \vdash 0 =^? \ell$$

Definition 4.10 (λ_{Rust} Expression Reductions). $\boxed{\mathcal{M}, \mathcal{V} \vdash e \xrightarrow{\varepsilon^?} e_1', e_2'^?}$

The expression reduction relation $\mathcal{M}, \mathcal{V} \vdash e \xrightarrow{\varepsilon^?} e_1', e_2'^?$ says that under the global memory $\mathcal{M}$ and the thread-view $\mathcal{V}$, the expression e reduces in one step to e_1', potentially with an optional memory event $\varepsilon^?$ and an optional expression e_2' that will be running concurrently in a newly forked thread. Only memory operations will generate a memory event ε, and only $\textbf{fork}\{e_2'\}$ generates the new thread's expression e_2'. The rules for the expression reduction are given in Figure 4.4.

- OE-ECTX is the general rule that drives the evaluation strategy through evaluation contexts (see Definition 4.2). The remaining rules are the *primitive reductions* that only reduce in one step.

OE-ECTX
$$\frac{e \to e_1', e_2'^{?}}{\mathcal{M}, \mathcal{V} \vdash K[e] \to K[e_1'], e_2'^{?}}$$

OE-PROJ
$$\mathcal{M}, \mathcal{V} \vdash \ell.n \to \ell +_\ell n$$

OE-ADD
$$\frac{z_1 +_{\mathbb{Z}} z_2 = z'}{\mathcal{M}, \mathcal{V} \vdash z_1 + z_2 \to z'}$$

OE-SUB
$$\frac{z_1 -_{\mathbb{Z}} z_2 = z'}{\mathcal{M}, \mathcal{V} \vdash z_1 - z_2 \to z'}$$

OE-LE-TRUE
$$\frac{z_1 \leq_{\mathbb{Z}} z_2}{\mathcal{M}, \mathcal{V} \vdash z_1 \leq z_2 \to \texttt{true}}$$

OE-LE-FALSE
$$\frac{z_1 >_{\mathbb{Z}} z_2}{\mathcal{M}, \mathcal{V} \vdash z_1 \leq z_2 \to \texttt{false}}$$

OE-EQ-TRUE
$$\frac{\mathcal{M} \vdash v_1 = v_2}{\mathcal{M}, \mathcal{V} \vdash v_1 == v_2 \to \texttt{true}}$$

OE-EQ-FALSE
$$\frac{\vdash v_1 \neq v_2}{\mathcal{M}, \mathcal{V} \vdash v_1 == v_2 \to \texttt{false}}$$

OE-APP
$$\frac{|\overline{x}| = |\overline{v}|}{\mathcal{M}, \mathcal{V} \vdash (\texttt{rec}\, f(\overline{x}) := e)(\overline{v}) \to e[\texttt{rec}\, f(\overline{x}) := e/f, \overline{v}/\overline{x}]}$$

OE-CASE
$$\frac{0 \leq i < |\overline{e}|}{\mathcal{M}, \mathcal{V} \vdash \texttt{case}\, i\, \texttt{of}\, \overline{e} \to \overline{e}_i}$$

OE-FORK
$$\mathcal{M}, \mathcal{V} \vdash \texttt{fork}\{\, e \,\} \to \text{☣}, e$$

OE-READ
$$\mathcal{M}, \mathcal{V} \vdash {}^{*o}\ell \xrightarrow{\mathtt{R}^o(\ell, v)} v$$

OE-WRITE
$$\mathcal{M}, \mathcal{V} \vdash \ell :=_o v \xrightarrow{\mathtt{W}^o(\ell, v)} \text{☣}$$

OE-CAS-FAIL
$$\frac{\texttt{rlx} \sqsubseteq o_f, o_r, o_w \qquad \forall \omega \in \text{Readable}(\ell, \mathcal{M}, \mathcal{V}).\, \exists v'.\, \omega \equiv v' \wedge \vdash v_1 =^{?} v' \qquad \vdash v_1 \neq v_r}{\mathcal{M}, \mathcal{V} \vdash \texttt{CAS}^{o_f, o_r, o_w}(\ell, v_1, v_2) \xrightarrow{\mathtt{R}^{o_f}(\ell, v_r)} \texttt{false}}$$

OE-CAS-SUCC
$$\frac{\texttt{rlx} \sqsubseteq o_f, o_r, o_w \qquad \forall \omega \in \text{Readable}(\ell, \mathcal{M}, \mathcal{V}).\, \exists v'.\, \omega \equiv v' \wedge \vdash v_1 =^{?} v' \qquad \mathcal{M} \vdash v_1 = v_r}{\mathcal{M}, \mathcal{V} \vdash \texttt{CAS}^{o_f, o_r, o_w}(\ell, v_1, v_2) \xrightarrow{\mathtt{U}^{o_r, o_w}(\ell, v_r, v_2)} \texttt{true}}$$

OE-FENCE
$$\mathcal{M}, \mathcal{V} \vdash \texttt{fence}_o \xrightarrow{\mathtt{F}^o} \text{☣}$$

OE-ALLOC
$$\frac{n > 0}{\mathcal{M}, \mathcal{V} \vdash \texttt{alloc}(n) \xrightarrow{\mathtt{A}(\ell, n)} \ell}$$

OE-FREE
$$\frac{n > 0}{\mathcal{M}, \mathcal{V} \vdash \texttt{free}(n, \ell) \xrightarrow{\mathtt{D}(\ell, n)} \text{☣}}$$

FIGURE 4.4: Relaxed λ_{Rust} expression semantics.

- OE-PROJ says that the path operator simply computes a new location with offset $n \in \mathbb{Z}$ from ℓ, using the meta-level operator $+_\ell$ which is defined as $(i, n') +_\ell n = (i, n' +_{\mathbb{Z}} n)$. (Note that the offsets are integers, and use the integer operator $+_{\mathbb{Z}}$.)

- OE-ADD, OE-SUB, OE-LE-TRUE, and OE-LE-FALSE say that integer operators reduce according to the meta-level integer operators. Note that we have no reduction rules for poison (☣), which means that any computation using poison will get stuck. Also recall that **true** and **false** are just notations for 1 and 0, respectively.

- OE-EQ-TRUE and OE-EQ-FALSE say that comparison reduce according to equality and inequality comparisons, respectively.[8] This means that comparing unallocated locations can non-deterministically reduce to either **true** or **false**.

- OE-CASE says that a **case** block reduces if the *choice* index i is an actual index into the list of expressions. Then the expression $\overline{e}_i$ will

[8] see Definition 4.7 and Definition 4.8.

be picked to reduce. Note that no expression in the list $\overline{e}$ is reduced before the **case** is reduced.

- OE-APP says that function application reduces once all arguments have been evaluated to a list of values $\overline{v}$.[9] It is also required that the list of binders and the list of arguments have the same length. Then, the reduction *substitutes* the arguments for the binders, including the recursive function binder f, in the function's body.

[9] Recall that by the Definition 4.2 for evaluation contexts, the arguments are evaluated left-to-right.

- OE-FORK says that **fork**$\{\, e \,\}$ returns poison in the forking thread (so that the return value should not be used), and e will be used for the newly forked thread (see §4.3).

- OE-READ says that a read simply reduces to the value v that comes with the read event that $\mathtt{R}^o(\ell, v)$ it is tied to. The memory event will be used to match this reduction with the view-based machine (§3.3) and the race detector (§3.4) in the complete semantics (§4.3).

- OE-WRITE says that a write reduces to poison and is tied to the corresponding write event.

- A compare-and-swap instruction $\mathbf{CAS}^{o_f, o_r, o_w}(\ell, v_1, v_2)$ takes three *atomic* access modes: o_f is the order that will be used when the **CAS** fails, in which case it acts like a read with the mode o_f; otherwise, if the **CAS** succeeds, then it acts as both a read with the mode o_r and a write with the mode o_w. The **CAS** *atomically* (i) reads the location ℓ, (ii) compares the value read v_r with v_1, and (iii) if the values are equal, writes v_2 to ℓ. OE-CAS-FAIL and OE-CAS-SUCC therefore both require that for any memory value ω *readable*[10] by the **CAS**, its injected value v' must be comparable[11] with v_1.

  - In case of failure, OE-CAS-FAIL says that it must be that v_1 is in-equal to the read value v_r, and the reduction reduces to **false** and is tied to the read event $\mathtt{R}^{o_f}(\ell, v_r)$.
  - In case of success, OE-CAS-SUCC says that it must be that v_1 is equal to the read value v_r, and the reduction reduces to **true** and is tied to the update event $\mathtt{U}^{o_r, o_w}(\ell, v_r, v_2)$.

  Note again that this means that the **CAS** can non-deterministically fail or succeed if v_1 and what ℓ stores can be unallocated location values.

[10] *i.e.*, those values that are not yet overshadowed by $\mathcal{V}.\mathsf{cur}(\ell)$—see Definition 4.6.

[11] see Definition 4.9.

- OE-FENCE says that a fence also reduces to poison and is tied to the corresponding fence event.

- Finally, OE-ALLOC and OE-FREE both require that the provided block size n is positive, and respectively are tied to the memory events $\mathtt{A}(\ell, n)$ and $\mathtt{D}(\ell, n)$. The allocation call returns the base location ℓ of the newly allocated block, while the deallocation call returns poison.

Note 4.11. In the expression reductions, the global memory $\mathcal{M}$ is used in equality comparison (OE-EQ-TRUE). Other than that, it is only used

$$\mathsf{ForkView}(\mathcal{V}) ::= (\varnothing, \varnothing, \mathcal{V}.\mathsf{cur}, \mathcal{V}.\mathsf{cur})$$

OC-PURE
$$\frac{\mathcal{M}, \mathcal{V} \vdash e \to e', e_f^?}{(\mathcal{M}, \mathcal{N}) \mid (e, \mathcal{V}) \xrightarrow{\bot, (e_f, \mathsf{ForkView}(\mathcal{V}))^?}_{\mathsf{t}} (\mathcal{M}, \mathcal{N}) \mid (e', \mathcal{V})}$$

OC-MEM
$$\frac{\forall \varepsilon'. (\mathcal{M}, \mathcal{V} \vdash e \xrightarrow{\varepsilon'} \_, \bot) \land (\mathcal{M} \mid \mathcal{V} \xrightarrow{\varepsilon', \_, \_} \_ \mid \_) \implies \mathcal{M}, \mathcal{N}, \mathcal{V} \vdash \mathsf{RaceFree}(\varepsilon') \quad \mathcal{M}, \mathcal{V} \vdash e \xrightarrow{\varepsilon} e', \bot \quad \mathcal{M} \mid \mathcal{V} \xrightarrow{\varepsilon, r^?, ms} \mathcal{M}' \mid \mathcal{V}' \quad \mathcal{N} \xrightarrow{\varepsilon, r^?, ms} \mathcal{N}'}{(\mathcal{M}, \mathcal{N}) \mid (e, \mathcal{V}) \xrightarrow{\varepsilon, \bot}_{\mathsf{t}} (\mathcal{M}', \mathcal{N}') \mid (e', \mathcal{V}')}$$

FIGURE 4.5: The combined 1-thread semantics of ORC11 machine semantics and λ_{Rust} expression semantics.

together with the thread-view $\mathcal{V}$ for comparison in **CAS** (OE-CAS-FAIL and OE-CAS-SUCC).

4.3 The Complete Operational Semantics of Relaxed λ_{Rust}

Definition 4.12 (1-Thread Reductions). $\varsigma \mid (e, \mathcal{V}) \xrightarrow{\varepsilon^?, (e_f, \mathcal{V}_f)^?}_{\mathsf{t}} \varsigma' \mid (e', \mathcal{V}')$

The combined *1-thread* (single-thread) reductions of ORC11 machine semantics and λ_{Rust} expression semantics is given in Figure 4.5. The configuration $(\mathcal{M}, \mathcal{N}) \mid (e, \mathcal{V})$ is called a 1-thread configuration which includes the thread's executing expression e, the thread-view $\mathcal{V}$, and the global state $\mathcal{M}$ and $\mathcal{N}$. The pair $(e, \mathcal{V})$ is also called a *thread-local configuration*. The combined 1-thread reductions define how a 1-thread configuration is transformed in one reduction step, possibly generating an optional memory event ε and an optional pair of expression and thread-view $(e_f, \mathcal{V}_f)$ for a newly forked thread.

- OC-PURE allows for a pure step that only reduces the expression of the configuration, and potentially generates an expression e_f for a new thread. In that case, the thread-view $\mathsf{ForkView}(\mathcal{V})$, derived from $\mathcal{V}$, is picked for the newly forked thread. The definition of $\mathsf{ForkView}(\mathcal{V})$ encodes our choice for **fork**'s behaviors with respect to synchronization: the forked (child) thread should be synchronized with the forking thread, but a **fork** does not act as a release fence for the forked thread, so its release views are empty ($\varnothing$).

- OC-MEM allows for a memory step that simultaneously (i) reduces the expression e in one step to e' ($\mathcal{M}, \mathcal{V} \vdash e \xrightarrow{\varepsilon} e', \bot$, Definition 4.10) with a memory event ε, and (ii) makes a view-based machine step ($\mathcal{M} \mid \mathcal{V} \xrightarrow{\varepsilon, r^?, ms} \mathcal{M}' \mid \mathcal{V}'$, Definition 3.22) and a race detector step ($\mathcal{N} \xrightarrow{\varepsilon, r^?, ms} \mathcal{N}'$, Definition 3.26) with the *same* memory event ε, potentially with an optional read action id $r^?$ and a list of write messages ms. The reduction needs to be race-free, *i.e.*, for any potential memory step that the current configuration can make and thus generate an event ε', it must be that $\mathcal{M}, \mathcal{N}, \mathcal{V} \vdash \mathsf{RaceFree}(\varepsilon')$ (Definition 3.25).

OT-STEP

$$\frac{\mathcal{T}(\pi) = (e, \mathcal{V}) \qquad (\mathcal{M}, \mathcal{N}) \mid (e, \mathcal{V}) \xrightarrow{\varepsilon^?, (e_f, \mathcal{V}_f)^?}_{\mathtt{t}} (\mathcal{M}', \mathcal{N}') \mid (e', \mathcal{V}')}{(\mathcal{M}, \mathcal{N}) \mid \mathcal{T} \to (\mathcal{M}', \mathcal{N}') \mid \mathcal{T}[\pi \leftarrow (e', \mathcal{V}')]\big[\rho \leftarrow (e_f, \mathcal{V}_f) \,\big|\, (e_f, \mathcal{V}_f)^? \neq \bot \wedge \rho \notin \mathtt{dom}(\mathcal{T})\big]}$$

FIGURE 4.6: Threadpool semantics.

Finally, we can lift the 1-thread semantics to the complete, concurrent semantics with a thread-pool.

Definition 4.13 (Thread-pools)**.** A thread-pool $\mathcal{T}$ is a partial, finite map from thread-ids to pairs of expressions and thread-views, *i.e.*,

$$\mathcal{T} \in \mathit{ThreadPool} ::= \mathit{Thread} \xrightarrow{\text{fin}} (\mathit{Expr} \times \mathit{ThreadView})$$

Definition 4.14 (Threadpool Reductions)**.** $\boxed{\varsigma \mid \mathcal{T} \to \varsigma' \mid \mathcal{T}'}$

The thread-pool semantics is given in Figure 4.6. It defines the reduction of a *thread-pool configuration* $\varsigma \mid \mathcal{T}$ that includes the global machine state (Definition 3.14) and the thread-pool for all threads. OT-STEP says that a thread-pool reduction just picks some random thread π in the thread-pool and make a 1-thread step using the 1-thread configuration $\varsigma \mid (e, \mathcal{V})$ for the global state and thread π. The results of the 1-thread step are then used to update the thread-pool configuration. In case thread π forks a new thread with $(e_f, \mathcal{V}_f)$, then a fresh thread-id $\rho \notin \mathtt{dom}(\mathcal{T})$ is picked to insert the newly forked thread into the thread-pool.

CHAPTER SUMMARY. This chapter presents the λ_{Rust} language and explains how to combine it with the machine semantics of ORC11 to achieve our target language. In the next chapter, we instantiate Iris with this language to obtain a vanilla separation logic for RMC. Note that Iris takes as input the 1-thread reductions (Definition 4.12) of a language and defines its own thread-pool reductions. We only state the thread-pool reductions (Definition 4.14) for completeness, which is similar to (but simpler than) that of Iris.

5

Related Work

Podkopaev et al.[1] develop an operational account of a subset of C11 that includes relaxed accesses and non-atomics. However, it lacks support for fences and thus could not be used as is to build a logic to verify libraries that use fences *e.g.*, Rust's `Arc`. Their semantics also does not forbid the data race in Example 3.29.

[1] Podkopaev et al., "Operational Aspects of C/C++ Concurrency" [PSN16].

Doherty et al.[2] develop an operational semantics based on event graphs for the release/acquire/relaxed fragment of RC11. They also develop an invariant-based logic geared towards automated verification for programs in that fragment. Their operational semantics supports neither non-atomics nor fences.

[2] Doherty et al., "Verifying C11 programs operationally" [Doh+19].

Kang et al.'s *promising semantics*[3] proposes using *promises* to fix C11's out-of-thin-air problem without prohibiting load-store reordering on relaxed accesses, as RC11 and ORC11 do. The promising semantics as well as its versions 2.0 and 2.1,[4] however, does not include non-atomic accesses, as the authors did not consider the problem of undefined behaviors (UBs). Instead, the semantics has plain accesses that can race among one another and still have defined semantics.

[3] Kang et al., "A promising semantics for relaxed-memory concurrency" [Kan+17].

[4] Lee et al., "Promising 2.0: global optimizations in relaxed memory concurrency" [Lee+20]; Cho et al., "Modular data-race-freedom guarantees in the promising semantics" [Cho+21b].

Recently, Cho et al.[5] extends version 2.1 (PS2.1) with UB-inducing non-atomics (dubbed PS^{na}) in order to verify optimizations on non-atomics across atomics. Non-atomics in PS^{na} can also be promised, and write-write races with a non-atomic access invokes UB, while read-write races results in the read of undef (which is more defined than the poison ☣ value).[6] The race detection mechanism of PS^{na} is similar to that of ORC11: a non-atomic access is racy if the thread is not locally aware of another access to the same location with a bigger timestamp (similar to DRF-READ-NA and DRF-WRITE-NA); and an atomic access is racy if the thread is not aware of another non-atomic access to the same location with a bigger timestamp (similar to DRF-READ-AT, DRF-WRITE-AT and DRF-UPDATE). Interestingly, the implementation of PS^{na}'s race detector is much simpler: instead of using a global non-atomic view $\mathcal{N}$ as in ORC11, PS^{na} distinguishes between atomic and non-atomic messages in the global memory $\mathcal{M}$, and the check for races simply looks at the timestamps and the messages' types. We believe that ORC11 can benefit from such a simplification.

[5] Cho et al., "Sequential reasoning for optimizing compilers under weak memory concurrency" [Cho+22].

[6] Lee et al., "Taming undefined behavior in LLVM" [Lee+17].

The follow-up work of PS^{na} by Lee et al.[7] proposes a two-layered semantics: a source language semantics *without* promises, and an intermediate representation (IR) semantics that can have promises (dubbed

[7] Lee et al., "Putting Weak Memory in Order via a Promising Intermediate Representation" [Lee+23].

PS^{IR}) but *only for race-detection purpose*. The paper also comes with a proposal to hardware developers to justify the semantics of PS^{IR}. If the proposal is accepted, it mean that iRC11 can be applied to most C and C++ RMC programs. As of now, iRC11 can only be used for promise-free programs (a property which can be checked externally), and the verification results can be ported to the full semantics using the DRF theorems by Cho et al.[8].

Meanwhile, semantics for low-level languages, such as that of the ARM ISA,[9] do not employ UBs. Regardless, their modern semantics include hardware optimizations such as speculative execution and multiple layers of caches, resulting in complex memory models. It remains to be seen if the view-based approach can be used to model various hardware features—there have been view-based semantics for non-volatile memory,[10] but, to the best of our knowledge, not for instruction caches[11] nor virtual memory.[12]

[8] Cho et al., "Sequential reasoning for optimizing compilers under weak memory concurrency" [Cho+22].

[9] [Pul+18; Flu+17; Pul+19; Sim+20].

[10] Cho et al., "Revamping hardware persistency models: view-based and axiomatic persistency models for Intel-x86 and Armv8" [Cho+21a].

[11] Simner et al., "ARMv8-A System Semantics: Instruction Fetch in Relaxed Architectures" [Sim+20].

[12] Simner et al., "Relaxed virtual memory in Armv8-A" [Sim+22].

Part II

SEPARATION LOGIC FOR RELAXED MEMORY

This part discusses the features and the construction of iRC11. We give a brief review of the Iris separation logic framework in Chapter 6 and discuss the instantiation of Iris with ORC11 in Chapter 7, which results in the *base logic* for ORC11. The base logic, however, is very close to the operational semantics, and only provides basic separation reasoning principles. Chapter 8, following iGPS,[13] presents the first abstraction layer that gives rise to the iRC11 logic: *view-monotone predicates*, or vProp for short. The chapter also presents several RMC-specific modalities of iRC11 in vProp, some of which are inspired by FSL and Cosmo.[14] Chapter 9 and Chapter 10 present the construction for the core ownership assertions of iRC11: the *non-atomic* and *atomic points-to*. Chapter 11 introduces *invariants*—the standard principle for concurrently sharing resources—but with RMC-specific limitations. Finally, Chapter 12 ends this part with several simple example verifications of RMC programs and libraries using iRC11. The bottom half of Figure 1.1 visualizes the dependency among these chapters.

[13] Kaiser et al., "Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris" [Kai+17].

[14] Doko and Vafeiadis, "A Program Logic for C11 Memory Fences" [DV16]; Mével et al., "Cosmo: a concurrent separation logic for multicore OCaml" [MJP20].

6

More Background: Iris, A Framework for Concurrent Separation Logics

In this chapter, we give a quick review of the concurrent separation logic framework Iris.[1] Readers who are familiar with Iris can skip this review, and jump to Chapter 7 for the instantiation of Iris with our relaxed λ_{Rust} + ORC11 language. On the other hand, for readers who prefer a deep dive into the details of Iris, please consult the journal paper [Jun+18b].

Iris as a framework contains (i) a higher-order, resource-aware, step-indexing *base logic* with bunched implications (BI),[2] (ii) extensions[3] that support program verification (*i.e.*, program logics with *weakest preconditions* and *impredicative invariants*) for an input language with an operational interleaving semantics, and (iii) a general Iris Proof Mode (IPM)[4] that supports interactive resource reasoning and that can be instantiated with any BI logics.

An excerpt of Iris grammar is given in Figure 6.1. Propositions in Iris belong to the type iProp, which has a *step-indexing* model on *resources*.[5]

Concept 6.1 (Iris Base Logic)**.** The Iris base logic supports the common logical connectives ($\mathsf{False}, \mathsf{True}, \Rightarrow, \wedge, \vee$). iProp propositions are to be interpreted with resources in mind, so the "classical" conjunction $P \wedge Q$ should be read as P and Q hold relying on the *same* resources. The base logic logic allows embedding *pure* facts ϕ which exist at the meta-level and which naturally do not occupy resources.

- It is a BI logic: the *separating* conjunction $P * Q$ says that P and Q hold on *disjoint* resources, and the *wand* implication $P \mathrel{-\!\!*} Q$ holds on some resource r that can be combined with some resource s where P holds to obtain the resource $r \cdot s$ where Q holds.
- The base logic also supports *higher-order* logical quantification ($\forall, \exists$) and recursive predicates (μ) because the quantified variable x can also be an iProp. Recursive predicates need to be *guarded*: occurrences of x in the body need to be under a *later* modality $\triangleright$.
- The later modality is the materialization of the step-indexing model in the logic: $\triangleright P$ intuitively means that P holds in the *next* step, so P only becomes available until the program takes a step (and so decrements the step-index). Step-indexing guarantees the logic's soundness in the presence of recursive higher-order quantifications, impredicative invariants, and higher-order ghost state.

[1] Jung et al., "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning" [Jun+15]; Jung et al., "Higher-order ghost state" [Jun+16]; Krebbers et al., "The Essence of Higher-Order Concurrent Separation Logic" [Kre+17]; Jung et al., "Iris from the ground up: A modular foundation for higher-order concurrent separation logic" [Jun+18b].

[2] O'Hearn and Pym, "The logic of bunched implications" [OP99]; Ishtiaq and O'Hearn, "BI as an Assertion Language for Mutable Data Structures" [IO01].

[3] derived from the base logic

[4] Krebbers et al., "Interactive Proofs in Higher-Order Concurrent Separation Logic" [KTB17]; Krebbers et al., "MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic" [Kre+18].

[5] [Jun+18b], §4.

$$
\begin{aligned}
P \in \mathsf{iProp} ::= {} & \text{(* base logic *)} \\
& \mid \phi \mid \mathsf{False} \mid \mathsf{True} \mid P \Rightarrow Q \mid P \wedge Q \mid P \vee Q \mid P * Q \mid P \mathrel{-\!\!*} Q \mid \exists x.\, P \mid \forall x.\, P \mid \mu\, x.\, P \\
& \mid \triangleright P \mid \Box P \mid \boxed{a : M}^{\gamma} \mid \Rightarrow\!\!\!\!\!\!\cdot\; P \mid \ldots \\
& \text{(* program logic *)} \\
& \mid \boxed{P}^{\mathcal{N}} \mid {}^{\mathcal{E}_1}\!\Rightarrow\!\!\!\!\!\!\cdot^{\mathcal{E}_2} P \mid \mathsf{wp}_{\mathcal{E}}\, e\, \{v.\, Q\} \mid \ldots
\end{aligned}
$$

FIGURE 6.1: An excerpt of Iris grammar.

- The *persistent* modality $\Box P$ says that P is known to hold without some *exclusive* resource, so that P can be freely duplicated, *i.e.*, $P \Rightarrow P * P$.
- The proposition $\boxed{a : M}^{\gamma}$ asserts the ownership of an element a of a *resource algebra* M for the *ghost location* γ. In case the resource algebra M is clear in context, we simply write $\boxed{a}^{\gamma}$.
- The *basic update* modality $\Rightarrow\!\!\!\!\!\!\cdot\; P$ hides away some *ghost updates* that can be performed to achieve P.

Concept 6.2 (Iris Program Logic)**.** To support program verification for a language, several constructions can be derived from the base logic:[6]

- $\boxed{P}^{\mathcal{N}}$ asserts the existence of some global *invariant* that holds the resource P, under some *invariant name* that is in the *namespace* $\mathcal{N}$. Namespaces provide some hierarchy to sets of invariant names.
- The *fancy update* ${}^{\mathcal{E}_1}\!\Rightarrow\!\!\!\!\!\!\cdot^{\mathcal{E}_2} P$ hides away some logical updates (including ghost updates) that can be performed to achieve P. The logical updates involve accessing (opening and closing) invariants and therefore trading resources with the involved invariants. The *masks* $\mathcal{E}_1$ and $\mathcal{E}_2$ are sets of invariant names that identify the invariants that hold (unopened) *before* and *after* the update, respectively.
- The *weakest pre-condition* $\mathsf{wp}_{\mathcal{E}}\, e\, \{v.\, Q\}$ assert the resources needed for e to safely execute[7] and maintain the invariants in $\mathcal{E}$ at every step, and *if* e terminates to a value v, then the post-condition Q holds. Note that the "if" signals that the program logic by default only guarantee *partial*, not *total* correctness.

[6] Not all connectives in the Iris base logic are primitives, some are also derived.

[7] never get stuck

6.1 Basic Rules

Figure 6.2 provides several basic rules of many connectives in the Iris base logic. The *logical entailment* $P \vdash Q$ says that Q is derivable from P using the rules of the logic. Intuitively, its interpretation is that for any resource and step-index where P holds, Q should also hold. The notation $P \dashv\vdash Q$ stands for $P \vdash Q$ and $Q \vdash P$.

The rules in Figure 6.2 are very general rules that apply to all Iris propositions. For example, they include commutativity, associativity, and distributivity among connectives and modalities; PERS-ELIM tells us that

$$\mathsf{True} * P \dashv\vdash P \qquad P * Q \dashv\vdash Q * P \qquad (P * Q) * R \dashv\vdash P * (Q * R)$$

$$\textsc{Sep-mono}\ \frac{P_1 \vdash Q_1 \qquad P_2 \vdash Q_2}{P_1 * P_2 \vdash Q_1 * Q_2} \qquad \textsc{Wand-intro-elim}\ \frac{P * Q \vdash R}{P \vdash Q \mathrel{-\!\!*} R}$$

Several rules for the persistent modality.

$$\frac{P \vdash Q}{\Box P \vdash \Box Q} \qquad \textsc{Pers-idemp}\ \Box\Box P \dashv\vdash \Box P \qquad \textsc{Pers-elim}\ \Box P \vdash P$$

$$\Box P \wedge \Box Q \dashv\vdash \Box P * \Box Q \qquad \Box(P * Q) \dashv\vdash \Box P * \Box Q$$

$$\Box \exists x.\, P \dashv\vdash \exists x.\, \Box P \qquad \Box \forall x.\, P \vdash \forall x.\, \Box P$$

Several rules for the later modality.

$$\textsc{Later-intro}\ P \vdash \triangleright P \qquad \textsc{Later-mono}\ \frac{P \vdash Q}{\triangleright P \vdash \triangleright Q} \qquad \textsc{Löb}\ (\triangleright P \Rightarrow P) \vdash P$$

$$\triangleright(P * Q) \dashv\vdash \triangleright P * \triangleright Q \qquad \Box \triangleright P \dashv\vdash \triangleright \Box P$$

$$\frac{\tau \text{ inhabited}}{\triangleright \exists x : \tau.\, P \dashv\vdash \exists x : \tau.\, \triangleright P \qquad \triangleright \forall x.\, P \dashv\vdash \forall x.\, \triangleright P}$$

FIGURE 6.2: Basic rules of several Iris connectives.

we can always get P from $\Box P$; or LATER-INTRO says that if we have P now, we also have P in the next step; or LÖB allows us to do Löb induction: if having P in the next step is sufficient to achieve P now, then we can have P *now*.

6.2 Ghost State and Resource Algebras

The ghost ownership assertion $\boxed{a : M}^{\gamma}$ requires a to be an element of a resource algebra M. Resource algebras give the separation structure for ghost state in separation logics, and are a generalization of Iris for *partial commutative monoids* (PCMs).

Definition 6.3 (Resource Algebras)**.** A resource algebra (RA) is a tuple $(M, \mathsf{valid} : M \to \mathsf{Prop}, |-| : M \to M^?, (\cdot) : M \times M \to M)$ where the type M has a commutative, associative *composition* $(\cdot)$; a *core* function $(|-|)$ that computes a optional *per-element* unit (in $M^?$) for each element $a \in M$; and a *validity* predicate (valid) to indicate legal compositions.

Compared to PCMs, RAs force composition to be total, and instead regain partiality with validity. Furthermore, where a PCM has a single unit element ε, an RA can have multiple per-element units $|a|$, and some elements may not have a unit, *i.e.*, $|a| = \bot$.

The following properties must hold for an RA.

$$\forall a, b.\, a \cdot b = b \cdot a \qquad \text{(RA-COMM)}$$

$$\forall a, b, c.\, (a \cdot b) \cdot c = a \cdot (b \cdot c) \qquad \text{(RA-ASSOC)}$$

$$\forall a.\, |a| \in M \Rightarrow |a| \cdot a = a \qquad \text{(RA-CORE-ID)}$$

$$\forall a.\, |a| \in M \Rightarrow ||a|| = a \qquad \text{(RA-CORE-IDEMP)}$$

$$\forall a.\, |a| \in M \wedge a \preccurlyeq b \Rightarrow |b| \in M \wedge |a| \preccurlyeq |b| \qquad \text{(RA-CORE-MONO)}$$

$$\forall a, b.\, \mathsf{valid}(a \cdot b) \Rightarrow \mathsf{valid}(a) \qquad \text{(RA-VALID-OP)}$$

where $\quad a^? \cdot \bot = \bot \cdot a^? = a^?$

$$a \preccurlyeq b ::= \exists c \in M.\, b = a \cdot c \qquad \text{(RA-INCL)}$$

GHOST-ALLOC
$$\frac{\text{valid}(a)}{\dot{\Rightarrow}\exists\gamma.\boxed{a}^{\gamma}}$$

GHOST-VALID
$$\boxed{a}^{\gamma} \Rightarrow \text{valid}(a)$$

GHOST-OP
$$\boxed{a \cdot b}^{\gamma} \Leftrightarrow \boxed{a}^{\gamma} * \boxed{b}^{\gamma}$$

GHOST-UPDATE-GEN
$$\frac{a \rightsquigarrow B}{\boxed{a}^{\gamma} \dot{\Rightarrow} \exists b \in B.\boxed{b}^{\gamma}}$$

GHOST-UPDATE
$$\frac{a \rightsquigarrow b}{\boxed{a}^{\gamma} \dot{\Rightarrow} \boxed{b}^{\gamma}}$$

BUPD-MONO
$$\frac{P \vdash Q}{\dot{\Rightarrow} P \vdash \dot{\Rightarrow} Q}$$

BUPD-INTRO
$$P \vdash \dot{\Rightarrow} P$$

BUPD-TRANS
$$\dot{\Rightarrow}\dot{\Rightarrow} P \vdash \dot{\Rightarrow} P$$

BUPD-FRAME
$$Q * \dot{\Rightarrow} P \vdash \dot{\Rightarrow}(Q * P)$$

where $P \dot{\Rightarrow} Q ::= \Box(P \mathrel{-\!\!*} \dot{\Rightarrow} Q)$.

FIGURE 6.3: Basic rules of Iris ghost ownership and basic updates.

RA-CORE-ID says that if a core exists for some a, then it is a unit for a, and RA-CORE-MONO says that the core function maintains *inclusion*, defined by RA-INCL using composition. RA-VALID-OP says that if a composition is considered valid, all of its components must also be valid.

A *unital* RA (uRA) has a unit element ε satisfying:

$$\text{valid}(\varepsilon) \qquad \forall a \in M.\, \varepsilon \cdot a = a \qquad |\varepsilon| = \varepsilon$$

The rules for ghost ownership are given in Figure 6.3. GHOST-ALLOC lets us allocate a fresh ghost location γ with an initial, valid element that is a. Under the hood, this is a ghost update to the global *ghost heap* to insert the fresh ghost location γ, and this update is hidden in the basic update modality $\dot{\Rightarrow}$. GHOST-VALID says that ghost ownership maintains validity. More importantly, GHOST-OP says that the RA composition gives the separation structure to ghost ownership.

Notation 6.4 (Basic Viewshifts). $\boxed{P \dot{\Rightarrow} Q}$

Intuitively, *basic viewshift* $P \dot{\Rightarrow} Q$ says that the resources owned by P can be turned into the resources owned by Q using some ghost updates.

$$P \dot{\Rightarrow} Q ::= \Box(P \mathrel{-\!\!*} \dot{\Rightarrow} Q)$$

GHOST-UPDATE-GEN and GHOST-UPDATE allow us to update some ghost state, using the basic viewshift. GHOST-UPDATE-GEN allows for an non-deterministic update: some element in b will be picked after the update. To maintain consistency of separation, the premises of these rules require that ghost updates are *frame-preserving*.

Concept 6.5 (Frame-preserving Ghost Updates). $\boxed{a \rightsquigarrow b}$

When doing a ghost update for $\boxed{a}^{\gamma}$, one must remember that one only has a *part* of the ghost location γ: with separation (*i.e.*, using GHOST-OP), other parts of γ that are compatible with a, called the *frame*, are owned by other parties. As such, for any b that we update a to, one must maintain that a is also compatible with the frame, *i.e.*, one cannot allow b composed with the frame being *invalid*, leading to inconsistency in the logic (turning a valid state into an invalid one). The relation $a \rightsquigarrow b$ encodes the fact that the update from a to b maintains validity of the whole ghost state, *i.e.*, it is a frame-preserving update.

$a \rightsquigarrow b$ is derived from the more general definition $a \rightsquigarrow B$ which says that a can be frame-preservingly updated to any element in B:

$$a \rightsquigarrow B ::= \forall c^? \in M^?.\, \text{valid}(a \cdot c^?) \Rightarrow \exists b \in B.\, \text{valid}(b \cdot c^?) \qquad \text{(RA-FRAME-UPD-GEN)}$$

$$a \rightsquigarrow b ::= a \rightsquigarrow \{b\} \qquad \text{(RA-FRAME-UPD)}$$

In this definition, "the frame" is universally quantified as c.

Figure 6.3 also provides some structural rules for basic updates.

6.3 Invariants and Fancy Updates

Invariants can be seen as logical, global spaces where resources can be stored for concurrent accesses.[8] The catch is that accesses must be *(physically) atomic*—take place during a single step of computation—and invariants must be re-established after each access, so that they indeed hold *invariantly* (*i.e.*, after each step). As such, invariants are used to build concurrent protocols on pieces of shared state, *i.e.*, to constrain how clients can change them.

[8] They are indeed implemented in Iris as a global chunk of resources, see ([Jun+18b], §7.1).

The construction of Iris invariants, however, is not tied to the notion of atomic expression. Instead, it uses *invariant namespaces* and *masks* to track which invariants are opened (being accessed), and only subsequently tie weakest pre-conditions to masks to enforce atomicity. We will see that in §6.7. Here, we look at the vanilla rules of Iris invariants.

The proposition $\boxed{I}^{\mathcal{N}}$ asserts the existence of I in the global invariant space with some invariant name ι in the namespace $\mathcal{N}$.[9] Several rules for Iris invariants are given in Figure 6.4, which rely on *fancy updates*, which in turn generalize basic updates with masks. Intuitively, ${}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P$ represents ownership of resources such that, assuming that the invariants in $\mathcal{E}_1$ are *enabled* (they are not opened) before, one can perform frame-preserving updates and afterwards obtaining P and having the invariants in $\mathcal{E}_2$ enabled. As such, if the masks $\mathcal{E}_1$ and $\mathcal{E}_2$ are different, some invariants may have been opened to achieve P, or some other resources must have been returned to close some invariants. Furthermore, masks prevent reentrancy: an invariant cannot be opened again without being closed first.

[9] The meta-variable I is preferred over P to indicate resources that are stored in invariants.

This intuition is on display in the invariant access rule INV-ACC:

- If we know that the set of enabled invariants $\mathcal{E}_1$ includes the namespace $\mathcal{N}$—meaning that the invariants in $\mathcal{N}$ are not opened yet, then we can open all invariants in $\mathcal{N}$ with the fancy update ${}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}\setminus\mathcal{N}}$ (so after that only the invariants in ${}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}\setminus\mathcal{N}}$ are enabled).

- Furthermore, if we know that $\boxed{I}^{\mathcal{N}}$, *i.e.*, I is stored in one of those invariants in $\mathcal{N}$ that are to be opened, then we get access to the resources of I, but under a *later* modality ($\triangleright I$). The later ensures that I is *guarded*, because invariants are impredicative, *e.g.*, I can refer to $\boxed{I}^{\mathcal{N}}$ itself.

INV-ALLOC
$$\triangleright I \vdash \Rightarrow_{\mathcal{E}} \boxed{I}^{\mathcal{N}}$$

INV-ACC
$$\frac{\mathcal{N} \subseteq \mathcal{E}}{\boxed{I}^{\mathcal{N}} \vdash {}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}\setminus\mathcal{N}} \left(\triangleright I * (\triangleright I \;{}^{\mathcal{E}\setminus\mathcal{N}}\!\Rightarrow\!\!\!*^{\mathcal{E}} \mathsf{True})\right)}$$

FUPD-BUPD
$$\dot{\Rightarrow} P \vdash \Rightarrow_{\mathcal{E}} P$$

FUPD-MONO
$$\frac{P \vdash Q}{{}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} Q}$$

FUPD-INTRO
$$P \vdash \Rightarrow_{\mathcal{E}} P$$

FUPD-INTRO-MASK
$$\frac{\mathcal{E}_2 \subseteq \mathcal{E}_1}{\mathsf{True} \vdash {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} {}^{\mathcal{E}_2}\!\Rightarrow^{\mathcal{E}_1} \mathsf{True}}$$

FUPD-FRAME
$$Q * {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} P \vdash {}^{\mathcal{E}_1 \uplus \mathcal{E}_f}\!\Rightarrow^{\mathcal{E}_2 \uplus \mathcal{E}_f} (Q * P)$$

FUPD-TRANS
$${}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_2} {}^{\mathcal{E}_2}\!\Rightarrow^{\mathcal{E}_3} P \vdash {}^{\mathcal{E}_1}\!\Rightarrow^{\mathcal{E}_3} P$$

where $\Rightarrow_{\mathcal{E}} ::= {}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}}$ and $P \;{}^{\mathcal{E}}\!\Rightarrow\!\!\!*^{\mathcal{E}'} Q ::= P \mathrel{-\!\!*} {}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}'} Q$.

FIGURE 6.4: Some rules for Iris invariants and fancy updates.

- Finally, we also get a "closing" update, $(\triangleright I \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}\!\Rightarrow^{\mathcal{E}} \mathsf{True})$, which allows us to return the invariant resources, also under a later, $\triangleright I$, we can close all invariants and re-establish that $\mathcal{E}$ is enabled.

Note that even though all invariants in $\mathcal{N}$ are opened during the access, we only take out $\triangleright I$ and so we only need to return $\triangleright I$. The resources of other invariants are untouched and are kept safe under the "closing" update. However, this means that the access rule does not support accessing two invariants stored under the same namespace together. To access two invariants I_1 and I_2 together, we need to allocate them in two *disjoint* namespaces $\mathcal{N}_1 \# \mathcal{N}_2 \subseteq \mathcal{E}$. Then we can apply INV-ACC twice, first with $\boxed{I_1}^{\mathcal{N}_1}$ and $\mathcal{E}$, then with $\boxed{I_2}^{\mathcal{N}_2}$ and $\mathcal{E} \setminus \mathcal{N}_2$, to get access to $\triangleright I_1 * \triangleright I_2$.

INV-ALLOC allows us to allocate an invariant I with some fresh invariant name ι picked non-deterministically from $\mathcal{N}$. Note that we only need to provide I under a later, and that $\Rightarrow_{\mathcal{E}}$ is a notation for a fancy update that does not change masks. This rule justifies the use of namespaces: if we had use only invariant names, then when accessing invariants we would have to deal with disjointness for names which are allocated dynamically. Instead, in this setup with namespaces, we only have to deal with disjointness of namespaces which can be picked statically. Note that this also means that both masks and namespaces need to contain an infinite number of names.

Figure 6.4 also provides some structural rules for fancy updates. Importantly, FUPD-BUPD says that a fancy update includes a basic update.

Notation 6.6 (Fancy Updates and Wand Viewshifts). $\Rightarrow_{\mathcal{E}}$ denotes non-mask-changing fancy updates ${}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}}$, and $P \;{}^{\mathcal{E}}\!\Rightarrow\!\!\!*^{\mathcal{E}'} Q$ denotes *wand viewshifts*, a combination of wand implication and fancy update:

$$P \;{}^{\mathcal{E}}\!\Rightarrow\!\!\!*^{\mathcal{E}'} Q ::= P \mathrel{-\!\!*} {}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}'} Q$$

$P \Rightarrow\!\!\!*_{\mathcal{E}} Q$ denotes non-mask-changing wand viewshifts.

Plain viewshifts are the persistent version of wand viewshifts:

$$P \;{}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}'} Q ::= \Box(P \mathrel{-\!\!*} {}^{\mathcal{E}}\!\Rightarrow^{\mathcal{E}'} Q)$$
$$P \Rightarrow_{\mathcal{E}} Q ::= \Box(P \mathrel{-\!\!*} \Rightarrow_{\mathcal{E}} Q)$$

6.4 Hoare Triples

Once we instantiate Iris with a language that has an interleaving operational semantics defined in the style of evaluation contexts, the Iris problem logic provides us a notion of *weakest pre-conditions* propositions that encode partial correctness of expressions. (We will see the instantiation of the relaxed λ_{Rust} language in Chapter 7.) Intuitively, the proposition $\mathsf{wp}_{\mathcal{E}}\, e\, \{v.\, Q\}$ asserts the ownership of some resources with which e can execute safely (*i.e.*, it never gets stuck) while maintaining the invariants in $\mathcal{E}$ at every step, and if e evaluates to a value v, then we arrive at some resources at which the *post-condition* Q holds.

The goal of program verification for some program e is to prove that the weakest pre-condition with some suitable target post-condition is derivable from some sufficient resources—the *pre-condition*—using the rules of the program logic. This can be seen more concretely in the Iris definition of *Hoare triples*.

Definition 6.7 (Iris Hoare Triples)**.** Hoare triples in Iris are defined in terms of weakest pre-conditions:

$$\{P\}\, e\, \{v.\, Q\}_{\mathcal{E}} ::= \Box\,(P \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, e\, \{v.\, Q\})$$

The persistent modality ($\Box$) guarantees that the precondition P is sufficient to prove the weakest pre-condition—that is, the wand does not need extra resources. The intuitive interpretation of Hoare triples is straightforward: the precondition P is sufficient for the expression e to safely execute while maintaining the invariants in $\mathcal{E}$, and if e terminates to v then Q holds.

6.5 Adequacy

Weakest pre-conditions and Hoare triples are also Iris propositions (iProp), and are in fact also defined entirely in the logic of Iris to encode the aforementioned intuition. However, once we have proven a weakest pre-condition or a Hoare triple for some program, we would like to achieve a guarantee *outside of the logic* that the program executes safely under the target operational semantics. This is called *adequacy*: for each instantiation of Iris with some *language* Λ, we need to prove once and for all the following Theorem.

Theorem 6.8 (Iris Adequacy)**.** *If* $\vdash \mathsf{wp}_{\top}\, e\, \{v.\, \phi(v)\}$ *is derivable in the Iris program logic for* Λ *where* $\phi(v)$ *is a pure (meta-level) fact, then the following holds.*

$$\begin{aligned}
&\forall \pi, \mathcal{T}, \sigma.\, ([\pi \mapsto e], \sigma_{\mathrm{init}}) \to^*_{\Lambda} (\mathcal{T}, \sigma) \Rightarrow \\
&\qquad \forall v.\, \mathcal{T}(\pi) = v \Rightarrow \phi(v) && \text{(IRIS-ADEQUACY-VAL)} \\
&\wedge \quad \forall \rho, e_\rho.\, \mathcal{T}(\rho) = e_\rho \Rightarrow \big(e_\rho \textit{ is a value} \vee \mathrm{red}(e_\rho, \sigma)\big) \\
& && \text{(IRIS-ADEQUACY-NO-STUCK)}
\end{aligned}$$

That is, if we start running e with the initial global configuration $([\pi \mapsto e], \sigma_{\mathrm{init}})$—where the threadpool only has a single thread π with

the expression e and the initial global state σ_{init} is typically empty—then for any configuration reachable $(\mathcal{T}, \sigma)$ through the reflexive, transitive closure of the threadpool reduction $\rightarrow_\Lambda$ for Λ, (IRIS-ADEQUACY-VAL) if thread π has reduces to a value v, the pure fact $\phi(v)$ must hold, and (IRIS-ADEQUACY-NO-STUCK) for any thread ρ with the expression e_ρ, it is not stuck: either e_ρ is a value, or e_ρ is still reducible on σ ($\mathrm{red}(e_\rho, \sigma)$).

Note that the mask of the weakest pre-condition is $\top$, which means that all invariants (any allocated) hold at every step. Note also that deriving $\vdash \mathsf{wp}_\top\, e\, \{v.\, \phi(v)\}$ is the same as deriving a Hoare triple with a trivial pre-condition, *i.e.*, $\vdash \{\mathsf{True}\}\, e\, \{v.\, Q\}_\top$.

6.6 Some Common Rules for WPs and Hoare Triples

On the other hand, during program verification, in order to derive $\mathsf{wp}_\mathcal{E}\, e\, \{v.\, Q\}$, one relies on the various rules for weakest pre-conditions (WP). Many of those rules are language-specific: after instantiate Iris with our target language Λ, we need to extend the logic with WP rules for the primitive instructions of Λ. We will see those primitive rules for our relaxed λ_{Rust} + ORC11 language in Chapter 7. Here, we look at some WP rules that are not so language-specific.

More concretely, these are the typical rules we would expect for a lambda-calculus based language with an operational semantics using evaluation contexts and fork-based concurrency. These rules, as well as their corresponding derived Hoare-triple versions, are given in Figure 6.5. Note that when reading the WP rules, by reading them *backward*—as typically when we use a rule of form $\mathsf{wp}_\mathcal{E}\, e'\, \{\Psi\} \vdash \mathsf{wp}_\mathcal{E}\, e\, \{\Phi\}$ to turn the verification goal $\mathsf{wp}_\mathcal{E}\, e\, \{\Phi\}$ into the goal $\mathsf{wp}_\mathcal{E}\, e'\, \{\Psi\}$, we are driving the symbolic execution of the program *forward*: the expression e reduces to the expression e'. Note that we also use the meta-variables Φ and Ψ for predicates on values (*Val* $\rightarrow$ iProp) which can be used for post-conditions (which so far have been written as $v.\, Q$).

- WP-VAL says that if the expression has reached a value v, then we simply have to prove the post-condition $\Phi(v)$. HOARE-VAL is derivable from WP-VAL with $\Phi ::= \lambda w.\, w = v$.

- WP-MONO allows for monotonicity for the post-conditions: to prove a WP with the post-condition Ψ we may want to prove a WP with the stronger post-condition Φ. It also additionally provides monotonicity for masks: if we can prove a WP relying on fewer invariants (with the smaller mask $\mathcal{E}_1$), then that WP also works with more invariants (with the bigger mask $\mathcal{E}_2$).[10] The well-known consequence rule HOARE-CONS is derivable from WP-MONO.

- WP-BIND is the core rule to drive sequential composition. It makes use of evaluation contexts. To prove a WP for $K[e]$, we prove a WP for e whose post-condition is another WP that plugs the resulting v into the "continuation" K. This corresponds to executing e first and then the continuation K. HOARE-BIND is derivable from WP-BIND

[10] So that from a $\mathsf{wp}_\mathcal{E}\, e\, \{\Phi\}$ one can, for example, obtain $\mathsf{wp}_\top\, e\, \{\Phi\}$ with the top ($\top$) mask, and subsequently apply adequacy.

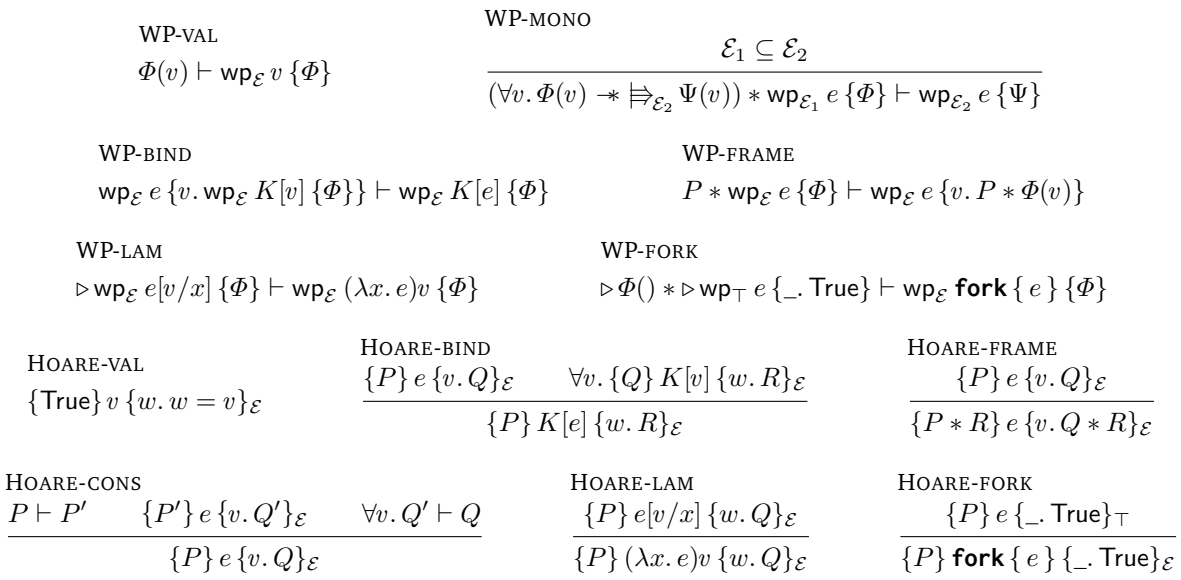


FIGURE 6.5: Some common rules for Iris weakest pre-conditions and Hoare triples.

and WP-MONO. Note that Hoare-triple rules should be read as the separating conjunction of the premises implies the conclusion.

- WP-FRAME allows for "framing": if the post-condition requires us to prove P that has nothing to do with the execution of e, then we can frame P out and separately prove the WP for e with the remaining post-condition Φ. The famous frame rule HOARE-FRAME is derivable from WP-FRAME.
- WP-LAM is a primitive rule for a language with beta-reduction. The rule says that we need to prove the WP for the expression after beta-reduction. However, because the reduction takes a step, we only need to prove the new WP under a later. HOARE-LAM is derivable from WP-LAM and LATER-INTRO.
- WP-FORK is a primitive rule for a language with fork-based concurrency. We need to prove, only under a later because the reduction takes a step, (i) the current post-condition $\Phi()$ for the current thread (assuming the returned value of **fork** is unit), and (ii) a WP for the newly forked thread ρ (with the expression e) with a trivial post-condition, which signifies that forked threads are "detached" by the default.[11] The rule HOARE-FORK is derivable from WP-FORK, and in that rule we can see that some resources P can be transferred from the forking thread to the forked thread.

[11] Iris, however, does support picking a fixed, non-trivial post-condition for all forked threads. But such a post-condition only restricts the set of possible final states of the forked threads. If one wants to communicate a post-condition Q back to the forking thread, however, one can implement a join operation (using an extra location) to wait for the thread ρ to receive Q—see §12.3.

6.7 Weakest Pre-conditions and Invariants

Figure 6.6 provides some rules for the interaction between WPs (Hoare triples) and invariants and fancy updates.

- WP-FUPD says that we can perform fancy updates around an expression if our goal is a WP. Combining this rule with FUPD-MONO

WP-FUPD
$$\Rrightarrow_{\mathcal{E}} \mathsf{wp}_{\mathcal{E}}\, e\, \{v.\, \Rrightarrow_{\mathcal{E}} \Phi(v)\} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$$

WP-ATOMIC
$$\frac{\mathsf{atomic(e)}}{{}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2} \mathsf{wp}_{\mathcal{E}_2}\, e\, \left\{v.\, {}^{\mathcal{E}_2}\Rrightarrow^{\mathcal{E}_1} \Phi(v)\right\} \vdash \mathsf{wp}_{\mathcal{E}_1}\, e\, \{\Phi\}}$$

WP-INV
$$\frac{\triangleright I \vdash \mathsf{wp}_{\mathcal{E}\setminus\mathcal{N}}\, e\, \{v.\, \triangleright I * \Phi(v)\} \qquad \mathsf{atomic(e)} \qquad \mathcal{N} \subseteq \mathcal{E}}{\boxed{I}^{\mathcal{N}} \vdash \mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}}$$

HOARE-VS
$$\frac{P \Rrightarrow_{\mathcal{E}} P' \qquad \{P'\}\, e\, \{v.\, Q'\}_{\mathcal{E}} \qquad \forall v.\, Q' \Rrightarrow_{\mathcal{E}} Q}{\{P\}\, e\, \{v.\, Q\}_{\mathcal{E}}}$$

HOARE-INV
$$\frac{\{\triangleright I * P\}\, e\, \{v.\, \triangleright I * Q\}_{\mathcal{E}\setminus\mathcal{N}} \qquad \mathsf{atomic(e)} \qquad \mathcal{N} \subseteq \mathcal{E}}{\boxed{I}^{\mathcal{N}} \vdash \{P\}\, e\, \{v.\, Q\}_{\mathcal{E}}}$$

where $P \Rrightarrow_{\mathcal{E}} Q ::= \square(P \mathrel{-\!\!*} \Rrightarrow_{\mathcal{E}} Q)$ (see Notation 6.6).

FIGURE 6.6: Some rules for Iris weakest pre-conditions and invariants.

and FUPD-FRAME (Figure 6.4), around a WP we can *eliminate* any hypotheses with fancy updates in our proof context if they have masks smaller than $\mathcal{E}$. With FUPD-BUPD, we can additionally perform ghost updates, and with FUPD-INTRO-MASK and INV-ACC, we can also open invariants and close them immediately to obtain some duplicable *knowledge*. Furthermore, HOARE-CONS can be strengthened to HOARE-VS.

- WP-ATOMIC allows us to perform *mask-changing* updates around *(physically) atomic* instructions—those whose execution takes place in a single step of computation. If e is atomic, then to prove $\mathsf{wp}_{\mathcal{E}_1}\, e\, \{\Phi\}$, we can perform a mask-changing fancy update from the mask $\mathcal{E}_1$ to the $\mathcal{E}_2$, and prove a WP for the mask $\mathcal{E}_2$ whose post-condition must perform a reverse mask-changing update from $\mathcal{E}_2$ back to $\mathcal{E}_1$ before establishing the original post-condition.

- WP-INV allows us to open invariants around atomic instructions, and it is derivable from WP-ATOMIC. If we let $\mathcal{E}_1 = \mathcal{E}$, and $\mathcal{E}_2 = \mathcal{E}\setminus\mathcal{N}$, we can use WP-ATOMIC and then INV-ACC to open the invariant $\boxed{I}^{\mathcal{N}}$ around a goal of $\mathsf{wp}_{\mathcal{E}}\, e\, \{\Phi\}$, and use $\triangleright I$ for the execution of the atomic expression e, if we can re-establish $\triangleright I$ after the step. HOARE-INV is easily derivable from WP-INV.

6.8 Properties of Propositions

There are two important properties of propositions that make proofs in step-indexing separation logics more convenient. Several rules for these two properties are given in Figure 6.7.

Property 6.9 (Timeless Propositions)**.** *Timeless* propositions are those who are independent of the step index, and thus are not affected by the later modality. Concretely TIMELESS-DISCRETE allows them to be used immediately—we say that the later is *stripped off*—using a fancy update. Timelessness is maintained structurally in many cases, such

$$\textsc{Timeless-vs}\ \frac{\mathsf{timeless}(P)}{\triangleright P \vdash \Rrightarrow_{\mathcal{E}} P} \qquad \textsc{Timeless-discrete}\ \frac{M \text{ is discrete}}{\mathsf{timeless}(\boxed{a : M}^{\gamma})} \qquad \textsc{Timeless-pure}\ \mathsf{timeless}(\phi)$$

$$\textsc{Persistent-dup}\ \frac{\mathsf{persistent}(P)}{P \Leftrightarrow P * P} \qquad \textsc{Persistent-pers}\ \mathsf{persistent}(\Box P) \qquad \textsc{Persistent-pure}\ \mathsf{persistent}(\phi) \qquad \textsc{Persistent-core}\ \frac{|a| = a}{\mathsf{persistent}(\boxed{a}^{\gamma})} \qquad \textsc{Persistent-inv}\ \mathsf{persistent}(\boxed{I}^{\mathcal{N}})$$

$$\textsc{Timeless-sep}\ \frac{\mathsf{timeless}(P) \qquad \mathsf{timeless}(Q)}{\mathsf{timeless}(P * Q)} \qquad \textsc{Timeless-pers}\ \frac{\mathsf{timeless}(P)}{\mathsf{timeless}(\Box P)} \qquad \textsc{Persistent-sep}\ \frac{\mathsf{persistent}(P) \qquad \mathsf{persistent}(Q)}{\mathsf{persistent}(P * Q)} \qquad \textsc{Persistent-later}\ \frac{\mathsf{persistent}(P)}{\mathsf{persistent}(\triangleright P)}$$

FIGURE 6.7: Some properties of timeless propositions and persistent propositions.

as in TIMELESS-SEP. Some typical timeless propositions are pure (meta-level) facts (TIMELESS-PURE), and ownership of a ghost element whose RA is discrete (TIMELESS-DISCRETE), *i.e.*, the equivalence relation is not step-indexed.[12]

[12] A step-indexing family of equivalence relations is needed for higher-order ghost state.

Property 6.10 (Persistent Propositions)**.** Intuitively, *persistent* propositions are those who do not consume resources. As such, they can be freely duplicated and be used many times without being consumed, as in PERSISTENT-DUP. For this reason, persistent propositions are often called *knowledge*, in contrast to non-persistent ones, which are generally called resources. Persistency is maintained structurally, for example, as in PERSISTENT-SEP and PERSISTENT-LATER. Naturally, $\Box P$ is persistent (PERSISTENT-PERS). Some typical persistent propositions are pure facts (PERSISTENT-PURE), the knowledge of some invariant (PERSISTENT-INV), or ghost ownership of some core element (PERSISTENT-CORE).

6.9 The Method of Fictional Separation

When instantiating Iris with our target language, apart from the expected WP rules mentioned above, we need to derive WP rules for our language's primitives (*e.g.*, for reads and writes). These derivations, as well as most derivations of rules that we will see in later chapters, all follow the method of *fictional separation*.[13] It is a method to turn the ownership of some monolithic, non-splittable resource r into separable ones. The splitting of r is fictional: we construct some RA that mirrors r and that has a suitable composition to provide the desirable separation structure. In other words, we create a *ghost* copy of r and we split the copy, while maintaining that the copy is in sync with the original r. For this, we need the *authoritative* RA.

[13] Jensen and Birkedal, "Fictional Separation Logic" [JB12].

Definition 6.11 (The Authoritative Resource Algebra)**.** The authoritative RA,[14] denoted $\textsc{Auth}(M)$, assumes a *unital* RA M, and provides two types of elements for some element $a \in M$: the *authoritative* element $\bullet\, a$, and the *fragmentary* element $\circ\, a$. The elements satisfy (but are not limited to) the rules given in Figure 6.8.

[14] Jung et al., "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning" [Jun+15].

$$\forall a.\, |\circ a| = \circ |a| \qquad \text{(AUTH-FRAG-CORE)}$$
$$\forall a, b.\, \circ (a \cdot b) = \circ a \cdot \circ b \qquad \text{(AUTH-FRAG-OP)}$$
$$\forall a, b.\, a \preccurlyeq b \Rightarrow \circ a \preccurlyeq \circ b \qquad \text{(AUTH-FRAG-MONO)}$$
$$\forall a.\, \mathsf{valid}(\bullet a) \Leftrightarrow \mathsf{valid}(a) \qquad \text{(AUTH-AUTH-VALID)}$$
$$\forall a.\, \mathsf{valid}(\circ a) \Leftrightarrow \mathsf{valid}(a) \qquad \text{(AUTH-FRAG-VALID)}$$
$$\forall a, b.\, \neg\mathsf{valid}(\bullet a \cdot \bullet b) \qquad \text{(AUTH-AUTH-OP-VALID)}$$
$$\forall a, b.\, \mathsf{valid}(\bullet a \cdot \circ b) \Leftrightarrow b \preccurlyeq a \wedge \mathsf{valid}(a) \qquad \text{(AUTH-BOTH-VALID)}$$
$$\forall a_1, b_1, a_2, b_2.\, (a_1, b_1) \rightsquigarrow_{\mathfrak{L}} (a_2, b_2) \Rightarrow \bullet a_1 \cdot \circ b_1 \rightsquigarrow \bullet a_2 \cdot \circ b_2 \qquad \text{(AUTH-UPDATE)}$$

FIGURE 6.8: Several rules for the AUTH(M) RA.

That is, fragmentary elements preserve core, composition and thus inclusion of M (AUTH-FRAG-CORE, AUTH-FRAG-OP, and AUTH-FRAG-MONO); both types of elements preserve validity (AUTH-AUTH-VALID and AUTH-FRAG-VALID), and the authoritative element is *exclusive* (AUTH-AUTH-OP-VALID). Most importantly, a valid composition of the authoritative element of a and a fragmentary element of b implies that b is included in a (AUTH-BOTH-VALID). This is why the RA is named "authoritative": the authoritative element includes all fragmentary elements.

We then can use the authoritative element as *the* ghost copy for our monolithic resource r, and the fragments as its splittable counterparts.

Concept 6.12 (The Method of Fictional Separation)**. To fictionally separate a monolithic, non-splittable resource r:**

1. Design an RA M that mirrors the original resource r and that has the desirable separation structure *i.e.*, an appropriate composition.
2. Apply the authoritative RA to M, *i.e.*, AUTH(M), and keep the ownership of the authoritative part $\bullet\, r$ in sync with r.
3. Use the fragmentary parts $\circ$ to build local assertions.
4. Derive rules for local assertions that update the splittable fragmentary parts $\circ$, in conjunction with $\bullet\, r$ and thus with r, but having r and $\bullet\, r$ *hidden*, typically by using an invariant.

To update the fragmentary elements together with the authoritative one, we use AUTH-UPDATE, which says that $\bullet\, a_1 \cdot \circ\, b_1$ can frame-preservingly updated to $\bullet\, a_2 \cdot \circ\, b_2$ if (a_1, b_1) can be *locally updated* to (a_2, b_2).

Definition 6.13 (Local Updates)**.** $\boxed{(a_1, b_1) \rightsquigarrow_{\mathfrak{L}} (a_2, b_2)}$

A pair (a_1, b_1) can be locally updated to a pair (a_2, b_2), if for any (optional) frame $a_{\mathrm{f}}^?$ completing b_1 to a_1, $a_{\mathrm{f}}^?$ also complete b_2 to a_2:

$$(a_1, b_1) \rightsquigarrow_{\mathfrak{L}} (a_2, b_2) ::= \\ \forall a_{\mathrm{f}}^?.\, \mathsf{valid}(a_1) \wedge a_1 = b_1 \cdot a_{\mathrm{f}}^? \Rightarrow \mathsf{valid}(a_2) \wedge a_2 = b_2 \cdot a_{\mathrm{f}}^?$$

In the case of AUTH-UPDATE, this means that the frame $a_{\mathrm{f}}^?$ is the fragmentary frame of $\circ\, b_1$, and when updating $\bullet\, a_1$ together with $\circ\, b_1$, we need to maintain that the update respects $a_{\mathrm{f}}^?$.

Now, Iris allows use to fictionally separate the physical machine state of our target language through the *physical state interpretation*.

6.10 The Physical State Interpretation

In fact, Iris requires us—the instantiator—to provide the state interpretation predicate $S : \mathit{State} \to \mathsf{iProp}$ where *State* is the type of the global physical state. This predicate is used in the definition of Iris weakest pre-conditions.

Definition 6.14 (Iris WP, simplified)**.**

$$
\begin{aligned}
&\mathsf{wp}^{S}_{\mathcal{E}}\, e\, \{\Phi\} ::= \\
&\quad e \in \mathit{Val} \wedge \Rightarrow_{\mathcal{E}} \Phi(e) \\
&\vee \Big(e \notin \mathit{Val} \wedge \forall \sigma.\, S(\sigma) \mathrel{-\!\!*} \\
&\qquad {}^{\mathcal{E}}\Rightarrow^{\varnothing} \big(\mathrm{red}(e,\sigma) * \forall e', \sigma', e_{\mathrm{f}}.\, (e,\sigma) \to_{\mathrm{t}} (e', \sigma', e_{\mathrm{f}}) \mathrel{-\!\!*} \\
&\qquad\quad \triangleright\, {}^{\varnothing}\Rightarrow^{\mathcal{E}} (S(\sigma') * \mathsf{wp}^{S}_{\mathcal{E}}\, e'\, \{\Phi\} * \mathsf{wp}^{S}_{\top}\, e_{\mathrm{f}}\, \{v.\, \mathsf{True}\})\big)\Big)
\end{aligned}
$$

Weakest pre-condition is defined as a recursive iProp predicate (guarded by a later modality), with two cases. In case the expression e is already a value, then the post-condition $\Phi(e)$ must hold. Otherwise, assuming $S(\sigma)$ for the current global physical state σ *before* a step, (i) e must be safe to take a 1-thread reduction step ($\to_{\mathrm{t}}$) in σ, *i.e.*, e is reducible in σ ($\mathrm{red}(e,\sigma)$), and (ii) for any resulting configuration $(e', \sigma', e_{\mathrm{f}})$ of such a 1-thread reduction step, the state interpretation $S(\sigma')$ for the physical state σ' *after* the step must hold, and the weakest pre-conditions hold recursively for e' and the forked expression e_{f}. The fancy updates enable ghost updates and invariant accesses around a single reduction step.

From this definition, we see that the proposition $S(\sigma)$ is needed to perform a step and must be re-established afterwards, where $\sigma \in \mathit{State}$ is the current physical state of the machine. As such, we can use S, whose definition is up to *us* (the instantiator) to pick, to fictionally separate the physical state σ. Specifically, we will use S to keep the current state σ in sync with our authoritative ghost copy $\bullet\, \sigma$, and give out the fragmentary element $\circ\, \sigma$—which can be separated into smaller elements—to define our local assertions. Conveniently through S, the WP definition not only keeps the physical state and the ghost copy in sync for us, but also hides them away, so our only remaining tasks are to define a suitable RA that enable the desirable separation on *State* and to prove our primitive WP rules (which will need to update the physical state, thus the authoritative ghost copy and the corresponding ghost fragments).

6.11 An Instantiation Example for Simple Heaps

To make it more concrete, we briefly consider an instantiation example for an SC language whose physical state is simple heap—a map from locations to values, *i.e.*, $\mathit{State} ::= \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Val}$.[15] Let us call this language

[15] This example is rephrased from [Jun+18b] and [Kai+17].

λ_{HEAP}. What we want is to derive the *small-footprint* rules for reads and writes, using the local *points-to* assertion.

HEAP-READ
$\{\ell \mapsto v\}\, {}^*\ell\, \{w.\, w = v * \ell \mapsto v\}_\top$

HEAP-WRITE
$\{\ell \mapsto v\}\, \ell := v'\, \{().\, \ell \mapsto v'\}_\top$

To do so, we pick a suitable RA to split a heap σ into multiple *singleton* heaps of the form $[\ell \leftarrow v]$, which can then be used to define $\ell \mapsto v$. This RA is called SHEAP, whose type is exactly *State* and composition is union of finite maps, but composition is only valid between disjoint maps. That is, $\text{valid}(\sigma \cdot \sigma') \Leftrightarrow \mathtt{dom}(\sigma) \cap \mathtt{dom}(\sigma') = \varnothing$. The state interpretation and the points-to assertion are then defined as:

$$S(\sigma) ::= \boxed{\bullet\, \sigma : \text{AUTH}(\text{SHEAP})}^{\gamma_{\text{HEAP}}}$$
$$\ell \mapsto v ::= \boxed{\circ\, [\ell \leftarrow v] : \text{AUTH}(\text{SHEAP})}^{\gamma_{\text{HEAP}}}$$

That is, the state interpretation $S(\sigma)$ is the ghost ownership of the authoritative heap $\bullet\, \sigma$ at the ghost location γ_{HEAP}, and the points-to assertion $\ell \mapsto v$ is the ghost ownership of the fragmentary singleton heap $\circ\, [\ell \leftarrow v]$ at the same ghost location γ_{HEAP}. The ghost location γ_{HEAP} is allocated before the program runs (a proof that indeed needs to be done in Adequacy—see §6.5), and is needed to establish the agreement between $\ell \mapsto v$ and the current physical state σ, indirectly through S. To see this in action, let us look at the proofs of HEAP-READ and HEAP-WRITE. Both proofs first proceed by unfolding the definitions of Hoare triples (§6.4, Definition 6.7), and WP (Definition 6.14).

Proof sketch of HEAP-READ. After introducing the assumptions and clearing the fancy update (using FUPD-INTRO-MASK and FUPD-MONO),[16] we arrive at the following goal.

[16] see §6.7

Context:	Goal:
$\ell \mapsto v$ $(= \boxed{\circ\, [\ell \leftarrow v]}^{\gamma_{\text{HEAP}}})$	
$S(\sigma)$ $(= \boxed{\bullet\, \sigma}^{\gamma_{\text{HEAP}}})$	$\text{red}({}^*\ell, \sigma) \land \forall e', \sigma', e_\text{f}.\ \ldots$

We first need to show that e is reducible on σ.

With GHOST-OP and GHOST-VALID (see §6.2), from our assumptions,

we have $\text{valid}(\bullet\, \sigma \cdot \circ\, [\ell \leftarrow v])$, and then by AUTH-BOTH-VALID,

we have $[\ell \leftarrow v] \preccurlyeq \sigma$ and $\text{valid}(\sigma)$, so $\sigma = [\ell \leftarrow v] \uplus \sigma_\text{f}$ for some σ_f.

By the definition of SHEAP's composition, we then know $\sigma(\ell) = v$.

Since $\ell \in \mathtt{dom}(\sigma)$, we can show $\text{red}({}^*\ell, \sigma)$. Our remaining goal is

$\ell \mapsto v * S(\sigma)$	$\forall e', \sigma', e_\text{f}.\, ({}^*\ell, \sigma) \to_\mathtt{t} (e', \sigma', e_\text{f}) \mathrel{-\!\!*} \triangleright\, {}^{\varnothing}\!\Rrightarrow^{\top} \ldots$

Looking at the operational semantics of $({}^*\ell, \sigma) \to_\mathtt{t} (e', \sigma', e_\text{f})$,

we learn that $e' = \sigma(\ell) \land \sigma' = \sigma \land e_\text{f} = \bot$, so our goal is

$\ell \mapsto v * S(\sigma)$	$\triangleright\, {}^{\varnothing}\!\Rrightarrow^{\top} S(\sigma) * \mathsf{wp}^{S}_{\top}\, \sigma(\ell)\, \{w.\, w = v * \ell \mapsto v\}$

After clearing the later and fancy update modalities and simplifying

(using LATER-INTRO and FUPD-MONO again),

$\ell \mapsto v * S(\sigma)$	$S(\sigma) * \sigma(\ell) = v * \ell \mapsto v$

Since we already know $\sigma(\ell) = v$, we are done. □

Note that to apply GHOST-OP in the above proof, it is important that $S(\sigma)$ and $\ell \mapsto v$ use the same ghost location γ_{HEAP}.

Proof sketch of HEAP-WRITE. Similar to the proof of HEAP-WRITE, by owning $\ell \mapsto v$ we know that $\sigma(\ell) = v$ where σ is the current physical state, so we can prove $\mathrm{red}(\ell := v', \sigma)$. We then introduce all assumptions and clear the fancy update and the later from the goal. However, we additionally need to use FUPD-TRANS to keep a fancy update $\Rightarrow_\top$: we need to update our ghost copy so that it is in sync with the new physical state after the step. Our goal then looks as follow.

Context:	Goal:
$\ell \mapsto v * S(\sigma)$	
$(\ell := v', \sigma) \to_{\mathtt{t}} (e', \sigma', e_{\mathrm{f}})$	$\Rightarrow_\top S(\sigma') * \mathsf{wp}^S_\top\, e'\, \{(). \ell \mapsto v'\} * \ldots$

From the operational semantics of $\to_{\mathtt{t}}$, we learn that $e' = () \land \sigma' = \sigma[\ell \leftarrow v'] \land e_{\mathrm{f}} = \bot$, so our goal is

$\ell \mapsto v * S(\sigma)$	$\Rightarrow_\top S(\sigma') * \mathsf{wp}^S_\top\, ()\, \{(). \ell \mapsto v'\}$

After simplifying, we arrive at

$\ell \mapsto v * S(\sigma)$	$\Rightarrow_\top S(\sigma') * \ell \mapsto v'$

This goal pins down to an update of our ghost copy to sync with the new state σ'. Indeed, after applying GHOST-OP, then FUPD-BUPD, and then GHOST-UPDATE, we have to prove the following frame-preserving update.

$$\bullet\, \sigma \cdot \circ\, [\ell \leftarrow v] \rightsquigarrow \bullet\, \sigma' \cdot \circ\, [\ell \leftarrow v']$$

Applying AUTH-UPDATE, we have to show

$$(\sigma, [\ell \leftarrow v]) \rightsquigarrow_{\mathfrak{L}} (\sigma', [\ell \leftarrow v'])$$

This is easy. We know that the frame σ_{f} that completes σ with $[\ell \leftarrow v]$ is disjoint from $[\ell \leftarrow v]$: $\sigma = \sigma_{\mathrm{f}} \uplus [\ell \leftarrow v]$. Thus we can show

$$(\sigma_{\mathrm{f}} \uplus [\ell \leftarrow v]\,, [\ell \leftarrow v]) \rightsquigarrow_{\mathfrak{L}} (\sigma_{\mathrm{f}} \uplus [\ell \leftarrow v']\,, [\ell \leftarrow v'])$$

easily by looking at the definition of local updates (Definition 6.13). We are done because $\sigma' = \sigma[\ell \leftarrow v'] = \sigma_{\mathrm{f}} \uplus [\ell \leftarrow v']$. □

7

A Base Logic for RMC in Iris

In this chapter, we demonstrate how to instantiate the Iris framework with our λ_{Rust} + ORC11 semantics (defined in Chapter 4) to achieve a "vanilla" relaxed-memory CSL. Even though vanilla, this so-called *base logic* for our target language is already quite expressive, because it is derived from the Iris program logic: it is a higher-order CSL with higher-order ghost state, impredicative invariants, and admits the WP and Hoare rules listed in Chapter 6. In this chapter, we establish more WP and Hoare rules for our language's relaxed memory primitives. These rules form the bottom-most basis of our logic, from which all other constructs of the higher-level iRC11 logic will be derived.

ROADMAP. §6.11 already gives an instantiation example for a language with simple heaps, but it worths articulating the process, more specifically for our λ_{Rust} + ORC11 semantics:

1. We instantiate Iris with the 1-thread reductions (Definition 4.12) of λ_{Rust} + ORC11. Since we are in a relaxed memory semantics with views, the resulting base logic will have to expose them. §7.1 discusses how views generally show up in our base-logic WP and Hoare triple rules.
2. In contrast to an SC logic whose main local assertion is points-to, we need new local assertions with appropriate separation structure to handle relaxed effects and data races. §7.2 discusses the design (interfaces) of those new assertions.
3. §7.3 presents the desired small-footprint, primitive rules that use the new local assertions.
4. §7.4 presents the resource algebras needed to give a model for the new local assertions and proves their properties.
5. Finally, §7.5 defines the state interpretation S for our language, and §7.6 provides proofs of some primitive rules as well as adequacy.

7.1 Thread-local Configurations as Expressions

The Iris framework requires as input a language with a reduction relation $(e, \sigma) \to_{\mathsf{t}} (e', \sigma', e_{\mathsf{f}})$—which we call a 1-thread reduction—where e is the expression of the thread being evaluated and σ is the physical machine

BL-WP-BIND
$$\mathsf{wp}_{\mathcal{E}}\,(e, \mathcal{V})\,\{(v, \mathcal{V}_v).\, \mathsf{wp}_{\mathcal{E}}\,(K[v], \mathcal{V}_v)\,\{\Phi\}\} \vdash \mathsf{wp}_{\mathcal{E}}\,(K[e], \mathcal{V})\,\{\Phi\}$$

BL-WP-FRAME
$$P * \mathsf{wp}_{\mathcal{E}}\,(e, \mathcal{V})\,\{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\,(e, \mathcal{V})\,\{(v, \mathcal{V}_v).\, P * \Phi(v, \mathcal{V}_v)\}$$

BL-WP-VAL
$$\Phi(v, \mathcal{V}) \vdash \mathsf{wp}_{\mathcal{E}}\,(v, \mathcal{V})\,\{\Phi\}$$

BL-WP-LAM
$$\triangleright \mathsf{wp}_{\mathcal{E}}\,(e[v/x], \mathcal{V})\,\{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\,((\lambda x.\, e)v, \mathcal{V})\,\{\Phi\}$$

BL-WP-FORK
$$\triangleright \Phi(☣, \mathcal{V}) * \triangleright \mathsf{wp}_{\top}\,(e, \mathrm{ForkView}(\mathcal{V}))\,\{\_.\,\mathsf{True}\} \vdash \mathsf{wp}_{\mathcal{E}}\,(\mathbf{fork}\,\{\,e\,\}, \mathcal{V})\,\{\Phi\}$$

BL-WP-PLUS
$$\triangleright \mathsf{wp}_{\mathcal{E}}\,(n +_{\mathbb{Z}} m, \mathcal{V})\,\{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\,(n + m, \mathcal{V})\,\{\Phi\}$$

BL-WP-IF
$$\triangleright \mathsf{wp}_{\mathcal{E}}\,((b)\,?\,e_1 : e_2, \mathcal{V})\,\{\Phi\} \vdash \mathsf{wp}_{\mathcal{E}}\,(\mathbf{if}\,b\,\mathbf{then}\,e_1\,\mathbf{else}\,e_2, \mathcal{V})\,\{\Phi\}$$

FIGURE 7.1: Pure primitive WPs in the RMC base logic

[1] In fact, Iris supports forking multiple threads, so e_f can be a list of expressions.

state. The resulting e' is the thread's new expression, and σ' the new physical state, and a new thread may be forked with the expression e_f.[1] Satisfying this requirement of a 1-thread reduction is rather straightforward for traditional SC languages, but for our RMC language, we need a little bit more care: the execution of a thread needs not only the globally-shared physical state, but also some thread-local state—specifically in this case, a thread-view. This can be seen clearly in our 1-thread reduction relation $\varsigma \mid (e, \mathcal{V}) \xrightarrow{\varepsilon^?, (e_f, \mathcal{V}_f)^?}_{\mathsf{t}} \varsigma' \mid (e', \mathcal{V}')$ (Definition 4.12).

Aside from some notation mismatches, we need to fit our 1-thread reduction relation to what Iris requires. The solution is simple: we instantiate what Iris considers "expressions" with pairs of expressions and thread-views $(e, \mathcal{V})$—our thread-local configurations. This is only a change of perspective: what Iris really requires is a 1-thread relation that describes how a thread-local configuration reduces together with the global state, but Iris has often been instantiated with languages where the thread-local configuration is just an expression. This perspective applies to languages with *thread-local state*, which in our case is a thread-view, but in other languages can be, for example, a call stack or an abstraction for some hardware component.

The effects of picking expression-thread-view pairs as "Iris expressions" are most visible in weakest pre-conditions and Hoare triples. They will generally have the following forms;

$$\mathsf{wp}_{\mathcal{E}}\,(e, \mathcal{V})\,\{(v, \mathcal{V}_v).\, Q\}$$
$$\{P\}\,(e, \mathcal{V})\,\{(v, \mathcal{V}_v).\, Q\}_{\mathcal{E}}$$

That is, a WP or Hoare triple encodes the behaviors of a thread-local configuration $(e, \mathcal{V})$ where $\mathcal{V}$ is the thread-view that e starts executing with. Therefore it may be necessary that $\mathcal{V}$ satisfies certain properties that can be stated in the precondition P. The configuration, if terminates, will reduce to a configuration $(v, \mathcal{V}_v)$ where $\mathcal{V}_v$ is the thread-view after the execution. Properties of $\mathcal{V}_v$, like those of v, can be stated in the post-condition, which now has the type *Expr* $\times$ *ThreadView* $\to$ iProp.

Figure 7.1 presents a few *pure* WP rules that do not involve memory.

They are the expected WP rules from §6.6, but adapted with an arbitrary thread-view that mostly stays unchanged. Most notably, WP-BIND propagates the thread-view after the expression has finished to the evaluation context, and BL-WP-FORK picks the correct thread-view for the forked thread. Rules for binary operations are in the same form as BL-WP-PLUS, and the rule for **case** is similar to BL-WP-IF, and thus are elided.

However, for ORC11 memory operations, we know that expressions cannot run with arbitrary thread-views, as that may cause data races. The safety properties of thread-views are therefore unavoidable in the logic, but we can keep them manageable in form of new local assertions.

7.2 Basic Local Assertions for View-based RMC

In traditional separation logics, the points-to assertion is essential to achieve thread-modular reasoning: when interacting with the shared global state, it is sufficient to just have a points-to $\ell \mapsto v$ to access the location ℓ, keeping the rest of the global state out of the picture and in the "frame". Thanks to the frame rule, traditional separation logics enjoy the simple, thread-modular, small-footprint rules like HEAP-READ and HEAP-WRITE (§6.11).

We have the same goal for our RMC base logic: we want to achieve small-footprint rules that only require minimal ownership of bits of the global state for the operations in question, and let the frame rule do its job. Unfortunately, the bits of the global state needed for our RMC memory accesses are rather involved: we need (i) the memory h of the location ℓ being accessed, (ii) the thread-view $\mathcal{V}$ of the executing thread, and (iii) the global race detector view for ℓ ($\mathcal{N}(\ell)$). Most importantly, the safety and the result of an access depends on the relations between the thread-view $\mathcal{V}$ and the location's memory h and the global race detector view $\mathcal{N}$. We therefore need more assertions to make those relations explicit. We present our choice of local assertions below. In §7.4, we will present the RAs needed to fictionally separate the machine state and define these local assertions within the logic.

Definition 7.1 (Local Assertions for the Base Logic)**.** These assertions concern knowledge or resources over the executing thread's thread-view, the memory of the location being accessed, the race detector state, and their relations. All assertions are in iProp.

- The *seen thread-view* observation $\mathsf{Seen}(\mathcal{V})$ is a persistent knowledge that some thread's thread-view $\mathcal{V}$ is closed in the global memory.[2] This assertion is needed to guarantee that a memory access is grounded in the global memory.

[2] see Property 3.12, §3.3

- The *history* ownership assertion $\mathsf{Hist}_q(\ell, h)$ is a fractional ownership[3] of the write messages $h \in \mathit{History}$ of the location ℓ in the global memory, where $\mathit{History} ::= \mathit{Time} \xrightarrow{\text{fin}} \left\{ \mathsf{val} : \mathit{Val}, \mathsf{view} : \mathit{View}^? \right\}$. The fraction $q \in (0, 1]$ denotes shared or full ownership of ℓ's history. The allocated assertion $\mathsf{Local}_{\mathtt{A}}(\ell, h, V)$ says that the simple view V

[3] Boyland, "Checking interference with fractional permissions" [Boy03].

has observed the knowledge that ℓ's history h has been allocated. Both assertions are needed to perform any access to ℓ.

- The *non-atomic read* assertion $\mathsf{Read}_q^{\mathtt{na}}(\ell, \alpha)$ is the fractional ownership of a *subset* α of ℓ's non-atomic reads in the global race detector state. That is, $\alpha \subseteq \mathcal{N}(\ell).\mathsf{nr}$ if $\mathcal{N}$ is the global race detector state A related persistent knowledge is the non-atomic read observation $\mathsf{Local}_{\mathtt{R}}^{\mathtt{na}}(\ell, \alpha, V)$, which asserts that the simple view V has observed a subset α of ℓ's non-atomic reads. Both assertions are needed to perform race-free non-atomic reads on ℓ.
- The *atomic read* assertion $\mathsf{Read}_q^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha)$ is the fractional ownership of a *subset* α of ℓ's atomic reads in the global race detector state. That is, $\alpha \subseteq \mathcal{N}(\ell).\mathsf{ar}$. The persistent knowledge $\mathsf{Local}_{\mathtt{R}}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha, V)$ asserts that V has observed a subset α of ℓ's atomic reads. Both assertions are needed to perform race-free atomic reads on ℓ.
- The *atomic write* assertion $\mathsf{Write}_q^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha)$ is the fractional ownership of a *subset* α of ℓ's atomic writes in the global race detector state. That is, $\alpha \subseteq \mathcal{N}(\ell).\mathsf{aw}$. The persistent knowledge $\mathsf{Local}_{\mathtt{W}}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha, V)$ asserts that V has observed a subset α of ℓ's atomic writes. Both assertions are needed to perform race-free atomic writes on ℓ.
- Last but not least, the *block* ownership assertion $\dagger_q^n \ell$ is inherited from RustBelt. This ownership is created at allocation of a block whose base location is ℓ, and is only needed at deallocation of that block. The ownership guarantees that the whole block is deallocated together, *i.e.*, any thread holding a fraction of the block knows that the constituent locations are still alive. The assertion simply tracks the location ℓ and the size $n \in \mathbb{N}$ of the block.

These assertions satisfy several properties, given in Figure 7.2.

Property 7.2 (Seen Thread-view Observations)**.** The seen thread-view observation is timeless and persistent (BL-SEEN-TIMELESS and BL-SEEN-PERS).[4] As $\mathsf{Seen}(\mathcal{V})$ is only a snapshot of some thread's thread-view at some point, it can be joined with others (BL-SEEN-JOIN), or can be forked to get old snapshots (BL-SEEN-DOWNCLOSED). With $\mathsf{Seen}(\mathcal{V})$, we know that the thread-view $\mathcal{V}$ is closed in the global memory $\mathcal{M}$, but we do not have a rule for this property here. We will see how the property is established by the state interpretation in §7.6.

[4] see §6.8

Property 7.3 (History Ownership)**.** The assertion $\mathsf{Hist}_q(\ell, h)$ is timeless (BL-HIST-TIMELESS) and fractional (BL-HIST-FRAC-VALID and BL-HIST-FRAC). Owning a fraction of the assertion is sufficient to know the history of ℓ, as implied by BL-HIST-AGREE. A change to the history requires the full fraction, written as $\mathsf{Hist}(\ell, h)$ without the fraction $q = 1$, which is exclusive, as in BL-HIST-EXCL. (BL-HIST-EXCL is derivable from BL-HIST-FRAC-VALID and BL-HIST-FRAC.)

BL-HIST-DROP-SINGLETON allows us to truncate the current history to just a singleton of the latest write. This is a convenient abstraction for ℓ's

BL-SEEN-TIMELESS
$\mathsf{timeless}(\mathsf{Seen}(\mathcal{V}))$

BL-SEEN-PERS
$\mathsf{persistent}(\mathsf{Seen}(\mathcal{V}))$

BL-SEEN-JOIN
$\mathsf{Seen}(\mathcal{V}) * \mathsf{Seen}(\mathcal{V}') \dashv\vdash \mathsf{Seen}(\mathcal{V} \sqcup \mathcal{V}')$

BL-SEEN-DOWNCLOSED
$$\frac{\mathcal{V} \sqsubseteq \mathcal{V}'}{\mathsf{Seen}(\mathcal{V}') \vdash \mathsf{Seen}(\mathcal{V})}$$

BL-HIST-TIMELESS
$\mathsf{timeless}(\mathsf{Hist}_q(\ell, h))$

BL-HIST-FRAC-VALID
$\mathsf{Hist}_q(\ell, h) \vdash q \in (0, 1]$

BL-HIST-FRAC
$\mathsf{Hist}_q(\ell, h) * \mathsf{Hist}_{q'}(\ell, h) \dashv\vdash \mathsf{Hist}_{q+q'}(\ell, h)$

BL-HIST-AGREE
$\mathsf{Hist}_q(\ell, h) * \mathsf{Hist}_{q'}(\ell, h') \vdash h = h'$

BL-HIST-EXCL
$\mathsf{Hist}(\ell, h) * \mathsf{Hist}(\ell, h') \vdash \mathsf{False}$

BL-HIST-DROP-SINGLETON
$$\frac{h(t) = (v, V) \qquad t = \mathtt{max}(\mathtt{dom}(h))}{\mathsf{Hist}(\ell, h) \Rrightarrow_{\mathcal{E}} \mathsf{Hist}(\ell, [t \leftarrow (v, V)])}$$

BL-NAR-TIMELESS
$\mathsf{timeless}(\mathsf{Read}^{\mathsf{na}}_q(\ell, \alpha))$

BL-NAR-FRAC-VALID
$\mathsf{Read}^{\mathsf{na}}_q(\ell, \alpha) \vdash q \in (0, 1]$

BL-NAR-JOIN
$\mathsf{Read}^{\mathsf{na}}_q(\ell, \alpha) * \mathsf{Read}^{\mathsf{na}}_{q'}(\ell, \alpha') \vdash \mathsf{Read}^{\mathsf{na}}_{q+q'}(\ell, \alpha \cup \alpha')$

BL-ATR-TIMELESS
$\mathsf{timeless}(\mathsf{Read}^{\sqsupseteq \mathtt{rlx}}_q(\ell, \alpha))$

BL-ATR-FRAC-VALID
$\mathsf{Read}^{\sqsupseteq \mathtt{rlx}}_q(\ell, \alpha) \vdash q \in (0, 1]$

BL-ATR-JOIN
$\mathsf{Read}^{\sqsupseteq \mathtt{rlx}}_q(\ell, \alpha) * \mathsf{Read}^{\sqsupseteq \mathtt{rlx}}_{q'}(\ell, \alpha') \vdash \mathsf{Read}^{\sqsupseteq \mathtt{rlx}}_{q+q'}(\ell, \alpha \cup \alpha')$

BL-ATW-TIMELESS
$\mathsf{timeless}(\mathsf{Write}^{\sqsupseteq \mathtt{rlx}}_q(\ell, \alpha))$

BL-ATW-FRAC-VALID
$\mathsf{Write}^{\sqsupseteq \mathtt{rlx}}_q(\ell, \alpha) \vdash q \in (0, 1]$

BL-ATW-FRAC
$\mathsf{Write}^{\sqsupseteq \mathtt{rlx}}_q(\ell, \alpha) * \mathsf{Write}^{\sqsupseteq \mathtt{rlx}}_{q'}(\ell, \alpha) \dashv\vdash \mathsf{Write}^{\sqsupseteq \mathtt{rlx}}_{q+q'}(\ell, \alpha)$

BL-ATW-AGREE
$\mathsf{Write}^{\sqsupseteq \mathtt{rlx}}_q(\ell, \alpha) * \mathsf{Write}^{\sqsupseteq \mathtt{rlx}}_{q'}(\ell, \alpha') \vdash \alpha = \alpha'$

BL-NAL-JOIN
$\mathsf{Local}^{\mathsf{na}}_{\mathtt{R}}(\ell, \alpha, V) * \mathsf{Local}^{\mathsf{na}}_{\mathtt{R}}(\ell, \alpha', V) \vdash \mathsf{Local}^{\mathsf{na}}_{\mathtt{R}}(\ell, \alpha \cup \alpha', V)$

BL-ATRL-JOIN
$\mathsf{Local}^{\sqsupseteq \mathtt{rlx}}_{\mathtt{R}}(\ell, \alpha, V) * \mathsf{Local}^{\sqsupseteq \mathtt{rlx}}_{\mathtt{R}}(\ell, \alpha', V) \vdash \mathsf{Local}^{\sqsupseteq \mathtt{rlx}}_{\mathtt{R}}(\ell, \alpha \cup \alpha', V)$

BL-ALLOC-MONO
$$\frac{V \sqsubseteq V'}{\mathsf{Local}_{\mathtt{A}}(\ell, h, V) \vdash \mathsf{Local}_{\mathtt{A}}(\ell, h, V')}$$

BL-NAL-MONO
$$\frac{V \sqsubseteq V'}{\mathsf{Local}^{\mathsf{na}}_{\mathtt{R}}(\ell, \alpha, V) \vdash \mathsf{Local}^{\mathsf{na}}_{\mathtt{R}}(\ell, \alpha, V')}$$

BL-ATRL-MONO
$$\frac{V \sqsubseteq V'}{\mathsf{Local}^{\sqsupseteq \mathtt{rlx}}_{\mathtt{R}}(\ell, \alpha, V) \vdash \mathsf{Local}^{\sqsupseteq \mathtt{rlx}}_{\mathtt{R}}(\ell, \alpha, V')}$$

BL-ATWL-MONO
$$\frac{V \sqsubseteq V'}{\mathsf{Local}^{\sqsupseteq \mathtt{rlx}}_{\mathtt{W}}(\ell, \alpha, V) \vdash \mathsf{Local}^{\sqsupseteq \mathtt{rlx}}_{\mathtt{W}}(\ell, \alpha, V')}$$

BL-BLOCK-TIMELESS
$\mathsf{timeless}(\dagger^n_q\, \ell)$

BL-BLOCK-FRAC-VALID
$\dagger^n_q\, \ell \vdash q \in (0, 1]$

BL-BLOCK-JOIN
$\dagger^n_q\, \ell * \dagger^{n'}_{q'}\, (\ell + n) \dashv\vdash \dagger^{n+n'}_{q+q'}\, \ell$

FIGURE 7.2: Main properties of the base logic's local assertions

physical write events: while we need to maintain a set of write events (instead of a single value like in SC) because they may be still visible to some threads, once we know that certain writes are no longer visible, we can simply forget about them. In particular, if one can perform a race-free non-atomic write, all previous writes must be unreachable and should be forgotten, because it would be racy to read them then. Consequently, unlike the physical memory that only grows with more write messages, the history h in $\mathsf{Hist}_q(\ell, h)$ is not monotone—it grows during a period of atomic accesses, but will shrink back to a singleton with a non-atomic write. In later chapters, we will use BL-HIST-DROP-SINGLETON to switch between non-atomic and atomic access modes.

Property 7.4 (Race Detector Ownership)**.** The ownership assertions for parts of the race detector state are also timeless and fractional. Like

history ownership, fractions of the atomic write assertion $\mathsf{Write}_q^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha)$ maintain agreement on ℓ's set of atomic writes in the race detector (BL-ATW-FRAC), and the full fraction is required to update the set. A fraction q of the read assertions $\mathsf{Read}_q^{\mathtt{na}}(\ell, \alpha)$ or $\mathsf{Read}_q^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha)$ on the other hand does not maintain agreement. Instead, a fraction only maintains that the set α is a subset of the race detector's sets for non-atomic and atomic reads, respectively. This difference is due to the fact that, while writes maintain a total order (mo_ℓ) and thus must be updated with the full fraction, we do not enforce an order among concurrent reads, so each thread needs only some fraction of a read assertion to independently track its own reads, and sets of reads can be joined together using BL-NAR-JOIN or BL-ATR-JOIN.

Property 7.5 (Local Observations)**.** The local observations (for allocation, non-atomic and atomic reads, and atomic writes) are pure facts, and thus are timeless and persistent. In fact, their definitions are as follow.

$$\begin{aligned}
\mathsf{Local}_{\mathtt{A}}(\ell, h, V) &::= \exists t \in \mathtt{dom}(h).\, t \sqsubseteq V(\ell).\mathsf{w} \\
\mathsf{Local}_{\mathtt{W}}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha, V) &::= \alpha \sqsubseteq V(\ell).\mathsf{aw} \\
\mathsf{Local}_{\mathtt{R}}^{\mathtt{na}}(\ell, \alpha, V) &::= \alpha \sqsubseteq V(\ell).\mathsf{nr} \\
\mathsf{Local}_{\mathtt{R}}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha, V) &::= \alpha \sqsubseteq V(\ell).\mathsf{ar}
\end{aligned}$$

More importantly, they are *view-monotone*, *i.e.*, if one holds on a smaller view, it also holds on a bigger view (*e.g.*, see BL-ALLOC-MONO). View monotonicity is an important property that we will rely on heavily (see Chapter 8).

The local observations for reads can be joined together using BL-NAL-JOIN and BL-ATRL-JOIN.

Property 7.6 (Block Ownership)**.** The block ownership assertion is also timeless and fractional. BL-BLOCK-JOIN allows splitting and joining not just with the fractions, but also with the offsets. As such, for each location in a block one can own its bit of the block without needing to know the block size, and is guaranteed that the block is still alive.

7.3 Primitive Memory Rules

We now see how the local assertions are meant to be used in our primitive memory rules, given in Figures 7.3 to 7.6. Recall that in our base logic, the executing "expression" is a thread-local configuration of the actual expression and the executing thread's thread-view. The rules for allocation and deallocation are similar to that of non-atomic writes (BL-HOARE-WRITE-NA), but with the block ownership assertions. For the sake of simplicity, we will present them in a cleaner form in Chapter 9 (see NA-ALLOC and NA-DEALLOC).

7.3.1 *Rules for Fences*

Figure 7.3 presents the simplest memory-related primitive rules, for release and acquire fences. Both BL-HOARE-REL-FENCE and BL-HOARE-ACQ-FENCE requires $\mathsf{Seen}(\mathcal{V})$ as the pre-condition for a fence running with the

BL-HOARE-REL-FENCE
$$\{\mathsf{Seen}(\mathcal{V})\}\,(\mathbf{fence_{rel}}, \mathcal{V})\,\{(☣, \mathcal{V}').\ \mathcal{V}' \sqsupseteq \mathcal{V} * \mathsf{Seen}(\mathcal{V}') * \mathcal{V}'.\mathsf{frel} = \mathcal{V}'.\mathsf{cur}\}_{\mathcal{E}}$$

BL-HOARE-ACQ-FENCE
$$\{\mathsf{Seen}(\mathcal{V})\}\,(\mathbf{fence_{acq}}, \mathcal{V})\,\{(☣, \mathcal{V}').\ \mathcal{V}' \sqsupseteq \mathcal{V} * \mathsf{Seen}(\mathcal{V}') * \mathcal{V}'.\mathsf{cur} = \mathcal{V}'.\mathsf{acq}\}_{\mathcal{E}}$$

FIGURE 7.3: The base logic's primitive Hoare rules for fences

thread-view $\mathcal{V}$. Their post-conditions say that the fence instructions will return poison ☣ with a new thread-view $\mathcal{V}' \sqsupseteq \mathcal{V}$.[5] The effects of the fences are approximated by the properties of $\mathcal{V}'$. In case of an acquire fence, the current component of $\mathcal{V}'$ is updated to include its acquire component, exactly reflecting OM-ACQ-FENCE (§3.3). In case of a release fence, the release-fence component of $\mathcal{V}'$ is updated to include its current component. This only approximates OM-REL-FENCE (also §3.3) because it hides away (in the relation $\mathcal{V}' \sqsupseteq \mathcal{V}$) the changes to the per-location release views $\mathcal{V}'.\mathsf{rel}$. We made this abstraction to keep the rule simple, as we have never needed such detailed information on $\mathcal{V}'.\mathsf{rel}$.

[5] see OE-FENCE, §4.2

7.3.2 *Rules for Reads and Writes*

The rules for reads and writes are given in Figure 7.4 and Figure 7.5.

NON-ATOMIC READS. To guarantee data-race freedom, BL-HOARE-READ-NA requires a pre-condition that implies DRF-READ-NA (§3.4). That is, the pre-condition ensures that the executing thread has observed all writes (non-atomic and atomic) to ℓ. The pre-condition thus includes:

- $\mathsf{Seen}(\mathcal{V})$, like in other memory-related rules, to know the lower bound of the executing thread's thread-view; and
- a fraction q of the current singleton history $\mathsf{Hist}_q(\ell, [t \leftarrow (v, V^?)])$ where $(t, v, V^?)$ is ℓ's latest write; and
- the knowledge $\mathsf{Local}_{\mathtt{A}}(\ell, [t \leftarrow (v, V^?)], \mathcal{V}.\mathsf{cur})$ that the current thread-view $\mathcal{V}.\mathsf{cur}$ has observed not only ℓ's allocation but also its latest write; and
- a fraction q of the race detector's atomic writes set $\mathsf{Write}^{\sqsupseteq \mathbf{rlx}}_q(\ell, \alpha_w)$ and the knowledge $\mathsf{Local}^{\sqsupseteq \mathbf{rlx}}_{\mathtt{W}}(\ell, \alpha_w, \mathcal{V}.\mathsf{cur})$ that the current thread-view has observed all atomic writes (α_w);[6] and
- a fraction q of the race detector's non-atomic reads set $\mathsf{Read}^{\mathbf{na}}_q(\ell, \alpha_r)$, needed to extend the read set α_r with the read to be performed.

[6] Recall that observing the latest write (the previous part of the precondition) does not guarantee observation of all writes—it only guarantees observation of all non-atomic writes because non-atomic writes cannot race with one another, while atomic writes can. See also the discussion in Definition 3.25.

The post-condition is simple: the singleton value v is returned, the history ownership and atomic writes set are unchanged, the non-atomic reads set is extended with a new action id r representing this read, and the thread arrives at a new thread-view $\mathcal{V}'$ represented by $\mathsf{Seen}(\mathcal{V}')$. We can make an abstraction here on how $\mathcal{V}'$ is related to $\mathcal{V}$ (like in the rule BL-HOARE-REL-FENCE), but the higher-level rules require a detailed relation between $\mathcal{V}'$ and $\mathcal{V}$, so we simply keep the "raw" relation $\mathcal{V} \xrightarrow{\mathtt{R}:\mathbf{na},\ell,t,\bot,r} \mathcal{V}'$ from the operational semantics.[7] Note that we do know that $\mathcal{V} \sqsubseteq \mathcal{V}'$.

[7] see OM-POST-READ-TVIEW, §3.3

BL-HOARE-READ-NA

$$\frac{\mathsf{Local}_{\mathtt{A}}(\ell, \left[t \leftarrow (v, V^?)\right], \mathcal{V}.\mathsf{cur}) \qquad \mathsf{Local}_{\mathtt{W}}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha_w, \mathcal{V}.\mathsf{cur}) \qquad \mathsf{Local}_{\mathtt{R}}^{\mathtt{na}}(\ell, \alpha_r, V_r)}{\left\{\mathsf{Seen}(\mathcal{V}) * \mathsf{Hist}_q(\ell, \left[t \leftarrow (v, V^?)\right]) * \mathsf{Write}_q^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha_w) * \mathsf{Read}_q^{\mathtt{na}}(\ell, \alpha_r)\right\}\ (*^{\mathtt{na}}\ell, \mathcal{V})\ \left\{\begin{array}{l}(w, \mathcal{V}').\ w = v * \exists r.\ \mathcal{V} \xrightarrow{\mathtt{R}:\mathtt{na},\ell,t,\bot,r} \mathcal{V}' * \mathsf{Local}_{\mathtt{R}}^{\mathtt{na}}(\ell, \alpha_r \cup \{r\}, V_r \sqcup \mathcal{V}'.\mathsf{cur}) * \\ \qquad \mathsf{Seen}(\mathcal{V}') * \mathsf{Hist}_q(\ell, \left[t \leftarrow (v, V^?)\right]) * \mathsf{Write}_q^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha_w) * \mathsf{Read}_q^{\mathtt{na}}(\ell, \alpha_r \cup \{r\})\end{array}\right\}_{\mathcal{E}}}$$

BL-HOARE-WRITE-NA

$$\frac{\mathsf{Local}_{\mathtt{A}}(\ell, \left[t \leftarrow (v, V^?)\right], \mathcal{V}.\mathsf{cur}) \qquad \mathsf{Local}_{\mathtt{R}}^{\mathtt{na}}(\ell, \alpha_1, \mathcal{V}.\mathsf{cur}) \qquad \mathsf{Local}_{\mathtt{R}}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha_2, \mathcal{V}.\mathsf{cur}) \qquad \mathsf{Local}_{\mathtt{W}}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha_w, \mathcal{V}.\mathsf{cur})}{\left\{\mathsf{Seen}(\mathcal{V}) * \mathsf{Hist}(\ell, \left[t \leftarrow (v, V^?)\right]) * \mathsf{Write}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha_w) * \mathsf{Read}^{\mathtt{na}}(\ell, \alpha_1) * \mathsf{Read}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha_2)\right\}\ (\ell :=_{\mathtt{na}} v', \mathcal{V})\ \left\{\begin{array}{l}(☣, \mathcal{V}').\ \exists t' > t.\ \mathcal{V} \xrightarrow{\mathtt{W}:\mathtt{na},\ell,t',\bot,\bot} \mathcal{V}' * \mathsf{Local}_{\mathtt{A}}(\ell, [t' \leftarrow (v', \bot)], \mathcal{V}'.\mathsf{cur}) * \\ \qquad \mathsf{Seen}(\mathcal{V}') * \mathsf{Hist}(\ell, [t' \leftarrow (v', \bot)]) * \mathsf{Write}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha_w) * \mathsf{Read}^{\mathtt{na}}(\ell, \alpha_1) * \mathsf{Read}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha_2)\end{array}\right\}_{\mathcal{E}}}$$

FIGURE 7.4: The base logic's primitive Hoare rules for non-atomic reads and writes

Last but not least, we have an additional assertion $\mathsf{Local}_{\mathtt{R}}^{\mathtt{na}}(\ell, \alpha_r, V_r)$ in the pre-condition that is updated to $\mathsf{Local}_{\mathtt{R}}^{\mathtt{na}}(\ell, \alpha_r \cup \{r\}, V_r \sqcup \mathcal{V}'.\mathsf{cur})$ in the post-condition. This assertion is not needed for the rule per se. It is used to track the view V_r (which is $V_r \sqcup \mathcal{V}'.\mathsf{cur}$ after the step) that has observed the subset α_r of all non-atomic reads performed so far with the fraction $\mathsf{Read}_q^{\mathtt{na}}(\ell, \alpha_r)$. The view V_r will only be needed later when a thread has recollected the full fraction $\mathsf{Read}^{\mathtt{na}}(\ell, \alpha_r)$ where α_r is then the complete set of ℓ's non-atomic reads. At that point, race-free operations would require that the executing thread has observed all reads in α_r, which pins down to the thread's thread-view $\mathcal{V}$ including V_r, *i.e.*, $V_r \sqsubseteq \mathcal{V}.\mathsf{cur}$. We will see concretely how this view is used in Chapter 9.

NON-ATOMIC WRITES. BL-HOARE-WRITE-NA is the most demanding rule, as a non-atomic write cannot race with any other memory accesses to the same location ℓ.[8] The pre-condition therefore requires full ownership (the fraction $q = 1$) of ℓ' current singleton history, and of the 3 sets of the race detector's state, and the knowledge that the current thread-view $\mathcal{V}.\mathsf{cur}$ has observed ℓ's allocation, latest write, and those sets of all reads and all atomic writes to ℓ.[9]

The post-condition keeps most ownership unchanged, and only updates the singleton history ownership to $\mathsf{Hist}(\ell, [t' \leftarrow (v', \bot)])$, where t' is the new timestamp for the new write message, with the value v' and no message view. The thread arrives at a new thread-view $\mathcal{V}'$ computed from $\mathcal{V}'$ ($\mathcal{V} \xrightarrow{\mathtt{W}:\mathtt{na},\ell,t',\bot,\bot} \mathcal{V}'$),[10] and knows that the new current view $\mathcal{V}'.\mathsf{cur}$ has observed the new write ($\mathsf{Local}_{\mathtt{A}}(\ell, [t' \leftarrow (v', \bot)], \mathcal{V}'.\mathsf{cur})$).

ATOMIC READS. The rule BL-HOARE-READ-AT for atomic reads is rather simple: it requires as pre-condition fractional ownership q of ℓ's current history, and of an atomic read subset of the race detector's state $\mathsf{Read}_q^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha)$ to add the read to be performed. An atomic read only needs to avoid race with non-atomic writes,[11] so $\mathsf{Local}_{\mathtt{A}}(\ell, h, \mathcal{V}.\mathsf{cur})$ is

[8] see DRF-WRITE-NA, §3.4

[9] Recall that a fraction of $\mathsf{Read}_q^{\mathtt{na}}(\ell, \alpha_1)$ or $\mathsf{Read}_q^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha_2)$ only says that α_1 or α_2 is only a subset of the global set—we need a full fraction to be guaranteed that α_1 or α_2 *is* the global set.

[10] see OM-POST-WRITE-TVIEW, §3.3

[11] see DRF-READ-AT, §3.4

BL-HOARE-READ-AT

$$\frac{\mathbf{rlx} \sqsubseteq o \qquad \mathsf{Local}_{\mathtt{A}}(\ell, h, \mathcal{V}.\mathsf{cur}) \qquad \mathsf{Local}_{\mathtt{R}}^{\sqsupseteq \mathbf{rlx}}(\ell, \alpha, V_r)}{\begin{array}{l}\left\{\mathsf{Seen}(\mathcal{V}) * \mathsf{Hist}_q(\ell, h) * \mathsf{Read}_q^{\sqsupseteq \mathbf{rlx}}(\ell, \alpha)\right\} \\ \quad ({}^{*o}\ell, \mathcal{V}) \\ \left\{\begin{array}{l}(v, \mathcal{V}').\, \exists t, V^?, r.\, h(t) = (v, V^?) * \mathcal{V} \xrightarrow{\mathtt{R}:o,\ell,t,V^?,r} \mathcal{V}' * \mathsf{Local}_{\mathtt{R}}^{\sqsupseteq \mathbf{rlx}}(\ell, \alpha \cup \{r\}, V_r \sqcup \mathcal{V}'.\mathsf{cur}) * \\ \qquad \mathsf{Seen}(\mathcal{V}') * \mathsf{Hist}_q(\ell, h) * \mathsf{Read}_q^{\sqsupseteq \mathbf{rlx}}(\ell, \alpha \cup \{r\})\end{array}\right\}_{\mathcal{E}}\end{array}}$$

BL-HOARE-WRITE-AT

$$\frac{\mathbf{rlx} \sqsubseteq o \qquad \mathsf{Local}_{\mathtt{A}}(\ell, h, \mathcal{V}.\mathsf{cur}) \qquad \mathsf{Local}_{\mathtt{R}}^{\mathbf{na}}(\ell, \alpha_r, \mathcal{V}.\mathsf{cur}) \qquad \mathsf{Local}_{\mathtt{W}}^{\sqsupseteq \mathbf{rlx}}(\ell, \alpha_w, V_w)}{\begin{array}{l}\left\{\mathsf{Seen}(\mathcal{V}) * \mathsf{Hist}(\ell, h) * \mathsf{Write}^{\sqsupseteq \mathbf{rlx}}(\ell, \alpha_w) * \mathsf{Read}^{\mathbf{na}}(\ell, \alpha_r)\right\} \\ \quad (\ell :=_o v', \mathcal{V}) \\ \left\{\begin{array}{l}(\not{v}, \mathcal{V}').\, \exists t' \notin \mathtt{dom}(h), V'.\, \mathcal{V} \xrightarrow{\mathtt{W}:o,\ell,t',\bot,V'} \mathcal{V}' * \mathsf{Local}_{\mathtt{W}}^{\sqsupseteq \mathbf{rlx}}(\ell, \alpha_w \cup \{t'\}, V_w \sqcup \mathcal{V}'.\mathsf{cur}) * \\ \qquad \mathsf{Seen}(\mathcal{V}') * \mathsf{Hist}(\ell, h[t' \leftarrow (v', V')]) * \mathsf{Write}^{\sqsupseteq \mathbf{rlx}}(\ell, \alpha_w \cup \{t'\}) * \mathsf{Read}^{\mathbf{na}}(\ell, \alpha_r)\end{array}\right\}_{\mathcal{E}}\end{array}}$$

FIGURE 7.5: The base logic's primitive Hoare rules for atomic reads and writes

sufficient, because every non-atomic write would reset the history h to a singleton, and $\mathsf{Local}_{\mathtt{A}}(\ell, h, \mathcal{V}.\mathsf{cur})$ guarantees that the current thread-view $\mathcal{V}.\mathsf{cur}$ has observed some write in h.

In the post-condition, some value v in the history will be read and returned, and the atomic reads set is extended with the new action id for this read ($\mathsf{Read}_q^{\sqsupseteq \mathbf{rlx}}(\ell, \alpha \cup \{r\})$), and the thread arrives at a new thread-view $\mathcal{V}'$ computed from $\mathcal{V}$ with $\mathcal{V} \xrightarrow{\mathtt{R}:o,\ell,t,V^?,r} \mathcal{V}'$.[12] Note that the relaxed memory effects are contained within this relation for $\mathcal{V}'$, and will be abstracted later with higher-level rules.

[12] see OM-POST-READ-TVIEW, §3.3

The view V_r in $\mathsf{Local}_{\mathtt{R}}^{\sqsupseteq \mathbf{rlx}}(\ell, \alpha, V_r)$ plays the same role as its counterpart in BL-HOARE-READ-NA. V_r is guaranteed to have observed the subset α of atomic reads performed so far with the fraction q of $\mathsf{Read}_q^{\sqsupseteq \mathbf{rlx}}(\ell, \alpha)$. We will see concretely how V_r is used in Chapter 10.

ATOMIC WRITES. An atomic write must not race with non-atomic accesses, both reads and writes,[13] so BL-HOARE-WRITE-AT requires as pre-condition full fractions of the current history ownership $\mathsf{Hist}(\ell, h)$ and of the non-atomic reads set $\mathsf{Read}^{\mathbf{na}}(\ell, \alpha_r)$, as well as the knowledge $\mathsf{Local}_{\mathtt{A}}(\ell, h, \mathcal{V}.\mathsf{cur})$ and $\mathsf{Local}_{\mathtt{R}}^{\mathbf{na}}(\ell, \alpha_r, \mathcal{V}.\mathsf{cur})$ that the current thread-view $\mathcal{V}.\mathsf{cur}$ has observed all non-atomic writes and reads to ℓ. Additionally, the full ownership $\mathsf{Write}^{\sqsupseteq \mathbf{rlx}}(\ell, \alpha_w)$ of the race detector's atomic writes set for ℓ is needed to extend the set α_w with the write to be performed.

[13] see DRF-WRITE-AT, §3.4

The post-condition keeps the non-atomic reads set unchanged, extends the atomic writes set with the new timestamp t' (fresh in h), and updates the history and the atomic writes set to insert the new write $(h[t' \leftarrow (v', V')]$ and $\alpha_w \cup \{t'\})$. The new thread-view $\mathcal{V}'$ is computed from $\mathcal{V}$ accordingly.[14]

[14] see OM-POST-WRITE-TVIEW, §3.3

The view V_w in $\mathsf{Local}_{\mathtt{W}}^{\sqsupseteq \mathbf{rlx}}(\ell, \alpha_w, V_w)$ plays the same role as the view V_r in BL-HOARE-READ-NA and BL-HOARE-READ-AT. It is the view that has observed all atomic writes α_w done so far using $\mathsf{Write}^{\sqsupseteq \mathbf{rlx}}(\ell, \alpha_w)$. We will

BL-HOARE-CAS

$$\frac{\begin{array}{c}\mathbf{rlx} \sqsubseteq o_f, o_r, o_w \qquad \mathsf{Local}_{\mathsf{A}}(\ell, h, \mathcal{V}.\mathsf{cur}) \qquad \mathsf{Local}^{\mathbf{na}}_{\mathsf{R}}(\ell, \alpha_1, \mathcal{V}.\mathsf{cur}) \\ \mathsf{Local}^{\sqsupseteq \mathbf{rlx}}_{\mathsf{R}}(\ell, \alpha_2, V_r) \qquad \mathsf{Local}^{\sqsupseteq \mathbf{rlx}}_{\mathsf{W}}(\ell, \alpha_w, V_w) \qquad \forall v_0 \in \mathsf{Readable}(h, \mathcal{V}).\ \vdash v_0 =^? v_r\end{array}}{\begin{array}{l}\left\{\begin{array}{l}\mathsf{Seen}(\mathcal{V}) * \mathsf{Hist}(\ell, h) * \mathsf{Write}^{\sqsupseteq \mathbf{rlx}}(\ell, \alpha_w) * \mathsf{Read}^{\mathbf{na}}(\ell, \alpha_1) * \mathsf{Read}^{\sqsupseteq \mathbf{rlx}}_q(\ell, \alpha_2) * \\ P_{\mathrm{cmp}} * \square\left((v_r = \ell_r)\ ?\ (P_{\mathrm{cmp}} \mathrel{-\!\!*} \Phi_{\mathrm{cmp}}(\ell_r)) : \mathsf{True}\right)\end{array}\right\} \\ \quad (\mathbf{CAS}^{o_f, o_r, o_w}(\ell, v_1, v_2), \mathcal{V}) \\ \left\{\begin{array}{l}(b, \mathcal{V}').\ P_{\mathrm{cmp}} * \exists h', t, t', v', V, V', r, \mathcal{V}_x.\ \mathcal{V} \sqsubseteq \mathcal{V}_x \sqsubseteq \mathcal{V}' * h(t') = (v', V') * \\ \qquad \mathsf{Local}^{\sqsupseteq \mathbf{rlx}}_{\mathsf{W}}(\ell, (b)\ ?\ (\alpha_w \cup \{t\}) : \alpha_w, V_w \sqcup \mathcal{V}'.\mathsf{cur}) * \mathsf{Local}^{\sqsupseteq \mathbf{rlx}}_{\mathsf{R}}(\ell, \alpha_2 \cup \{r\}, V_r \sqcup \mathcal{V}'.\mathsf{cur}) * \\ \qquad \mathsf{Seen}(\mathcal{V}') * \mathsf{Hist}(\ell, h') * \mathsf{Write}^{\sqsupseteq \mathbf{rlx}}(\ell, (b)\ ?\ (\alpha_w \cup \{t\}) : \alpha_w) * \mathsf{Read}^{\mathbf{na}}(\ell, \alpha_1) * \mathsf{Read}^{\sqsupseteq \mathbf{rlx}}_q(\ell, \alpha_2 \cup \{r\}) * \\ \qquad\quad b = \mathbf{false} * v_r \neq v' * h' = h * \mathcal{V} \xrightarrow{\mathsf{R}:o_f, \ell, t', V', r} \mathcal{V}_x \\ \quad \vee \quad b = \mathbf{true} * v_r = v' * t \notin \mathsf{dom}(h) * t = t' + 1 * V \sqsupseteq V' * h' = h[t \leftarrow (v_w, V)] * \\ \qquad\quad \mathcal{V} \xrightarrow{\mathsf{R}:o_r, \ell, t', V', r} \mathcal{V}_x \xrightarrow{\mathsf{W}:o_w, \ell, t, V', V} \mathcal{V}'\end{array}\right\}_{\mathcal{E}}\end{array}}$$

where $\Phi_{\mathrm{cmp}}(\ell_r) ::= (\Rightarrow\!\!\!\!\!\!\Rightarrow_{\mathcal{E}} \exists q_r, h_r.\ \triangleright \mathsf{Hist}_{q_r}(\ell_r, h_r)) \wedge (\forall \ell' \in \mathsf{Readable}(h, \mathcal{V}) \setminus \{\ell_r\}.\ \Rightarrow\!\!\!\!\!\!\Rightarrow_{\mathcal{E}} \exists q', h'.\ \triangleright \mathsf{Hist}_{q'}(\ell', h'))$

FIGURE 7.6: The base logic's primitive Hoare rule for CASes

also see concretely how V_w is used in Chapter 10.

7.3.3 *A Rule for CASes*

We present a Hoare rule BL-HOARE-CAS for CASes in Figure 7.6. It is a rather complicated rule, because a CAS is a combination of a read, a write, and a comparison that can be a pointer comparison.

First of all, a CAS cannot race with non-atomic accesses (reads and writes),[15] so the pre-condition requires the full fractions of the history ownership $\mathsf{Hist}(\ell, h)$, and of the non-atomic reads set $\mathsf{Read}^{\mathbf{na}}(\ell, \alpha_1)$, and the knowledge $\mathsf{Local}_{\mathsf{A}}(\ell, h, \mathcal{V}.\mathsf{cur})$ and $\mathsf{Local}^{\mathbf{na}}_{\mathsf{R}}(\ell, \alpha_1, \mathcal{V}.\mathsf{cur})$ that $\mathcal{V}.\mathsf{cur}$ has observed ℓ's allocation and all of its non-atomic reads and writes.

[15] see DRF-UPDATE, §3.4

Second, the pre-condition needs the full fraction of the atomic writes set $\mathsf{Write}^{\sqsupseteq \mathbf{rlx}}(\ell, \alpha_w)$ and a fraction of the atomic reads set $\mathsf{Read}^{\sqsupseteq \mathbf{rlx}}_q(\ell, \alpha_2)$ in order to potentially extend those sets with a write event and a read event that are to be generated by this CAS operation. The views V_w and V_r have observed the sets α_w and α_2 respectively, and play the similar role to those in the read and write rules, which we will see in Chapter 10.

Third, the post-condition is a combination of read and write effects. The operation returns a boolean value b to indicate success or failure. In any case, a message (t', v', V') in h will be read, and the atomic reads set α_2 will be extended with a new action id r for that read. The non-atomic reads set remains unchanged.

- In case the CAS fails, *i.e.*, $b = \mathbf{false}$, we know that the value read v' is not equal to the expected value v_r ($\vdash v' \neq v_r$),[16] that the history h and the atomic writes set α_w also remain unchanged, and that the thread-view effect is similar to that of a read with the mode o_f ($\mathcal{V} \xrightarrow{\mathsf{R}:o_f, \ell, t', V', r} \mathcal{V}_x$).[17]

[16] see Definition 4.8, §4.2

[17] see OM-POST-READ-TVIEW, §3.3

- In case the CAS succeeds, *i.e.*, $b = \texttt{true}$, the read value v' is exactly the expected value v_r, and a new message (t, v_w, V) is inserted into h ($h' = h[t \leftarrow (v_w, V)]$) right next to the read message ($t = t' + 1$) which guarantees the atomicity of the read and the write generated by the CAS. The new thread-view $\mathcal{V}'$ and the write message view V are computed from the old thread-view $\mathcal{V}$ and the read message view V' accordingly.[18] The atomic writes set α_w is also extended with the new timestamp t.

[18] see OM-UPDATE, §3.3

Finally, we look at the rule's components concerning (pointer) comparison. The rule requires safety in the comparison between the expected value v_r and any potential value v_0 that the CAS may read: $\forall v_0 \in \text{Readable}(h, \mathcal{V}).\ \vdash v_0 \stackrel{?}{=} v_r$. The set of readable values $\text{Readable}(h, \mathcal{V})$ is lifted for histories from Definition 4.6 for the global memory. The comparison is safe if the values are comparable (Definition 4.9).

DETERMINISTIC POINTER COMPARISON. If the comparison is between locations, *i.e.*, if v_r is a non-null location ℓ_r, the pre-condition of BL-HOARE-CAS requires some extra resources P_{cmp} to learn that compared locations are alive, and thus to guarantee deterministic comparison.[19] Furthermore, P_{cmp} will only be used to derive facts and will not be consumed, so it is returned as-is in the post-condition. How P_{cmp} will be used is encoded in the predicate $\Phi_{\text{cmp}}(\ell_r)$ which employs a *classical* conjunction. Intuitively, the persistent implication $\square(P_{\text{cmp}} \mathrel{-\!\!*} \Phi_{\text{cmp}}(\ell_r))$ requires that the resources in P_{cmp} *simultaneously* support two goals:

[19] Comparability makes sure that we need not care about the case where the expected value v_r is an integer but the readable values can be locations. If that is the case, v_r must be 0, and the comparison always fails. See also Definition 4.8.

1. using P_{cmp} and potentially opening some invariant with the fancy update $\Rightarrow\!\!\!\!\Rightarrow_{\mathcal{E}}$, one gets some fraction of the ownership $\text{Hist}_{q_r}(\ell_r, h_r)$ of the expected value ℓ_r, which is sufficient to learn that ℓ_r is alive.
2. for any location ℓ' readable from h that is not ℓ_r,[20] using P_{cmp} and potentially opening some invariant, one also learns that ℓ' is alive.

[20] If the read value ℓ' is also ℓ_r, then the first part of the classical conjunction is also the proof that ℓ' is alive.

7.3.4 *A Stronger WP Rule for CASes*

We present the rule BL-WP-CAS (Figure 7.7) for CASes that is stronger than BL-HOARE-CAS. Note that this rule is very technical and is only used to get stronger GPS rules that will be used in Part III. Readers are welcome to skip this rule and continue with the next section.

The rule is written in form of weakest pre-conditions—a general fashion that is common with Iris WPs, where the post-condition is universally quantified as Φ. This style is not only more convenient to use in practice in Coq, but also important to make our CAS rule stronger.

Notation 7.7 (Iris WP-style Rules)**.** Recall Definition 6.7 where Hoare triples are derived from WPs. In practice (in Coq), Iris rules for Hoare triples and WPs are usually written with a universally quantified post-condition Φ, so that they can be easily applied to a goal with an arbitrarily shaped WP. For example, if e is not a value,[21] a Hoare rule $\vdash \{P\}\, e\, \{v.\, Q\}$ for e can instead be written as:

[21] If it is a value, we do not have a later modality.

$$\vdash \square\,(P \mathrel{-\!\!*} \forall \Phi.\, (\triangleright \forall v.\, Q \mathrel{-\!\!*} \Phi(v)) \mathrel{-\!\!*} \mathsf{wp}\, e\, \{\Phi\})$$

That is, the rule intuitively encodes that Q is the *strongest post-condition* for e under the pre-condition P. The later modality allows us to prove the post-condition only after the step, at which point our resources which are under a later before the step have been made available. This from of rule is more applicable to a goal of form $\mathsf{wp}\, K[e]\,\{\Psi\}$: we first apply the bind rule WP-BIND to focus on the expression e and push the continuation into the post-condition (*i.e.*, $\mathsf{wp}_{\mathcal{E}}\, e\,\{v.\, \mathsf{wp}_{\mathcal{E}}\, K[v]\,\{\Psi\}\}$) which will then be used to instantiate the predicate Φ of the rule.

We state our CAS rule in the following form:

$$\frac{R \qquad R' \qquad \triangleright \forall v, \mathcal{V}_v.\, Q \mathrel{-\!\!*} \Phi(v, \mathcal{V}_v)}{P \vdash \mathsf{wp}\,(e, \mathcal{V})\,\{\Phi\}}$$

where P is the pre-condition, Q is the post-condition, and R and R' are extra premises. The wand implication $\triangleright \forall v, \mathcal{V}_v.\, Q \mathrel{-\!\!*} \Phi(v, \mathcal{V}_v)$ is for the post-condition and is the right-most premise. A rule of this form can be read as the following Hoare rule:

$$\frac{R \qquad R'}{\vdash \{P\}\,(e, \mathcal{V})\,\{(v, \mathcal{V}_v).\, Q\}}$$

Our CAS rule BL-WP-CAS, however, is strengthened by moving the later inside and adding fancy updates to the post-condition—a combination called *wand step viewshifts*.

Notation 7.8 (Wand Step Viewshifts). $\boxed{P \Rrightarrow\!\!\!\!\!*\,{}^{\mathcal{E}'}_{\mathcal{E}}\; Q}$

The wand step viewshift is a balanced (potentially mask-changing) viewshift that has a later in between.

$$P \Rrightarrow\!\!\!\!\!*\,{}^{\mathcal{E}'}_{\mathcal{E}}\; Q ::= P \mathrel{-\!\!*} {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}'} \triangleright\, {}^{\mathcal{E}'}\!\Rrightarrow^{\mathcal{E}} Q$$

We can now look at BL-WP-CAS in Figure 7.7. The rule can be applied with an arbitrary post-condition Ψ which typically is the continuation after executing the CAS. The rule says that the client can go on proving Ψ *assuming* the return value $(b, \mathcal{V}')$ (together with other variables) universally quantified in the right-most premise, as well as the resources on the left-hand side of the wand implications. Compared to the alternative WP-style reading (Notation 7.7) of the Hoare rule BL-HOARE-CAS, BL-HOARE-CAS are strengthened in several ways.

- The client of the rule does not need to specify and provide P_{cmp} in the pre-condition. Instead, the client can pick P_{cmp} *after* learning all information about the results of the CAS (*e.g.*, the return value b, the read and write timestamps, the new history h', the thread-views and views). In fact, the client only needs to provide (prove) P_{cmp} and how it is to be used (Φ) if the CAS succeeds ($b = \texttt{true}$).

- Note that the client however does not know that "$v_r = v'$ in case $b = \texttt{true}$" *before* picking and proving P_{cmp}, because P_{cmp} is needed to achieve that deterministic comparison result.

BL-WP-CAS

$$\frac{\begin{array}{c}\mathbf{rlx} \sqsubseteq o_f, o_r, o_w \\ \mathsf{Local}_{\mathtt{A}}(\ell, h, \mathcal{V}.\mathsf{cur}) \\ \mathsf{Local}^{\mathbf{na}}_{\mathtt{R}}(\ell, \alpha_1, \mathcal{V}.\mathsf{cur}) \\ \mathsf{Local}^{\sqsupseteq \mathbf{rlx}}_{\mathtt{R}}(\ell, \alpha_2, V_r) \\ \mathsf{Local}^{\sqsupseteq \mathbf{rlx}}_{\mathtt{W}}(\ell, \alpha_w, V_w) \\ \forall v_0 \in \mathsf{Readable}(h, \mathcal{V}). \\ \vdash v_0 =^? v_r\end{array}}{\begin{array}{l}\forall b, \mathcal{V}', h', t, t', v', V, V', r, \mathcal{V}_x. \\ \left(\begin{array}{l} \mathcal{V} \sqsubseteq \mathcal{V}_x \sqsubseteq \mathcal{V}' * h(t') = (v', V') * \\ \mathsf{Local}^{\sqsupseteq \mathbf{rlx}}_{\mathtt{W}}(\ell, (b)\,?\,\alpha_w \cup \{t\} : \alpha_w, V_w \sqcup \mathcal{V}'.\mathsf{cur}) * \\ \mathsf{Local}^{\sqsupseteq \mathbf{rlx}}_{\mathtt{R}}(\ell, \alpha_2 \cup \{r\}, V_r \sqcup \mathcal{V}'.\mathsf{cur}) * \\ \quad b = \mathbf{false} * v_r \neq v' * h' = h * \mathcal{V} \xrightarrow{\mathtt{R}:o_f,\ell,t',V',r} \mathcal{V}_x \\ \vee \quad b = \mathbf{true} * t \notin \mathtt{dom}(h) * t = t' + 1 * V \sqsupseteq V' * \\ \quad\quad h' = h[t \leftarrow (v_w, V)] * \mathcal{V} \xrightarrow{\mathtt{R}:o_r,\ell,t',V',r} \mathcal{V}_x \xrightarrow{\mathtt{W}:o_w,\ell,t,V',V} \mathcal{V}' \end{array}\right) \\ \mathrel{-\!\!*} \exists P_{\mathrm{cmp}}.\, P_{\mathrm{cmp}} * \square\left((b \wedge v_r = \ell_r)\,?\,\Phi_{\mathrm{cmp}}(P_{\mathrm{cmp}}, \ell_r, v') : \mathsf{True}\right) * \\ \left(\begin{array}{l} ((b)\,?\,v_r = v' : \mathsf{True}) * P_{\mathrm{cmp}} * \mathsf{Seen}(\mathcal{V}') * \mathsf{Hist}(\ell, h') * \\ \mathsf{Write}^{\sqsupseteq \mathbf{rlx}}(\ell, (b)\,?\,\alpha_w \cup \{t\} : \alpha_w) * \mathsf{Read}^{\mathbf{na}}(\ell, \alpha_1) * \mathsf{Read}^{\sqsupseteq \mathbf{rlx}}_{q}(\ell, \alpha_2 \cup \{r\}) \\ \quad \Rrightarrow\!\!\!*^{\mathcal{E}'}_{\mathcal{E}}\ \Psi(b, \mathcal{V}') \end{array}\right)\end{array}}$$

$$\mathsf{Seen}(\mathcal{V}) * \mathsf{Hist}(\ell, h) * \mathsf{Write}^{\sqsupseteq \mathbf{rlx}}(\ell, \alpha_w) * \mathsf{Read}^{\mathbf{na}}(\ell, \alpha_1) * \mathsf{Read}^{\sqsupseteq \mathbf{rlx}}_{q}(\ell, \alpha_2) \vdash \mathsf{wp}_{\mathcal{E}}\ (\mathbf{CAS}^{o_f,o_r,o_w}(\ell, v_1, v_2), \mathcal{V})\ \{\Psi\}$$

where $\Phi_{\mathrm{cmp}}(P_{\mathrm{cmp}}, \ell_r, v') ::= \wedge \begin{array}{l} P_{\mathrm{cmp}} \overset{\mathcal{E}}{\Rrightarrow\!\!\!*}{}^{\mathcal{E}_r}\ \exists q_r, h_r.\ \triangleright \mathsf{Hist}_{q_r}(\ell_r, h_r) \\ \forall \ell' = v' \neq \ell_r.\, P_{\mathrm{cmp}} \overset{\mathcal{E}}{\Rrightarrow\!\!\!*}{}^{\mathcal{E}'} \triangleright \overset{\mathcal{E}'}{\Rrightarrow}{}^{\mathcal{E}_0(\ell')} \exists q', h'.\ \triangleright \mathsf{Hist}_{q'}(\ell', h') \end{array}$

FIGURE 7.7: The base logic's primitive WP rule for CASes

- *After* proving P_{cmp} and Φ, the client does get the deterministic comparison result and P_{cmp} back. Recall that P_{cmp} is not consumed and is only needed to know that compared locations are alive.

- The client also acquires the returned resources (*i.e.*, the history ownership and the ownership of the reads and writes sets), and then can prove the continuation with the wand step viewshift. The wand viewshift $\Rrightarrow\!\!\!*^{\mathcal{E}'}_{\mathcal{E}}$ allows the client to make a mask-changing viewshift from the mask $\mathcal{E}$ to the mask $\mathcal{E}'$ to open invariants ($\mathcal{E}'$ is of the client's choice), then to strip a later in any resources that the client owns at that point, and then to close the invariants and return to the mask $\mathcal{E}$, all in order to prove the continuation $\Psi(b, \mathcal{V}')$. Note that if the client uses BL-HOARE-CAS, they would not have a later at their disposal, because the results of the CAS operation are only available *after* the later is introduced.

- Finally, the client can also rely on a later and mask-changing viewshifts when proving Φ_{cmp}, *i.e.*, the proof that P_{cmp} implies that the compared locations are alive. In particular, the mask $\mathcal{E}_r$ and the function $\mathcal{E}_0$ from locations to masks are also of the client's choice. The client only needs to show that expected value ℓ_r and the read value ℓ' are alive. Interestingly, the client can do so by opening invariants (of the client's choice) without closing them.

7.4 Resource Algebras for Basic Local Assertions

We briefly explain the resource algebras needed to define our local assertions and tie them to the physical state. We will need 5 RAs.

Definition 7.9 (Lattice RA for Seen Thread-view Observations)**.** The *lattice* RA $\textsc{Lat}(A)$ takes a join semi-lattice A and defines the composition as the lattice's join operation, the core function as the identity function (so that every element is the core of itself), and validity is trivial. That is, $\textsc{Lat}(A) ::= (A, (\lambda\_.\, \mathsf{True}), \mathrm{id}, \sqcup)$. If the lattice A has a bottom element $\bot \sqsubseteq a$ for any $a \in A$, then $\bot$ is the unit of $\textsc{Lat}(A)$. Note that RA inclusion $\preccurlyeq$ then coincides with the lattice order $\sqsubseteq$. Most importantly, the RA has the following properties.

$$\forall a, b.\, \mathsf{valid}(\bullet\, a \cdot \circ\, b) \Rightarrow b \sqsubseteq a \qquad \text{(AUTH-LAT-VALID)}$$

$$\forall a, b.\, b \sqsubseteq a \Rightarrow \bullet\, b \rightsquigarrow \bullet\, a \cdot \circ\, a \qquad \text{(AUTH-LAT-UPDATE)}$$

For $\mathsf{Seen}(\mathcal{V})$, we use the RA $\textsc{SeenR} = \textsc{Auth}(\textsc{Lat}(\mathit{View}))$. It is an optimization that we only track a simple view with $\textsc{Lat}(\mathit{View})$ and not a thread-view with $\textsc{Lat}(\mathit{ThreadView})$. Recall that the role of $\mathsf{Seen}(\mathcal{V})$ is to guarantee that $\mathcal{V}$ is closed in the global memory, which can be done instead by just guaranteeing that $\mathcal{V}.\mathsf{acq}$ is closed in the global memory, because $\mathcal{V}.\mathsf{acq}$ includes all other components of a wellformed $\mathcal{V}$. The closedness condition also means that we need not track one view per thread: we simply track the upper bound V_{up} for all acquire components of all thread-views and require that V_{up} is closed in the global memory.

In particular, the authoritative element $\bullet\, V_{up}$ guarantees that, for any other fragmentary element $\circ\, \mathcal{V}'.\mathsf{acq}$, thanks to AUTH-LAT-VALID, $\mathcal{V}'.\mathsf{acq} \sqsubseteq V_{up}$. Furthermore, due to AUTH-LAT-UPDATE, $\bullet\, V_{up}$ can only be updated to a bigger view, mirroring the property that views only grow.

Definition 7.10 (Fractional Agreement Map RA for History)**.** We use the RA $\textsc{HistR} = \textsc{Auth}(\textsc{Map}(\mathit{Loc}, \textsc{Frac} \times \textsc{Ag}(\mathit{History}^?)))$ for $\mathsf{Hist}_q(\ell, h)$.

The *agree* RA Ag only provides valid composition between elements that are the same, and the *fractional* RA Frac provides valid composition between non-negative quotients that sum up to no greater than 1 (*i.e.*, they are in the range $[0, 1)$). We use them together using the product RA (written here as $\times$) which provides valid composition point-wise. The map RA Map takes a key type and a value RA, and provides valid composition key-wise, using the valid composition of the value RA.

As such, our combined use of Map with Frac and Ag gives per-location agreement between fractions of history ownership, and with the full fraction we can change the history. We use the option type $\mathit{History}^?$ to support deallocation: when a location is deallocated, then its history will be None. We use Auth to have the authoritative element be the complete memory, and the fragmentary elements of singleton maps will be used to define $\mathsf{Hist}_q(\ell, h)$. More concretely, we have the following properties.

$$\forall m, \ell, q, h.\, \mathsf{valid}(\bullet\, m \cdot \circ\, [\ell \leftarrow (q, \mathsf{ag}(h))]) \Rightarrow m(\ell) = (1, \mathsf{ag}(h))$$

$$\forall \ell, q, h, q', h'.\, \mathsf{valid}(\circ\, [\ell \leftarrow (q, \mathsf{ag}(h))] \cdot \circ\, [\ell \leftarrow (q', \mathsf{ag}(h'))]) \Rightarrow h = h'$$

$$\forall m, \ell, h.\, \bullet\, m \cdot \circ\, [\ell \leftarrow (1, \mathsf{ag}(h))] \rightsquigarrow \bullet\, m[\ell \leftarrow (1, \mathsf{ag}(h'))] \cdot \circ\, [\ell \leftarrow (1, \mathsf{ag}(h'))]$$

The injection ag is the constructor of the RA AG.

Definition 7.11 (Fractional Map RA for Atomic Writes Sets)**.** We use the RA WRITER $=$ AUTH(MAP(*Loc*, FRAC $\times$ AG(*ActIds*))) for the fractional per-location ownership of atomic writes sets. This RA is similar to that for history ownership, but tracks a set of action ids instead.

Definition 7.12 (Fractional Set Lattice RA for Reads Sets)**.** We use a slightly different RA READR $=$ AUTH(MAP(*Loc*, FRAC $\times$ LAT(*ActIds*))). That is, we use LAT(*ActIds*) in place of AG(*ActIds*). As such, we do not have agreement between the fractions of read sets. In exchange, a fraction is sufficient to grow the set: we can update the element $\circ\, [\ell \leftarrow (q, \alpha)]$ (together with the authoritative element) to $\circ\, [\ell \leftarrow (q, \alpha')]$ where $\alpha \subseteq \alpha'$ without requiring $q = 1$. Note that because we use the lattice RA LAT, the sets can only grow. More concretely, we have the following properties.

$$\forall m, \ell, q, \alpha.\, \mathsf{valid}(\bullet\, m \cdot \circ\, [\ell \leftarrow (q, \alpha)]) \Rightarrow \exists \alpha'.\, m(\ell) = (1, \alpha') \land \alpha \subseteq \alpha'$$
$$\forall m, \ell, q, \alpha, \alpha', \alpha''.\, m(\ell) = (1, \alpha') \Rightarrow \alpha \subseteq \alpha'' \Rightarrow$$
$$\bullet\, m \cdot \circ\, [\ell \leftarrow (1, \alpha)] \rightsquigarrow \bullet\, m[\ell \leftarrow (q, \alpha' \cup \alpha'')] \cdot \circ\, [\ell \leftarrow (q, \alpha'')]$$

Definition 7.13 (Fractional Block RA for Block Ownership)**.** We use the RA BLOCKR $=$ AUTH(MAP($\mathbb{N}^+$, FRAC $\times$ MAP($\mathbb{Z}$, EX(1)))). That is, we use a map from block indices (in $\mathbb{N}^+$) to fractional maps from offsets (in $\mathbb{Z}$) to exclusive tokens (of type unit 1). The outer map allows us to have per-block ownership with full fraction, and the inner map allows use to split that full fraction between the offsets in the same block. The ownership of every single offset in a block represents the block ownership of a location and, thanks to the exclusive RA EX, such per-location block ownership is unique.

7.5 State Interpretation

We now define the local assertions and the state interpretation S for our base logic. We first need a few *global* ghost locations to store the RAs defined in the previous section. They are γ_{SEEN}, γ_{HIST}, γ_{NAR}, γ_{ATW}, γ_{ATR}, and γ_{BLK}. These ghost locations will need to be allocated before any program runs (in the adequacy proof, see Theorem 7.19).

Definition 7.14 (Ghost State Model of Local Assertions)**.** We define our local assertions purely as ghost ownership of fragmentary elements.

$$\begin{aligned}
\mathsf{Seen}(\mathcal{V}) &::= \boxed{\circ\, \mathcal{V}.\mathsf{acq} : \text{SEENR}}^{\gamma_{\text{SEEN}}} \\
\mathsf{Hist}_q(\ell, h) &::= \boxed{\circ\, [\ell \leftarrow (q, \mathsf{ag}(\mathsf{Some}(h)))] : \text{HISTR}}^{\gamma_{\text{HIST}}} \\
\mathsf{Write}_q^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha) &::= \boxed{\circ\, [\ell \leftarrow (q, \mathsf{ag}(\alpha))] : \text{WRITER}}^{\gamma_{\text{ATW}}} \\
\mathsf{Read}_q^{\mathtt{na}}(\ell, \alpha) &::= \boxed{\circ\, [\ell \leftarrow (q, \alpha)] : \text{READR}}^{\gamma_{\text{NAR}}} \\
\mathsf{Read}_q^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha) &::= \boxed{\circ\, [\ell \leftarrow (q, \alpha)] : \text{READR}}^{\gamma_{\text{ATR}}} \\
\dagger_q^n\, \ell &::= \boxed{\circ\, [\ell \leftarrow (q, [m \leftarrow \mathsf{ex}() \mid m \in [0, n)])] : \text{BLOCKR}}^{\gamma_{\text{BLK}}}
\end{aligned}$$

The injection ex is the constructor of the RA EX.

Definition 7.15 (Ghost Ownership for the Global State)**.** The ghost ownership GlobalGhost that mirrors the global physical state is defined with ownership of authoritative elements. It takes as inputs the physical state $(\mathcal{M}, \mathcal{N})$ and the global upper bound V_{up} of all threads's thread-views. Additionally, to support truncating histories with BL-HIST-DROP-SINGLETON, it also takes as input a view that tracks for each location the timestamp of its latest write. We call this view the *cut* view V_{cut}.

$$
\begin{aligned}
&\mathsf{GlobalGhost}(\mathcal{M}, \mathcal{N}, V_{up}, V_{cut}) ::= \\
&\quad \boxed{\bullet\, V_{up} : \textsc{SeenR}}^{\gamma_{\text{SEEN}}} * \\
&\quad \boxed{\bullet\, [\ell \leftarrow (1, \mathsf{ag}(\mathtt{trunc}(\mathcal{M}(\ell), V_{cut}(\ell).\mathsf{w}))) \mid \ell \in \mathtt{dom}(\mathcal{M})] : \textsc{HistR}}^{\gamma_{\text{HIST}}} * \\
&\quad \boxed{\bullet\, [\ell \leftarrow (1, \mathsf{ag}(\mathcal{N}(\ell).\mathsf{aw})) \mid \ell \in \mathtt{dom}(\mathcal{N})] : \textsc{WriteR}}^{\gamma_{\text{ATW}}} * \\
&\quad \boxed{\bullet\, [\ell \leftarrow (1, \mathcal{N}(\ell).\mathsf{nr}) \mid \ell \in \mathtt{dom}(\mathcal{N})] : \textsc{ReadR}}^{\gamma_{\text{NAR}}} * \\
&\quad \boxed{\bullet\, [\ell \leftarrow (1, \mathcal{N}(\ell).\mathsf{ar}) \mid \ell \in \mathtt{dom}(\mathcal{N})] : \textsc{ReadR}}^{\gamma_{\text{ATR}}} * \\
&\quad \boxed{\bullet\, [i \leftarrow (1, [n \leftarrow \mathsf{ex}() \mid (i, n) \in \mathtt{dom}(\mathcal{M})]) \mid (i, \_) \in \mathtt{dom}(\mathcal{M})] : \textsc{BlockR}}^{\gamma_{\text{BLK}}}
\end{aligned}
$$

where

$$
\mathtt{trunc}(h, t_0) ::= \begin{cases} \mathsf{None} & \text{if } h \text{ is deallocated} \\ \mathsf{Some}([\![t \leftarrow h(t) \mid t_0 \leq t \in \mathtt{dom}(h)]\!]) & \text{if } h \text{ is alive} \end{cases}
$$

So $\mathsf{GlobalGhost}(\mathcal{M}, \mathcal{N}, V_{up}, V_{cut})$ contains:

- the authoritative ownership of the upper-bound view V_{up} for all thread-views; and
- the authoritative full fraction ownership of all histories in $\mathcal{M}$, truncated by the cut view V_{cut};[22] and
- the authoritative full fraction ownership of all atomic writes sets, non-atomic reads sets, and atomic reads sets in $\mathcal{N}$; and
- the authoritative full fraction ownership of all blocks in $\mathcal{M}$.

[22] Note that we also need to convert memory values to values, following Definition 3.16.

We use the map insert notation, *e.g.*, $[\ell \leftarrow (1, \mathcal{N}(\ell).\mathsf{nr}) \mid \ell \in \mathtt{dom}(\mathcal{N})]$ to convert the map $\mathcal{N}$ to a map from locations to pairs of fractions and non-atomic reads sets that come from $\mathcal{N}$.

Lemma 7.16 (Agreements between the Global Ghost State and Local Assertions)**.** *The global ghost state ownership* GlobalGhost *and the local assertions satisfy several agreement properties given in Figure 7.8. They are all derived from validity of the corresponding RAs.*

Lemma 7.17 (Updates of the Global Ghost State and Local Assertions)**.** GlobalGhost *can be updated together the local assertions following the rules in Figure 7.9. They are all derived from frame-preserving updates of the corresponding RAs, and the properties of* `trunc`*.*

Most notably, BL-GHOST-UPDATE-HIST-DROP-SINGLETON demonstrates that shrinking a history to a singleton is simply a logical change (a *viewshift*). It is done by bumping the cut view V'_{cut} for ℓ up to its latest timestamp t, which is the input to `trunc`. The rule is used to prove

BL-GHOST-SEEN
$$\mathsf{GlobalGhost}(\mathcal{M}, \mathcal{N}, V_{up}, V_{cut}) * \mathsf{Seen}(\mathcal{V}) \vdash \mathcal{V}.\mathsf{acq} \sqsubseteq V_{up} \land \mathcal{V} \in \mathcal{M}$$

BL-GHOST-HIST
$$\mathsf{GlobalGhost}(\mathcal{M}, \mathcal{N}, V_{up}, V_{cut}) * \mathsf{Hist}_q(\ell, h) \vdash \mathtt{trunc}(\mathcal{M}(\ell), V_{cut}(\ell)) = h \land \ell \notin \mathrm{unalloc}(\mathcal{M})$$

BL-GHOST-AT-WRITE
$$\mathsf{GlobalGhost}(\mathcal{M}, \mathcal{N}, V_{up}, V_{cut}) * \mathsf{Write}_q^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha) \vdash \mathcal{N}(\ell).\mathsf{aw} = \alpha$$

BL-GHOST-NA-READ
$$\mathsf{GlobalGhost}(\mathcal{M}, \mathcal{N}, V_{up}, V_{cut}) * \mathsf{Read}_q^{\mathtt{na}}(\ell, \alpha) \vdash \alpha \subseteq \mathcal{N}(\ell).\mathsf{nr} \land (q = 1 \Rightarrow \mathcal{N}(\ell).\mathsf{nr} = \alpha)$$

BL-GHOST-AT-READ
$$\mathsf{GlobalGhost}(\mathcal{M}, \mathcal{N}, V_{up}, V_{cut}) * \mathsf{Read}_q^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha) \vdash \alpha \subseteq \mathcal{N}(\ell).\mathsf{ar} \land (q = 1 \Rightarrow \mathcal{N}(\ell).\mathsf{ar} = \alpha)$$

BL-GHOST-BLOCK-FULL
$$\mathsf{GlobalGhost}(\mathcal{M}, \mathcal{N}, V_{up}, V_{cut}) * \dagger^n \ell \vdash 0 < n * \forall m.\, (\ell +_\ell m) \in \mathtt{dom}(\mathcal{M}) \Leftrightarrow m \in [0, n)$$

FIGURE 7.8: Several agreements between the global ghost state and local assertions

BL-GHOST-UPDATE-SEEN
$$\frac{V_{up} \sqsubseteq V'_{up} \qquad \mathcal{V}.\mathsf{cur} \sqsubseteq V'_{up} \qquad V'_{up} \in \mathcal{M}}{\mathsf{GlobalGhost}(\mathcal{M}, \mathcal{N}, V_{up}, V_{cut}) \dot{\Rrightarrow} \mathsf{GlobalGhost}(\mathcal{M}, \mathcal{N}, V'_{up}, V_{cut}) * \mathsf{Seen}(\mathcal{V})}$$

BL-GHOST-UPDATE-HIST-DROP-SINGLETON
$$\frac{h(t) = (v, V) \qquad t = \mathtt{max}(\mathtt{dom}(h)) \qquad V'_{cut} = V_{cut}[\ell \leftarrow \{V_{cut}(\ell)\,[\mathsf{w} := t]\}]}{\mathsf{GlobalGhost}(\mathcal{M}, \mathcal{N}, V_{up}, V_{cut}) * \mathsf{Hist}(\ell, h) \dot{\Rrightarrow} \mathsf{GlobalGhost}(\mathcal{M}, \mathcal{N}, V_{up}, V'_{cut}) * \mathsf{Hist}(\ell, [t \leftarrow (v, V)])}$$

BL-GHOST-UPDATE-NA-WRITE
$$\frac{t > \mathtt{max}(\mathtt{dom}(h)) \geq V_{cut}(\ell).\mathsf{w} \qquad \mathcal{M}' = \mathcal{M}[\ell \leftarrow \mathcal{M}(\ell)[t \leftarrow (v, V)]] \qquad \mathcal{N}' = \mathcal{N}[\ell \leftarrow \{\mathcal{N}(\ell)\,[\mathsf{w} := t]\}] \qquad V'_{cut} = V_{cut}[\ell \leftarrow \{V_{cut}(\ell)\,[\mathsf{w} := t]\}]}{\mathsf{GlobalGhost}(\mathcal{M}, \mathcal{N}, V_{up}, V_{cut}) * \mathsf{Hist}(\ell, h) \dot{\Rrightarrow} \mathsf{GlobalGhost}(\mathcal{M}', \mathcal{N}', V_{up}, V'_{cut}) * \mathsf{Hist}(\ell, [t \leftarrow (v, V)])}$$

BL-GHOST-UPDATE-AT-WRITE
$$\frac{t \geq V_{cut}(\ell).\mathsf{w} \qquad \mathcal{M}' = \mathcal{M}[\ell \leftarrow \mathcal{M}(\ell)[t \leftarrow (v, V)]] \qquad \mathcal{N}' = \mathcal{N}[\ell \leftarrow \{\mathcal{N}(\ell)\,[\mathsf{aw} := \mathcal{N}(\ell).\mathsf{aw} \cup \{t\}]\}] \qquad V'_{cut} = V_{cut}[\ell \leftarrow \{V_{cut}(\ell)\,[\mathsf{aw} := V_{cut}(\ell).\mathsf{aw} \cup \{t\}]\}]}{\begin{array}{c}\mathsf{GlobalGhost}(\mathcal{M}, \mathcal{N}, V_{up}, V_{cut}) * \mathsf{Hist}(\ell, h) * \mathsf{Write}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha) \\ \dot{\Rrightarrow} \mathsf{GlobalGhost}(\mathcal{M}', \mathcal{N}', V_{up}, V'_{cut}) * \mathsf{Hist}(\ell, h[t \leftarrow (v, V)]) * \mathsf{Write}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha \cup \{t\})\end{array}}$$

BL-GHOST-UPDATE-NA-READ
$$\frac{\mathcal{N}' = \mathcal{N}[\ell \leftarrow \{\mathcal{N}(\ell)\,[\mathsf{nr} := \mathcal{N}(\ell).\mathsf{nr} \cup \{r\}]\}]}{\mathsf{GlobalGhost}(\mathcal{M}, \mathcal{N}, V_{up}, V_{cut}) * \mathsf{Read}_q^{\mathtt{na}}(\ell, \alpha) \dot{\Rrightarrow} \mathsf{GlobalGhost}(\mathcal{M}, \pi', V_{up}, V_{cut}) * \mathsf{Read}_q^{\mathtt{na}}(\ell, \alpha \cup \{r\})}$$

BL-GHOST-UPDATE-AT-READ
$$\frac{\mathcal{N}' = \mathcal{N}[\ell \leftarrow \{\mathcal{N}(\ell)\,[\mathsf{ar} := \mathcal{N}(\ell).\mathsf{ar} \cup \{r\}]\}]}{\mathsf{GlobalGhost}(\mathcal{M}, \mathcal{N}, V_{up}, V_{cut}) * \mathsf{Read}_q^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha) \dot{\Rrightarrow} \mathsf{GlobalGhost}(\mathcal{M}, \mathcal{N}', V_{up}, V_{cut}) * \mathsf{Read}_q^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha \cup \{r\})}$$

FIGURE 7.9: Several update rules for the global ghost state and local assertions

BL-HIST-DROP-SINGLETON, by hiding the global ghost state GlobalGhost in the state interpretation S, as we will see next.

Definition 7.18 (State Interpretation for the Base Logic)**.**

$$\begin{aligned}\mathsf{GlobalInv}(\varsigma) ::= {} & \exists V_{up}, V_{cut}.\, \mathsf{GlobalGhost}(\varsigma.\mathcal{M}, \varsigma.\mathcal{N}, V_{up}, V_{cut}) * \\ & \varsigma \text{ is wellformed} * V_{up} \in \varsigma.\mathcal{M} * \varsigma.\mathcal{N} \sqsubseteq V_{cut} \\ \mathcal{I}_{\text{HIST}} ::= {} & \boxed{\exists \varsigma.\, \boxed{\circ\, \mathsf{ex}(\varsigma)}^{\gamma_{\text{STATE}}} * \mathsf{GlobalInv}(\varsigma)}^{\mathcal{N}_{\text{HIST}}} \\ S(\varsigma) ::= {} & \boxed{\bullet\, \mathsf{ex}(\varsigma) : \textsc{Auth}(\textsc{Ex}(\mathit{GlobalState}))}^{\gamma_{\text{STATE}}} * \mathcal{I}_{\text{HIST}}\end{aligned}$$

The global physical state ς, together with the logical states V_{up} and V_{cut}, need to satisfy some basic properties, as written in the *global invariant* $\mathsf{GlobalInv}(\varsigma)$: ς is wellformed (Property 3.15), V_{up} is closed in $\varsigma.\mathcal{M}$ (Property 3.12), and the cut view V_{cut} must be at least the race detector view $\mathcal{N}$ to guarantee that accessing some history h is not racy.

The definition of the state interpretation is a bit peculiar. We would simply want $S = \mathsf{GlobalInv}$. But recall that S is only accessible with a weakest pre-condition (Definition 6.14), so if we let $S = \mathsf{GlobalInv}$, we can only prove rules with WPs. This is usually the case: an update to the physical state needs to be done by some instruction, whose rule will come with a WP, by which we can access the state interpretation S and then the global ghost state GlobalGhost, with which we can perform a ghost update to keep the ghost state and the physical state in sync.

However, what if we simply want to change the ghost state without changing the physical state, *i.e.*, a "view shift"? For example, the rule BL-HIST-DROP-SINGLETON is simply a logical move that changes the "view"[23] of the logic from a history h to a singleton, without changing the physical memory for ℓ. To support such rules with not just WPs but also viewshifts, we put GlobalInv inside an invariant with the fixed namespace $\mathcal{N}_{\text{HIST}}$, and employ extra ghost state to maintain that the state ς existentially quantified in the invariant $\mathcal{I}_{\text{HIST}}$ is always exactly the parameter ς of $S(\varsigma)$, which is the actual physical state. The RA $\textsc{Auth}(\textsc{Ex}(\mathit{GlobalState}))$ ensures that the states agree: $\text{valid}(\bullet\, \mathsf{ex}(\varsigma) \cdot \circ\, \mathsf{ex}(\varsigma')) \Rightarrow \varsigma = \varsigma'$.

Note that we also need another global ghost location γ_{STATE}, and for every viewshift $\Rightarrow\!\!\!\!\!\!\Rightarrow_{\mathcal{E}}$ that wants to access GlobalInv, we implicitly assume that $\mathcal{N}_{\text{HIST}} \subseteq \mathcal{E}$ and the invariant $\mathcal{I}_{\text{HIST}}$ is known.

[23] Note how the "view" in viewshifts is different from views of the relaxed memory machine.

7.6 Proofs of Some Primitive Rules and Adequacy

Now, we show proof sketches of some base-logic rules. All rules have been proven and checked by Coq.

Proof sketch of BL-HIST-DROP-SINGLETON *(§7.2).* Assuming $\mathcal{N}_{\text{HIST}} \subseteq \mathcal{E}$, we use INV-ACC (§6.3) to open $\mathcal{I}_{\text{HIST}}$. Note that since the contents of $\mathcal{I}_{\text{HIST}}$ are all timeless, the later we get after opening $\mathcal{I}_{\text{HIST}}$ can be stripped off right away. We then use BL-GHOST-UPDATE-HIST-DROP-SINGLETON to truncate the history with a new cut view $V'_{cut} \sqsupseteq V_{cut} \sqsupseteq \varsigma.\mathcal{N}$. The invariant contents only change in V'_{cut}. We therefore can easily re-establish invariant and close it using the closing wand viewshift we get earlier from INV-ACC. □

Proof sketch of BL-HOARE-ACQ-FENCE *(§7.3.1)*. We start by unfolding the definition of Hoare triples (Definition 6.7 or Notation 7.7), and then of WPs (Definition 6.14). Our goal then looks as follow.

Context:	Goal:
$\mathsf{Seen}(\mathcal{V}) * S(\varsigma)$	${}^{\mathcal{E}}\!\Rightarrow^{\varnothing} (\mathrm{red}((\mathbf{fence}_{\mathbf{acq}}, \mathcal{V}), \varsigma) * \forall(e', \mathcal{V}').\ \ldots)$

Since we have a fancy update ${}^{\mathcal{E}}\!\Rightarrow^{\varnothing}$ in the goal, we can open $\mathcal{I}_{\text{HIST}}$:

$\boxed{\bullet\,\mathsf{ex}(\varsigma)}^{\gamma_{\text{STATE}}} * \boxed{\circ\,\mathsf{ex}(\varsigma)}^{\gamma_{\text{STATE}}} * \mathsf{GlobalInv}(\varsigma) * (\ldots \; {}^{\varnothing}\!\Rrightarrow^{\mathcal{E}} \ldots)$

The obligation $\mathrm{red}((\mathbf{fence}_{\mathbf{acq}}, \mathcal{V}), \varsigma)$ is easily discharged.

For the remaining goal, after introducing assumptions, we know

(i) $\varsigma \mid (\mathbf{fence}_{\mathbf{acq}}, \mathcal{V}) \xrightarrow{\mathrm{F}^{\mathbf{acq}}, \bot}_{\mathrm{t}} \varsigma \mid (\textcircled{\scriptsize t}, \mathcal{V}')$

and the goal is $\quad \triangleright\, {}^{\varnothing}\!\Rightarrow^{\mathcal{E}} (S(\varsigma) * \mathcal{V}' \sqsubseteq \mathcal{V} * \mathsf{Seen}(\mathcal{V}') * \mathcal{V}'.\mathsf{cur} = \mathcal{V}'.\mathsf{acq})$

Using BL-GHOST-SEEN, we know $\mathcal{V}.\mathsf{acq} \sqsubseteq V_{up}$, so we can use BL-GHOST-UPDATE-SEEN to get $\mathsf{Seen}(\mathcal{V}')$ without updating V_{up}.

We can then use the closing wand viewshift $(\ldots \; {}^{\varnothing}\!\Rrightarrow^{\mathcal{E}} \ldots)$ to close the invariant without updating ς. The goal is now:

$S(\varsigma) * \mathsf{Seen}(\mathcal{V}')$	$S(\varsigma) * \mathcal{V}' \sqsubseteq \mathcal{V} * \mathsf{Seen}(\mathcal{V}') * \mathcal{V}'.\mathsf{cur} = \mathcal{V}'.\mathsf{acq}$

This is easily done by looking at the reduction (i) for acquire fences. □

Proof sketch of BL-HOARE-WRITE-AT *(§7.3.2)*. We have the following assumptions: (1) $\mathbf{rlx} \sqsubseteq o$, (2) $\mathsf{Local}_{\mathrm{A}}(\ell, h, \mathcal{V}.\mathsf{cur})$, (3) $\mathsf{Local}^{\mathbf{na}}_{\mathrm{R}}(\ell, \alpha_r, \mathcal{V}.\mathsf{cur})$, and (4) $\mathsf{Local}^{\sqsupseteq \mathbf{rlx}}_{\mathrm{W}}(\ell, \alpha_w, V_w)$. After unfolding Hoare triple and WP definitions, we have the following goal.

Context:	Goal:
$\mathsf{Seen}(\mathcal{V}) * \mathsf{Hist}(\ell, h) * \mathsf{Write}^{\sqsupseteq \mathbf{rlx}}(\ell, \alpha_w) * \mathsf{Read}^{\mathbf{na}}(\ell, \alpha_r)$	
$S(\varsigma)$	${}^{\mathcal{E}}\!\Rightarrow^{\varnothing} (\mathrm{red}((\ell :=_o v', \mathcal{V}), \varsigma) * \forall(e', \mathcal{V}').\ \ldots)$

We unfold S and open the invariant $\mathcal{I}_{\text{HIST}}$:

$\boxed{\bullet\,\mathsf{ex}(\varsigma)}^{\gamma_{\text{STATE}}} * \boxed{\circ\,\mathsf{ex}(\varsigma)}^{\gamma_{\text{STATE}}} * \mathsf{GlobalInv}(\varsigma) * (\ldots \; {}^{\varnothing}\!\Rrightarrow^{\mathcal{E}} \ldots)$

We first show safety: $\quad \mathrm{red}((\ell :=_o v', \mathcal{V}), \varsigma)$

By BL-GHOST-HIST and BL-GHOST-NA-READ, we have

$\mathtt{trunc}(\varsigma.\mathcal{M}(\ell), V_{cut}(\ell)) = h \wedge \ell \notin \mathrm{unalloc}(\varsigma.\mathcal{M}) \wedge \varsigma.\mathcal{N}(\ell).\mathsf{nr} = \alpha_r.$

Combining these with (2), (3), and $\varsigma.\mathcal{N} \sqsubseteq V_{cut}$, we have

$\varsigma.\mathcal{N}(\ell).\mathsf{w} \le \mathcal{V}.\mathsf{cur}(\ell).\mathsf{w} \wedge \varsigma.\mathcal{N}(\ell).\mathsf{nr} \sqsubseteq \mathcal{V}.\mathsf{cur}(\ell).\mathsf{nr}.$

So we satisfy DRF-WRITE-AT (§3.4), *i.e.*, we are race-free.

Consequently, we satisfy OC-MEM (§4.3), so we are done.

For the remaining goal, after introducing assumptions, we know

(i) $\varsigma \mid (\ell :=_o v', \mathcal{V}) \xrightarrow{\mathrm{W}^o(\ell, v'), \bot}_{\mathrm{t}} \varsigma' \mid (\textcircled{\scriptsize t}, \mathcal{V}')$

and the goal is

$$\triangleright\, {}^{\varnothing}\!\Rightarrow^{\mathcal{E}} \begin{pmatrix} S(\varsigma') * \exists t' \notin \mathtt{dom}(h), V'.\, \mathcal{V} \xrightarrow{\mathrm{W}:o,\ell,t',\bot,V'} \mathcal{V}' * \\ \mathsf{Local}^{\sqsupseteq \mathbf{rlx}}_{\mathrm{W}}(\ell, \alpha_w \cup \{t'\}, V_w \sqcup \mathcal{V}'.\mathsf{cur}) * \\ \mathsf{Seen}(\mathcal{V}') * \mathsf{Hist}(\ell, h[t' \leftarrow (v', V')]) * \\ \mathsf{Write}^{\sqsupseteq \mathbf{rlx}}(\ell, \alpha_w \cup \{t'\}) * \mathsf{Read}^{\mathbf{na}}(\ell, \alpha_r) \end{pmatrix}$$

By looking at the reduction (i), we can get the new timestamp t' and the write message view V', and the fact $\mathcal{V} \xrightarrow{\mathtt{W}:o,\ell,t',\bot,V'} \mathcal{V}'$, and that $\mathcal{N}' = \mathcal{N}[\ell \leftarrow \{\mathcal{N}(\ell)\,[\mathsf{aw} := \mathcal{N}(\ell).\mathsf{aw} \cup \{t\}]\}]$.

Additionally, by BL-GHOST-AT-WRITE we have $\varsigma.\mathcal{N}(\ell).\mathsf{aw} = \alpha_w$. So we can pick at new cut view V'_{cut}, and use BL-GHOST-UPDATE-AT-WRITE to update both the history ownership to $\mathsf{Hist}(\ell, h[t' \leftarrow (v', V')])$ and the atomic write ownership to $\mathsf{Write}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha_w \cup \{t'\})$. By using (4), we also discharge the local observation $\mathsf{Local}^{\sqsupseteq \mathtt{rlx}}_{\mathtt{W}}(\ell, \alpha_w \cup \{t'\}, V_w \sqcup \mathcal{V}'.\mathsf{cur})$. Using BL-GHOST-UPDATE-SEEN, we also get $\mathsf{Seen}(\mathcal{V}')$. The non-atomic read ownership $\mathsf{Read}^{\mathtt{na}}(\ell, \alpha_r)$ is returned as-is.

We eventually end up with the following goal.

$$\boxed{\bullet\,\mathsf{ex}(\varsigma)}^{\gamma_{\text{STATE}}} * \boxed{\circ\,\mathsf{ex}(\varsigma)}^{\gamma_{\text{STATE}}} * \mathsf{GlobalInv}(\varsigma') * (\ldots \overset{\varnothing}{\Rrightarrow}\!\!\!\!*^{\mathcal{E}} \ldots) \\ \twoheadrightarrow \overset{\varnothing}{\Rrightarrow}^{\mathcal{E}} S(\varsigma')$$

This is easily done by first updating the ghost ownership $\boxed{\bullet\,\mathsf{ex}(\varsigma)}^{\gamma_{\text{STATE}}} * \boxed{\circ\,\mathsf{ex}(\varsigma)}^{\gamma_{\text{STATE}}}$ to that of ς', and then use the closing wand viewshift to close the invariant $\mathcal{I}_{\text{HIST}}$. □

Finally, we show adequacy for our base logic, which is slightly difference from Theorem 6.8 in that we do not fix the initial state.

Theorem 7.19 (Base Logic Adequacy)**.** *Assuming that the state ς is wellformed and a thread-view $\mathcal{V}$ is closed in $\varsigma.\mathcal{M}$ ($\mathcal{V} \in \varsigma.\mathcal{M}$), if $\vdash \mathit{wp}_{\top}\,(e, \mathcal{V})\,\{(v, \_).\,\phi(v)\}$ is derivable in the base logic for λ_{Rust} + ORC11 where $\phi(v)$ is a pure (meta-level) fact, then the following holds.*

$$\forall \pi, \mathcal{T}', \varsigma'.\,([\pi \mapsto (e, \mathcal{V})], \varsigma) \to^* (\mathcal{T}', \varsigma') \Rightarrow \\ \forall v.\,\mathcal{T}(\pi) = v \Rightarrow \phi(v) \qquad \text{(BL-ADEQUACY-VAL)}$$

$$\wedge \quad \forall \rho, e_\rho, \mathcal{V}_\rho.\,\mathcal{T}(\rho) = (e_\rho, \mathcal{V}_\rho) \Rightarrow \big(e_\rho \textit{ is a value} \vee \mathrm{red}((e_\rho, \mathcal{V}_\rho), \varsigma')\big) \qquad \text{(BL-ADEQUACY-NO-STUCK)}$$

Proof. The proof follows from a Iris-provided adequacy theorem (that also implies Theorem 6.8). All we need to do is to allocate the various global ghost locations with the correct RAs, and establish the global invariant $\mathcal{I}_{\text{HIST}}$ of the state interpretation $S(\varsigma)$, which requires ς's well-formedness and that $\mathcal{V} \in \varsigma.\mathcal{M}$. □

CHAPTER SUMMARY. In this chapter, we demonstrated the instantiation of Iris with the λ_{Rust} + ORC11 language to achieve a RMC base logic. The most important feature of the logic is the explicit use of thread-views in conjunction with various local assertions to achieve abstraction for the relaxed memory effects. In the next chapter, we provide the next abstractions for thread-views. In Chapter 9 and Chapter 10, we will provide more abstractions for the local assertions.

8

vProp: *View-monotone Predicates*

Following iGPS[1], in this chapter we introduce an abstraction to hide thread-views in the base logic, and lift the base logic to a surface-level logic whose propositions have the type vProp, which stands for *view propositions*. We call this surface logic of vProp the iRC11 logic. We will, in this chapter as well as later ones, develop more reasoning principles for iRC11, within iRC11 itself or on top of the base logic.

[1] Kaiser et al., "Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris" [Kai+17].

The motivation for hiding thread-views and views is that most of the time, they do not have interesting behaviors, and when they do (in the relaxed memory operations), the effects are usually that the thread-views or views have grown in certain ways. If we can provide new assertions that abstract those ways that views change (*e.g.*, an observation of a value written or read, or that a thread-view's current view has been upgraded to its acquire view), then we can achieve SC-like rules that have been developed in many previous logics.[2] In this chapter, we establish some of such core rules for iRC11.

[2] Vafeiadis and Narayan, "Relaxed separation logic: a program logic for C11 concurrency" [VN13]; Doko and Vafeiadis, "A Program Logic for C11 Memory Fences" [DV16]; Doko and Vafeiadis, "Tackling Real-Life Relaxed Concurrency with FSL++" [DV17]; Turon et al., "GPS: navigating weak memory with ghosts, protocols, and separation" [TVD14].

Note, however, that hiding views is an abstraction that *weaken* the logic. While such abstraction is sufficient in many cases, views are inevitable in order to provide strong reasoning principles or specifications for very relaxed algorithms. This observation has been made by the $\mathsf{RB}_{\mathtt{rlx}}$ work (Part III), the Cosmo logic,[3] and the Compass specifications (Part IV), chronologically. §8.5 will introduce several modalities to restore explicit view reasoning in the logic of vProp.

[3] Mével et al., "Cosmo: a concurrent separation logic for multicore OCaml" [MJP20].

8.1 View-monotone Predicates

We define vProp as the type of view-monotone predicates over iProp.

Definition 8.1 (vProp)**.**

$$\mathsf{vProp} ::= \mathit{View} \xrightarrow{\mathsf{mon}} \mathsf{iProp}$$

$$\text{satisfying } \forall P : \mathsf{vProp}.\, \forall V, V'.\, V \sqsubseteq V' \Rightarrow P(V) \Rightarrow P(V') \quad (\textsc{vprop-mono})$$

An assertion $P : \mathsf{vProp}$ is to be interpreted as some resource that holds at a simple view. This view usually is the current component $\mathcal{V}.\mathsf{cur}$ of a thread π's thread-view $\mathcal{V}$ in case P is owned locally by the thread π; or a view of some write message m in case we attach P to the message m in order to transfer P from m's writer to its readers.

Another choice is to define vProp as a predicate of thread-views (*ThreadView* $\xrightarrow{\text{mon}}$ iProp), but such a definition does not have an actual use. Thread-views are tied locally to threads, and so such a definition is not suitable to represent resources that are not tied to a thread, but instead, for example, are tied to a message, or are put inside a shared invariant. Furthermore, resources are typically not tied to a whole thread-view $\mathcal{V}$, but rather to one of its components (the release-fence, current, or acquire view). In short, simple view predicates are used pervasively, while thread-view predicates are not.

The monotonicity requirement is needed to maintain "stability" of the frame when the view grows. That is, more observations made by a thread's step should not invalidate resources that are not relevant to the step. As a result, we explicitly monotonize vProp propositions when necessary (*i.e.*, when they are not already monotone). Note that this is a source of weakening the surface logic compared to the base logic.

We lift the many logical connectives and modalities of the base logic (which we inherit from Iris) straightforwardly.

Definition 8.2 (Model of vProp propositions)**.** The function $[\![\cdot]\!]$ provides a model of vProp propositions into predicates from views to iProp, embedding proofs that the predicates are monotone.[4]

$$
\begin{aligned}
[\![\phi]\!] &::= \lambda\_.\ \phi \\
[\![\mathsf{False}]\!] &::= \lambda\_.\ \mathsf{False} \\
[\![\mathsf{True}]\!] &::= \lambda\_.\ \mathsf{True} \\
[\![P \Rightarrow Q]\!] &::= \lambda V.\ \forall V' \sqsupseteq V.\ [\![P]\!](V') \Rightarrow [\![Q]\!](V') \\
[\![P \wedge Q]\!] &::= \lambda V.\ [\![P]\!](V) \wedge [\![Q]\!](V) \\
[\![P \vee Q]\!] &::= \lambda V.\ [\![P]\!](V) \vee [\![Q]\!](V) \\
[\![P * Q]\!] &::= \lambda V.\ [\![P]\!](V) * [\![Q]\!](V) \\
[\![P \mathrel{-\!\!*} Q]\!] &::= \lambda V.\ \forall V' \sqsupseteq V.\ [\![P]\!](V') \mathrel{-\!\!*} [\![Q]\!](V') \\
[\![\exists x.\ P]\!] &::= \lambda V.\ \exists x.\ [\![P]\!](V) \\
[\![\forall x.\ P]\!] &::= \lambda V.\ \forall x.\ [\![P]\!](V) \\
[\![\triangleright P]\!] &::= \lambda V.\ \triangleright [\![P]\!](V) \\
[\![\Box P]\!] &::= \lambda V.\ \Box [\![P]\!](V) \\
[\![\boxed{a}^{\gamma}]\!] &::= \lambda\_.\ \boxed{a}^{\gamma} \\
[\![\dot{\Rrightarrow} P]\!] &::= \lambda V.\ \dot{\Rrightarrow} [\![P]\!](V) \\
[\![{}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2} P]\!] &::= \lambda V.\ {}^{\mathcal{E}_1}\Rrightarrow^{\mathcal{E}_2} [\![P]\!](V) \\
&\dots
\end{aligned}
$$

[4] This model is also useful elsewhere, and has been generalized in Iris to *monotone predicates* over types that come with a partial order.

Note that, $[\![P * Q]\!]$, for example, is view-monotone assuming P and Q are vProp and thus view-monotone. On the other hand, $[\![P \mathrel{-\!\!*} Q]\!]$ needs to be monotonized explicitly. The model of ghost state ownership $[\![\boxed{a}^{\gamma}]\!]$ interestingly simply ignores the input view.

Lemma 8.3 (Properties of iRC11 connectives)**.** *The properties in Figure 6.2, Figure 6.3, Figure 6.4, and Figure 6.7 are preserved for* iRC11 *connectives by the encoding of Definition 8.2.*

The most interesting encodings are those of weakest pre-conditions (from which Hoare triples are derived similarly as before), non-atomic and atomic points-to assertions, and invariants. We will discuss non-atomic points-to in Chapter 9, atomic points-to in Chapter 10, and invariants in Chapter 11. In the remaining, we discuss the models of iRC11 (vProp) weakest pre-conditions and several RMC-specific modalities.

8.2 Model of iRC11 Weakest Pre-conditions

iRC11 WPs are a vProp proposition that is built upon the base logic WPs, and that hides away thread-views. Nevertheless, we need a way to refer to the thread-view, specifically to define the *release* and *acquire* modalities (§8.5) that provide the abstraction of fence behaviors. For this purpose, we expose thread-ids in the WP definition, which are used under the hood to associate with thread-views. Fortunately, these iRC11-level thread-ids need not be the same thread-ids of the threadpool, so we can simply store thread-views in ghost state—with the RA $\text{TVIEWR} = \text{AUTH}(\text{LAT}(\textit{ThreadView}))$, and use ghost locations as thread-ids.

Definition 8.4 (iRC11 Weakest Pre-conditions)**.**

$$\begin{aligned}&[\![\mathsf{wp}_{\mathcal{E}}\, e \,\mathsf{in}\, \pi\, \{\Phi\}]\!] ::= \\ &\quad \lambda V.\, \forall \mathcal{V}.\, V \sqsubseteq \mathcal{V}.\mathsf{cur} \mathrel{-\!\!*} \boxed{\bullet \mathcal{V} : \text{TVIEWR}}^{\pi} \mathrel{-\!\!*} \mathsf{Seen}(\mathcal{V}) \mathrel{-\!\!*} \\ &\qquad \mathsf{wp}_{\mathcal{E}}\, (e, \mathcal{V}) \left\{ (v, \mathcal{V}').\, \boxed{\bullet \mathcal{V}'}^{\pi} * [\![\Phi(v)]\!](\mathcal{V}'.\mathsf{cur}) \right\}\end{aligned}$$

The WP definition takes care of several things.

- It makes sure that the definition is view-monotone explicitly, by requiring that the underlying base logic WP take a thread-view $\mathcal{V}$ whose current component $\mathcal{V}.\mathsf{cur}$ is in the upward closure of the input view V.
- It threads through the authoritative ghost ownership $\boxed{\bullet \mathcal{V}}^{\pi}$ of the thread-view being executed with the expression e, in the pre- and post-conditions. This also allows for creating snapshots (fragmentary ownership) $\boxed{\bullet \mathcal{V}}^{\pi}$ for the lower bound of the executing thread π's thread-view, which in turn will be used to define release and acquire modalities.
- By hiding thread-views, it also hides the assertion $\mathsf{Seen}(\mathcal{V})$. Accordingly, it also provides $\mathsf{Seen}(\mathcal{V})$ as assumption to the base logic WP. It does not require $\mathsf{Seen}(\mathcal{V}')$ in the post-condition, because this can be easily obtained from the state interpretation S (hidden in the base logic WP) using BL-GHOST-UPDATE-SEEN.

Definition 8.5 (iRC11 Hoare triples)**.** iRC11 Hoare triples are defined similarly as before.

$$\{P\}\, e \,\mathsf{in}\, \pi\, \{v.\, Q\}_{\mathcal{E}} ::= \square(P \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\, e \,\mathsf{in}\, \pi\, \{v.\, Q\})$$

If we interpret this definition in iProp, we will arrive at the following.

$$\begin{aligned}&[\![\{P\}\, e \,\mathsf{in}\, \pi\, \{v.\, Q\}_{\mathcal{E}}]\!] ::= \\ &\quad \lambda V.\ \Box \forall V', \mathcal{V}.\, V \sqsubseteq V' \sqsubseteq \mathcal{V}.\mathsf{cur} \mathrel{-\!\!*} [\![P]\!](V) \mathrel{-\!\!*} \boxed{\bullet \mathcal{V}}^{\pi} \mathrel{-\!\!*} \mathsf{Seen}(\mathcal{V}) \\ &\qquad \mathrel{-\!\!*} \mathsf{wp}_{\mathcal{E}}\,(e, \mathcal{V}) \left\{(v, \mathcal{V}').\, \boxed{\bullet \mathcal{V}'}^{\pi} * [\![Q]\!](\mathcal{V}'.\mathsf{cur})\right\}\end{aligned}$$

As one can see, generally both the pre-condition P and post-condition Q are interpreted at the current components $\mathcal{V}.\mathsf{cur}$ and $\mathcal{V}'.\mathsf{cur}$, respectively, of the executing thread's thread-view.

Lemma 8.6 (Properties of iRC11 WPs and Hoare triples)**.** *The properties in Figure 6.5 and Figure 6.6, except those concerning invariants which are not yet defined, also hold for* iRC11 *WPs and Hoare triples.*

Theorem 8.7 (iRC11 Adequacy)**.** *If* $\forall \pi.\ \vdash \mathit{wp}_{\top}\, e \,\mathsf{in}\, \pi\, \{v.\, \phi(v)\}$ *is derivable in the* iRC11 *logic for* λ_{Rust} + ORC11 *where* $\phi(v)$ *is a pure (meta-level) fact, then the following holds.*

$$\begin{aligned}&\forall \pi, \mathcal{T}', \varsigma'.\, ([\pi \mapsto (e, \mathcal{V}_{\mathrm{init}})], \varsigma_{\mathrm{init}}) \to^* (\mathcal{T}', \varsigma') \Rightarrow \\ &\qquad \forall v.\, \mathcal{T}(\pi) = v \Rightarrow \phi(v) \qquad \text{(ADEQUACY-VAL)} \\ &\quad \wedge \quad \forall \rho, e_\rho, \mathcal{V}_\rho.\, \mathcal{T}(\rho) = (e_\rho, \mathcal{V}_\rho) \Rightarrow \big(e_\rho \textit{ is a value} \vee \mathrm{red}((e_\rho, \mathcal{V}_\rho), \varsigma')\big) \\ &\qquad\qquad \text{(ADEQUACY-NO-STUCK)}\end{aligned}$$

where $\mathcal{V}_{\mathrm{init}} = (\varnothing, \varnothing, \varnothing, \varnothing)$ *and* $\varsigma_{\mathrm{init}} = (\varnothing, \varnothing)$.

Proof sketch. The proof follows from the base logic adequacy (Theorem 7.19). Note that the initial thread-view $\mathcal{V}_{\mathrm{init}}$ and state $\varsigma_{\mathrm{init}}$ are wellformed and $\mathcal{V}_{\mathrm{init}} \in \varsigma_{\mathrm{init}}$. We then need to allocate the ghost state $\boxed{\bullet \mathcal{V}}^{\pi}$ for some π and $\mathcal{V}$, and get $\mathsf{Seen}(\mathcal{V})$, so that we can instantiate and apply our assumption $\forall \pi.\ \vdash \mathsf{wp}_{\top}\, e \,\mathsf{in}\, \pi\, \{v.\, \phi(v)\}$ and finish the proof. □

8.3 Fence Modalities

To model the effects of relaxed accesses and fences, iRC11 inherits two modalities from FSL[5]—the *release* modality Δ and the *acquire* modality ∇—which allow us to talk about ownership of resources at a thread's release-fence or acquire views. The assertion $\Delta_\pi\, P$ represents ownership of P at thread π's release-fence view, while the assertion $\nabla_\pi\, P$ represents ownership of P at thread π's acquire view.

[5] Doko and Vafeiadis, "A Program Logic for C11 Memory Fences" [DV16]; Doko and Vafeiadis, "Tackling Real-Life Relaxed Concurrency with FSL++" [DV17].

The motivation for these modalities as follows. Recall the Message-Passing example using a pair of a relaxed write and a relaxed read, together with fences (Example 2.1(d), Figure 2.2(d)). We have some resource described by the proposition P that we want to transfer from the left-hand thread π to the right-hand thread ρ. However, when the "producer" thread π performs its relaxed write, the message view of that write is drawn from π's release-fence view, not its current view. Hence, we need a way of insisting (in the precondition of the relaxed write) that the P that π is sending holds under its release-fence view—that is what is denoted by $\Delta_\pi\, P$. Dually, when the "consumer" thread ρ performs its relaxed read, the message view it reads will only be joined into its

HOARE-REL-FENCE
$\{P\}\ \mathbf{fence_{rel}}\ \mathsf{in}\ \pi\ \{\Delta_\pi\, P\}$

HOARE-ACQ-FENCE
$\{\nabla_\pi\, P\}\ \mathbf{fence_{acq}}\ \mathsf{in}\ \pi\ \{P\}$

HOARE-REL-FENCE-ELIM
$$\frac{\{P\}\, e\ \mathsf{in}\ \pi\ \{\Phi\}}{\{\Delta_\pi\, P\}\, e\ \mathsf{in}\ \pi\ \{\Phi\}}$$

HOARE-ACQ-FENCE-INTRO
$$\frac{\{\nabla_\pi\, P\}\, e\ \mathsf{in}\ \pi\ \{\Phi\}}{\{P\}\, e\ \mathsf{in}\ \pi\ \{\Phi\}}$$

RELMOD-GHOST
$\Delta_\pi \boxed{a}^\gamma \vdash \boxed{a}^\gamma$

ACQMOD-GHOST
$\nabla_\pi \boxed{a}^\gamma \vdash \boxed{a}^\gamma$

GHOST-RELMOD
$\boxed{a}^\gamma \Rrightarrow \Delta_\pi \boxed{a}^\gamma$

GHOST-ACQMOD
$\boxed{a}^\gamma \Rrightarrow \nabla_\pi \boxed{a}^\gamma$

RELMOD-MONO
$$\frac{P \vdash Q}{\Delta_\pi\, P \vdash \Delta_\pi\, Q}$$

RELMOD-PURE
$\Delta_\pi\, \phi \vdash \phi$

RELMOD-AND
$\Delta_\pi\,(P \wedge Q) \vdash \Delta_\pi\, P \wedge \Delta_\pi\, Q$

RELMOD-OR
$\Delta_\pi\,(P \vee Q) \dashv\vdash \Delta_\pi\, P \vee \Delta_\pi\, Q$

RELMOD-FORALL
$\Delta_\pi\, \forall x.\, P \vdash \forall x.\, \Delta_\pi\, P$

RELMOD-EXIST
$\Delta_\pi\, \exists x.\, P \dashv\vdash \exists x.\, \Delta_\pi\, P$

RELMOD-SEP
$\Delta_\pi\,(P * Q) \dashv\vdash \Delta_\pi\, P * \Delta_\pi\, Q$

RELMOD-WAND
$\Delta_\pi\,(P \mathrel{-\!\!*} Q) \vdash \Delta_\pi\, P \mathrel{-\!\!*} \Delta_\pi\, Q$

RELMOD-LATER-INTRO
$\triangleright\, \Delta_\pi\, P \Rrightarrow_{\mathcal{E}} \Delta_\pi\, \triangleright P$

RELMOD-UNOPS
$$\frac{\cdot \in \left\{\Box, \triangleright, \dot{\Rrightarrow}, {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2}\right\}}{\Delta_\pi \cdot P \vdash \cdot\, \Delta_\pi\, P}$$

ACQMOD-MONO
$$\frac{P \vdash Q}{\nabla_\pi\, P \vdash \nabla_\pi\, Q}$$

ACQMOD-PURE
$\nabla_\pi\, \phi \vdash \phi$

ACQMOD-AND
$\nabla_\pi\,(P \wedge Q) \vdash \nabla_\pi\, P \wedge \nabla_\pi\, Q$

ACQMOD-OR
$\nabla_\pi\,(P \vee Q) \dashv\vdash \nabla_\pi\, P \vee \nabla_\pi\, Q$

ACQMOD-FORALL
$\nabla_\pi\, \forall x.\, P \vdash \forall x.\, \nabla_\pi\, P$

ACQMOD-EXIST
$\nabla_\pi\, \exists x.\, P \dashv\vdash \exists x.\, \nabla_\pi\, P$

ACQMOD-SEP
$\nabla_\pi\,(P * Q) \dashv\vdash \nabla_\pi\, P * \nabla_\pi\, Q$

ACQMOD-WAND
$\nabla_\pi\,(P \mathrel{-\!\!*} Q) \vdash \nabla_\pi\, P \mathrel{-\!\!*} \nabla_\pi\, Q$

ACQMOD-LATER-INTRO
$\triangleright\, \nabla_\pi\, P \Rrightarrow_{\mathcal{E}} \nabla_\pi\, \triangleright P$

ACQMOD-UNOPS
$$\frac{\cdot \in \left\{\Box, \triangleright, \dot{\Rrightarrow}, {}^{\mathcal{E}_1}\!\Rrightarrow^{\mathcal{E}_2}\right\}}{\nabla_\pi \cdot P \vdash \cdot\, \nabla_\pi\, P}$$

FIGURE 8.1: iRC11 rules for fence modalities

acquire view, not its current view. Hence, we need a way of insisting (in the post-condition of the relaxed read) that ρ only receives ownership of P under its acquire view—that is what is denoted by $\nabla_\rho\, P$. We will see how this is materialized in the iRC11 rules for atomic operations in Chapter 10.

Of course, we need a way of actually *introducing* $\Delta_\pi\, P$ and *eliminating* $\nabla_\rho\, P$. These steps are achieved by rules HOARE-REL-FENCE and HOARE-ACQ-FENCE (Figure 8.1), which allow one to transfer any proposition *into the release modality* at the point of a **rel** fence, or *out of the acquire modality* at the point of an **acq** fence, because those are the points where the current and release-fence/acquire views get synchronized.

HOARE-REL-FENCE-ELIM and HOARE-ACQ-FENCE-INTRO are the reverse of HOARE-REL-FENCE and HOARE-ACQ-FENCE, and demonstrate that the release-fence view is included in the current view, which in turn is included in the acquire view of a thread. So $\Delta_\pi\, P$ can be easily turned into P, which can be turned into $\nabla_\pi\, P$. We note that we need a goal in form of a WP or a Hoare triple to perform these moves, but this is only an artifact of

our simple model of fence modalities (§8.3.1).

RELMOD-GHOST, ACQMOD-GHOST, GHOST-RELMOD, and GHOST-ACQMOD together state that the ghost ownership assertion $\boxed{a}^\gamma$ can move freely in and out of the fence modalities. Intuitively, ghost state belongs to the class of *view-agnostic* assertions, in the sense that their ownership interpretation is *not tied to any view at all!* Since $\boxed{a}^\gamma$ is view-agnostic and thus does not care at which view it is interpreted,[6] it is equivalent to $\Delta_\pi \boxed{a}^\gamma$ or $\nabla_\pi \boxed{a}^\gamma$. As a result, $\boxed{a}^\gamma$ can be transferred from one thread to another *without the need for physical synchronization*—in particular, without the need for release/acquire fences.

[6] This can be seen clearly in the model of iRC11 ghost ownership, in Definition 8.2.

The remaining rules in Figure 8.1 state various properties between the fence modalities and other modalities. Some of the rules only have one direction or need to use basic or fancy updates. This is due to our simple model of fence modalities—which we will see next—but fortunately they do not cause any problem in practice.

8.3.1 *Model of the Fence Modalities*

We rely on the extra ghost state of the RA TVIEWR that we have added to iRC11 WPs (Definition 8.4) to get access to the executing thread's hidden thread-view, so that we can give a model to our fence modalities.

Definition 8.8 (Model of the Fence Modalities)**.**

$$[\![\Delta_\pi\, P]\!] ::= \lambda\_.\, \exists \mathcal{V}.\, \boxed{\circ\, \mathcal{V}}^\pi * [\![P]\!](\mathcal{V}.\mathsf{frel}) \qquad \text{(RELMOD-MODEL)}$$

$$[\![\nabla_\pi\, P]\!] ::= \lambda\_.\, \exists \mathcal{V}.\, \boxed{\circ\, \mathcal{V}}^\pi * [\![P]\!](\mathcal{V}.\mathsf{acq}) \qquad \text{(ACQMOD-MODEL)}$$

Note that due to validity of TVIEWR, from $\boxed{\bullet\, \mathcal{V}'}^\pi * \boxed{\circ\, \mathcal{V}}^\pi$, we know that $\mathcal{V} \sqsubseteq \mathcal{V}'$. Consequently, if we own $\Delta_\pi\, P$ in a goal of a WP for the thread π, we know that P holds at the view $\mathcal{V}.\mathsf{frel}$ where $\mathcal{V} \sqsubseteq \mathcal{V}'$ and $\mathcal{V}'$ is π's actual thread-view, and thus by view-monotonicity, P also holds at $\mathcal{V}'.\mathsf{frel}$. In fact, let us sketch the proofs of some of the rules in Figure 8.1.

Proof of HOARE-REL-FENCE. We prove the rule in the base logic. After unfolding the Hoare triples (interpreting them in the base logic, as in Definition 8.5), we have the following goal.

Context:	Goal:
$V \sqsubseteq \mathcal{V}.\mathsf{cur} * [\![P]\!](V) * \boxed{\bullet\, \mathcal{V}}^\pi * \mathsf{Seen}(\mathcal{V})$	$\mathsf{wp}_{\mathcal{E}}\, (e, \mathcal{V}) \left\{ (☣, \mathcal{V}').\, \boxed{\bullet\, \mathcal{V}'}^\pi * [\![\Delta_\pi\, P]\!](\mathcal{V}'.\mathsf{cur}) \right\}$

We then apply WP-MONO (§6.6) and BL-HOARE-REL-FENCE (§7.3.1).

The goal, after unfolding the model of the release modality, is now:

$V \sqsubseteq \mathcal{V}.\mathsf{cur} \wedge \mathcal{V} \sqsubseteq \mathcal{V}' \wedge \mathcal{V}'.\mathsf{frel} = \mathcal{V}'.\mathsf{cur}$	
$[\![P]\!](V) * \boxed{\bullet\, \mathcal{V}}^\pi * \mathsf{Seen}(\mathcal{V}')$	$\boxed{\bullet\, \mathcal{V}'}^\pi * \exists \mathcal{V}_0.\, \boxed{\circ\, \mathcal{V}_0}^\pi * [\![P]\!](\mathcal{V}_0.\mathsf{frel})$

We update the ghost thread-view of π using AUTH-LAT-UPDATE (§7.4).

We are then left with:

$[\![P]\!](V) * \boxed{\circ\, \mathcal{V}'}^\pi$	$\exists \mathcal{V}_0.\, \boxed{\circ\, \mathcal{V}_0}^\pi * [\![P]\!](\mathcal{V}_0.\mathsf{frel})$

And then:

$[\![P]\!](V)$	$[\![P]\!](\mathcal{V}'.\mathsf{frel})$

But we know that $V \sqsubseteq \mathcal{V}.\mathsf{cur} \sqsubseteq \mathcal{V}'.\mathsf{cur} = \mathcal{V}'.\mathsf{frel}$. By view-monotonicity (VPROP-MONO), we are done. □

Proof of HOARE-ACQ-FENCE. The proof is similar to that of HOARE-REL-FENCE. Eventually we will arrive at the goal $[\![P]\!](\mathcal{V}_0.\mathsf{acq}) \vdash [\![P]\!](\mathcal{V}'.\mathsf{cur})$. But by AUTH-LAT-VALID and BL-HOARE-ACQ-FENCE we know that $\mathcal{V}_0.\mathsf{acq} \sqsubseteq \mathcal{V}'.\mathsf{acq} = \mathcal{V}'.\mathsf{cur}$. Again this is done by VPROP-MONO. □

Proof of RELMOD-GHOST. After unfolding, we arrive at the following base-logic goal: $\exists \mathcal{V}. \boxed{\circ \mathcal{V}}^{\pi} * \boxed{a}^{\gamma} \vdash \boxed{a}^{\gamma}$. This is easily done. □

Proof of GHOST-RELMOD. After unfolding, we arrive at the following base-logic goal: $\boxed{a}^{\gamma} \Rrightarrow \exists \mathcal{V}. \boxed{\circ \mathcal{V}}^{\pi} * \boxed{a}^{\gamma}$. We only need to pick any $\mathcal{V}$ for which we own $\boxed{\circ \mathcal{V}}^{\pi}$, because the ghost ownership does not depend on $\mathcal{V}$. With a basic update, we can get ownership of an RA's unit element. In case of TVIEWR, the unit element is $\circ \varnothing$ for the empty thread-view $\varnothing$. Therefore we can easily get $\boxed{\circ \varnothing}^{\pi}$ and we are done. □

Proof of RELMOD-FORALL. After unfolding, we arrive at the following base-logic goal: $\exists \mathcal{V}. \boxed{\circ \mathcal{V}}^{\pi} * \forall x. [\![P]\!](\mathcal{V}.\mathsf{frel}) \vdash \forall x. \exists \mathcal{V}. \boxed{\circ \mathcal{V}}^{\pi} * [\![P]\!](\mathcal{V}.\mathsf{frel})$. This is easily done.

The result of the unfolding also demonstrates that the reverse direction RELMOD-FORALL is not provable for our simple model of the release modality: we would need to go from a $\forall\exists$ assumption to a $\exists\forall$ goal. □

8.4 Objective Propositions and The Objective Modality

We previously mentioned that ghost state ownership belongs to the class of view-agnostic propositions whose interpretations are not tied to any view at all. That is, relaxed memory has no effects on them. We formally call this class *objective* propositions, because they hold regardless of any subjective views of any threads in the program. They are thus important to establish global consensus among concurrent threads.

Definition 8.9 (Objective Propositions)**.** A proposition $P : \mathsf{vProp}$ is objective if its interpretation does not depend any view. [7]

$$\mathsf{objective}(P) ::= \forall V, V'.\, [\![P]\!](V) \vdash [\![P]\!](V')$$

[7] Note that objectivity and the objective modality have also been generalized in Iris for monotone predicates, not just vProp.

Definition 8.10 (The Objective Modality)**.** The objective modality carries the proof that some resource P holds at any view.

$$[\![\langle \mathsf{obj} \rangle\, P]\!] ::= \lambda\_.\, \forall V.\, [\![P]\!](V)$$

Figure 8.2 presents many rules for objective propositions and the objective modality. Unsurprisingly, pure facts, True, False, ghost ownership, and a resource under the objective modality are all objective. Objectivity is maintained structurally, but it is not always so for the objective modality, due to our use of a universal quantifier ($\forall$) in its model. OBJMOD-INTRO allows one to put objective propositions under the objective modality, so that one can store the meta-level objectivity

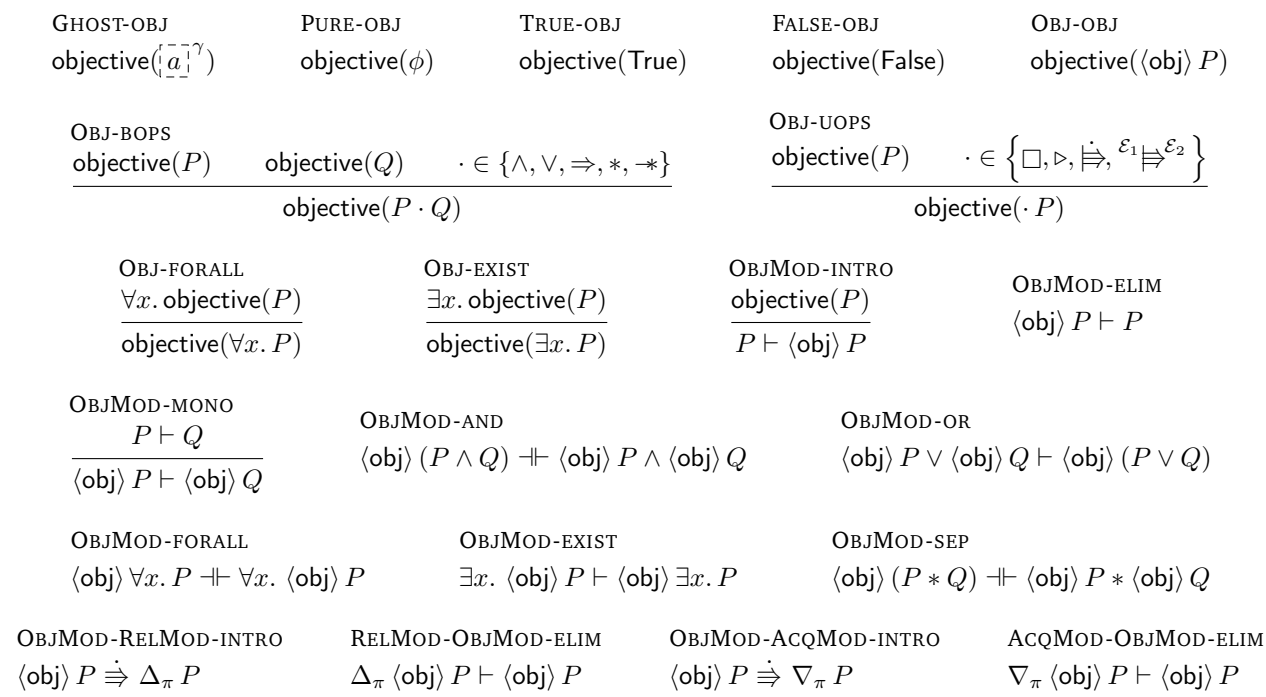


FIGURE 8.2: iRC11 rules for objective propositions and the objective modality

fact in the logic. OBJMOD-ELIM says that a resource P under an objective modality can be used any time, because it holds at any view.

Last but not least, OBJMOD-RELMOD-INTRO, RELMOD-OBJMOD-ELIM, OBJMOD-ACQMOD-INTRO, and ACQMOD-OBJMOD-ELIM together state that resources under the objective modality move freely in and out of the fence modalities, because they do not depend on any view. In fact, the rules for ghost state interaction with fence modalities (RELMOD-GHOST, ACQMOD-GHOST, GHOST-RELMOD, and GHOST-ACQMOD) are derived from these rules, together with GHOST-OBJ, OBJMOD-INTRO, and OBJMOD-ELIM.

Note 8.11 (On the objectivity of fence modalities). A resource P under a fence modality, *e.g.*, $\Delta_\pi\, P$, is objective, but that does not mean that P is objective. P is still interpreted at some snapshot view of the thread π's thread-view.

8.5 View-explicit Modalities

As mentioned in the beginning of this chapter—and as will be demonstrated in later chapters, it is not always desirable to hide views. We therefore would like the ability to *briefly* perform explicit view reasoning *without* dropping back to the base logic. The solution is to introduce view-explicit modalities. This has been done on an ad hoc basis in the RB$_{\texttt{rlx}}$ work (Part III), then developed more formally by the Cosmo logic,[8] and then used extensively in Compass (Part IV).

In the following, we present a formal account of these modalities, and their interaction with other modalities as well as among themselves. Note that this formalization can also be generalized further beyond vProp, to achieve modalities in a logic with thread-local state.

[8] Mével et al., "Cosmo: a concurrent separation logic for multicore OCaml" [MJP20].

Definition 8.12 (The View-Seen Observations). $\boxed{\sqsupseteq V}$

The view-seen observation $\sqsupseteq V$ asserts that the implicit view that is being used to interpret resources is at least V.

$$[\![\sqsupseteq V]\!] ::= \lambda V'.\, V \sqsubseteq V'$$

The view-seen observation is similar to the seen thread-view observation (Definition 7.9), but is a vProp proposition and is not limited to a view of some thread. It provides a lower bound of the view being used to interpreted resources at the current point of a proof, regardless whether that proof is for a program execution (a WP) or not.

Definition 8.13 (The View-At Modality). $\boxed{@_V P}$

The view-at modality $@_V P$ asserts that P holds explicitly (at least) at the view V.

$$[\![@_V P]\!] ::= \lambda\_.\, [\![P]\!](V)$$

When we progress through a proof—with or without a program execution (*i.e.*, a WP)—in the iRC11 logic, either due to program execution or due to possible explicit monotonization of vProp propositions, the view being used to interpret our resources may grow. The view-at modality $@_V P$ allows us to keep some resource P *frozen* at some view V and not affected by the growth of the implicit interpreting view. This ability is needed in case the interpreting view grows too big, rendering our ownership of P useless.

Definition 8.14 (The View-Join Modality). $\boxed{\sqcup_V P}$

The view-join modality $\sqcup_V P$ asserts that P holds at the join of V and the implicit view that is being used to interpret resources.

$$[\![\sqcup_V P]\!] ::= \lambda V'.\, [\![P]\!](V' \sqcup V)$$

The view-join modality is a compromise between a implicit view and a view-at modality: it remembers the difference between the implicit interpreting view and the view that justifies P. This allows the view that justifies P to still grow, but not too far away from the implicit view of the current proof.

Figure 8.3 lists several important properties of these new propositions.

- The seen-view observation is timeless and persistent. The observation for the empty view is always available (VS-BOT) and objective. The seen-view observation lifts the join operation of the view lattice to separation in the logic (VS-JOIN). Observations are also downward closed (VS-MONO).
- The view-at modality makes the interpreting view explicit and therefore is objective. It preserves timelessness and persistency, and is upward closed (VA-MONO), due to view monotonicity. The modality commutes with most connectives and modalities, almost in both directions (VA-BOPS, VA-UNOPS, VA-IMPL, and VA-WAND).

VS-BOT
$\vdash \sqsupseteq\varnothing$

$\mathsf{objective}(\sqsupseteq\varnothing)$
$\mathsf{timeless}(\sqsupseteq V)$
$\mathsf{persistent}(\sqsupseteq V)$

VS-JOIN
$\sqsupseteq(V_1 \sqcup V_2) \dashv\vdash \sqsupseteq V_1 * \sqsupseteq V_2$

VS-MONO
$$\frac{V_1 \sqsubseteq V_2}{\sqsupseteq V_2 \vdash \sqsupseteq V_1}$$

$\mathsf{objective}(@_V P)$

$$\frac{\mathsf{timeless}(P)}{\mathsf{timeless}(@_V P)}$$

$$\frac{\mathsf{persistent}(P)}{\mathsf{persistent}(@_V P)}$$

VA-MONO
$$\frac{V_1 \sqsubseteq V_2 \qquad P \vdash Q}{@_{V_1} P \vdash @_{V_2} Q}$$

$$\frac{\mathsf{objective}(P)}{\mathsf{objective}(\sqcup_V P)}$$

$$\frac{\mathsf{timeless}(P)}{\mathsf{timeless}(\sqcup_V P)}$$

$$\frac{\mathsf{persistent}(P)}{\mathsf{persistent}(\sqcup_V P)}$$

VJ-MONO
$$\frac{V_1 \sqsubseteq V_2 \qquad P \vdash Q}{\sqcup_{V_1} P \vdash \sqcup_{V_2} Q}$$

VA-INTRO
$P \vdash \exists V.\ \sqsupseteq V * @_V P$

VA-INTRO-INCL
$P * \sqsupseteq V' \vdash \exists V \sqsupseteq V'.\ \sqsupseteq V * @_V P$

VA-ELIM
$\sqsupseteq V * @_V P \vdash P$

VA-VS
$V_1 \sqsubseteq V_2 \dashv\vdash @_{V_2} \sqsupseteq V_1$

VA-OBJ
$$\frac{\mathsf{objective}(P)}{@_V P \dashv\vdash P}$$

VA-INTRO-OBJ
$\langle \mathsf{obj} \rangle\ P \vdash @_V P$

VA-IDEMP
$@_{V_2} @_{V_1} P \dashv\vdash @_{V_1} P$

VA-BOPS
$$\frac{\cdot \in \{\wedge, \vee, *\}}{@_V P \cdot Q \dashv\vdash @_V P \cdot @_V Q}$$

VA-UNOPS
$$\frac{\cdot \in \left\{ \forall x.\,, \exists x.\,, \Box, \triangleright, \dot{\Rightarrow}, {}^{\varepsilon_1}\!\Rightarrow^{\varepsilon_2} \right\}}{@_V \cdot P \dashv\vdash \cdot @_V P}$$

VA-IMPL
$@_V P \Rightarrow Q \vdash @_V P \Rightarrow @_V Q$

VA-WAND
$@_V P \mathrel{-\!\!*} Q \vdash @_V P \mathrel{-\!\!*} @_V Q$

VJ-UNFOLD
$\sqcup_V P \dashv\vdash \sqsupseteq V \Rightarrow P$

VJ-JOIN
$\sqcup_{V_1} \sqcup_{V_2} P \dashv\vdash \sqcup_{(V_1 \sqcup V_2)} P$

VA-VJ
$@_{V_1} \sqcup_{V_2} P \dashv\vdash @_{(V_1 \sqcup V_2)} P$

VJ-VA
$\sqcup_{V_2} @_{V_1} P \dashv\vdash @_{V_1} P$

VA-TO-VJ
$@_V P \vdash \sqcup_V P$

VJ-INTRO-NOW
$P \vdash \sqcup_V P$

VJ-ELIM
$\sqsupseteq V * \sqcup_V P \vdash P$

VJ-ELIM-VA
$\sqsupseteq V_1 * \sqcup_{V_2} P \vdash \exists V' \sqsupseteq V_1.\ \sqsupseteq V' * @_{(V' \sqcup V_2)} P$

VJ-VA-ACC
$\sqsupseteq V_1 * \sqcup_{V_2} P \vdash \exists V' \sqsupseteq V_1.\ \sqsupseteq V' * @_{(V' \sqcup V_2)} P * (\forall V''.\ \sqsupseteq V'' * @_{(V'' \sqcup V_2)} P \mathrel{-\!\!*} \sqcup_{V_2} P)$

VJ-INTRO-OBJ
$\langle \mathsf{obj} \rangle\ P \vdash \sqcup_V P$

VJ-OBJ
$$\frac{\mathsf{objective}(P)}{\sqcup_V P \dashv\vdash P}$$

VJ-BOPS
$$\frac{\cdot \in \{\wedge, \vee, \Rightarrow, *, \mathrel{-\!\!*}\}}{\sqcup_V P \cdot Q \dashv\vdash \sqcup_V P \cdot \sqcup_V Q}$$

VJ-UNOPS
$$\frac{\cdot \in \left\{ \forall x.\,, \exists x.\,, \Box, \triangleright, \dot{\Rightarrow}, {}^{\varepsilon_1}\!\Rightarrow^{\varepsilon_2} \right\}}{\sqcup_V \cdot P \dashv\vdash \cdot \sqcup_V P}$$

FIGURE 8.3: iRC11 rules for view-explicit modalities

Objective propositions often ignore the view-at modality (VA-OBJ and VA-INTRO-OBJ). VA-IDEMP says that the inner-most view-at modality dominates. VA-VS says what it means for a view V_2 to observe a view V_1: it is simply that $V_1 \sqsubseteq V_2$.

The two most important rules for the modality are its introduction and elimination rules. VA-INTRO allows us to freeze an owned resource P at some view V that we have observed ($\sqsupseteq V$). As such, we can send $@_V P$ and $\sqsupseteq V$ away on different routes—*a separation of resources and observations*. A receiver once receives both parts can use VA-ELIM to regain P. VA-INTRO-INCL strengthens VA-INTRO to know more about the fixed view V.

- The view-join modality preserves timelessness, persistency, and

objectivity, and is also upward closed. It commutes with most connectives and modalities (VJ-BOPS and VJ-UNOPS). Again, objective propositions ignore the view-join modality (VJ-OBJ and VJ-INTRO-OBJ).

VJ-UNFOLD provides an alternative definition for the view-join modality, which states more clearly that P holds at view whose difference with the implicit view is V. VA-VJ, VJ-VA, and VA-TO-VJ provide important relations between the view-at and view-join modalities. VJ-INTRO-NOW allows us to move the owned P to a bigger view and introduce $\sqcup_V P$. VJ-ELIM allows us to eliminate the modality, in the same way as VA-ELIM. Finally, VJ-ELIM-VA allows us to go from the view-join modality to the view-at modality. Combining that with VA-VJ and VA-ELIM, we get the rule VJ-VA-ACC that allows us to switch between the two modalities.

Definition 8.15 (Alternative Model for Fence Modalities)**.** The fence modalities are in fact defined in vProp using the view-at modality.

$$\Delta_\pi\, P ::= \exists \mathcal{V}. \boxed{\circ \mathcal{V}}^\pi * @_{\mathcal{V}.\mathsf{frel}} P$$
$$\nabla_\pi\, P ::= \exists \mathcal{V}. \boxed{\circ \mathcal{V}}^\pi * @_{\mathcal{V}.\mathsf{acq}} P$$

After unfolding into the base logic, this is exactly the same as Definition 8.8.

We note that the release modality and the view-at modality can interact through the following rule.

$$\frac{\text{RELMOD-VA-REVERT}\quad \forall V.\, \{@_V P * \Delta_\pi \sqsupseteq V\}\, e\, \mathsf{in}\, \pi\, \{\Phi\}_\mathcal{E}}{\{\Delta_\pi\, P\}\, e\, \mathsf{in}\, \pi\, \{\Phi\}_\mathcal{E}}$$

That is, with a goal in the form of a WP for the thread π, we can turn the assumption $\Delta_\pi\, P$ into $@_V P * \Delta_\pi \sqsupseteq V$ for some view V.

8.6 The Subjective Modality

Finally, we introduce the subjective modality, a derivation from the view-at modality.

Definition 8.16 (The Subjective Modality)**.**

$$\langle \mathsf{subj} \rangle\, P ::= \exists V.\, @_V P$$

That is, the subjective modality asserts that $P : \mathsf{vProp}$ holds at *some* view that is hidden from others. The name "subjective" comes from the fact that P holds in someone's subjective view.

The subjective modality satisfies the rules in Figure 8.4, which are derivable from the rules for the view-at modality. Some of these properties hold for general monotone predicates, but some (*e.g.*, the reverse direction of SUBJMOD-SEP) only hold for monotone predicates on a lattice, which for vProp is the view lattice.

$\mathsf{objective}(\langle \mathsf{subj} \rangle\, P)$

$$\frac{\mathsf{timeless}(P)}{\mathsf{timeless}(\langle \mathsf{subj} \rangle\, P)}$$

$$\frac{\mathsf{persistent}(P)}{\mathsf{persistent}(\langle \mathsf{subj} \rangle\, P)}$$

SUBJMOD-MONO
$$\frac{P \vdash Q}{\langle \mathsf{subj} \rangle\, P \vdash \langle \mathsf{subj} \rangle\, Q}$$

SUBJMOD-INTRO
$P \vdash \langle \mathsf{subj} \rangle\, P$

SUBJMOD-VA
$\exists V.\, @_V P \dashv\vdash \langle \mathsf{subj} \rangle\, P$

SUBJMOD-ELIM-OBJ
$$\frac{\mathsf{objective}(P)}{\langle \mathsf{subj} \rangle\, P \vdash P}$$

SUBJMOD-AND
$\langle \mathsf{subj} \rangle\, (P \land Q) \vdash \langle \mathsf{subj} \rangle\, P \land \langle \mathsf{subj} \rangle\, Q$

SUBJMOD-OR
$\langle \mathsf{subj} \rangle\, P \lor \langle \mathsf{subj} \rangle\, Q \dashv\vdash \langle \mathsf{subj} \rangle\, (P \lor Q)$

SUBJMOD-FORALL
$\langle \mathsf{subj} \rangle\, \forall x.\, P \vdash \forall x.\, \langle \mathsf{subj} \rangle\, P$

SUBJMOD-EXIST
$\exists x.\, \langle \mathsf{subj} \rangle\, P \dashv\vdash \langle \mathsf{subj} \rangle\, \exists x.\, P$

SUBJMOD-SEP
$\langle \mathsf{subj} \rangle\, (P * Q) \dashv\vdash \langle \mathsf{subj} \rangle\, P * \langle \mathsf{subj} \rangle\, Q$

SUBJMOD-LATER
$\triangleright \langle \mathsf{subj} \rangle\, P \dashv\vdash \langle \mathsf{subj} \rangle\, \triangleright P$

FIGURE 8.4: iRC11 rules for the subjective modality

CHAPTER SUMMARY. In this chapter, we presented view-monotone predicates vProp—the type of iRC11 propositions—and the lifting of many base-logic connectives and modalities to those of iRC11. We have defined iRC11 WPs also in terms of the base logic WPs, and showed iRC11 adequacy. We have also defined various iRC11 modalities: fence modalities and view-explicit modalities. In the next chapters, we follow the same approaches to derive more iRC11 assertions and their rules on top of the base logic: the non-atomic and atomic points-to assertions, and invariants.

9

Non-Atomic Points-To

The points-to assertion $\ell \mapsto v$ is a well-known feature of separation logics. It represents unique ownership of the location ℓ, which allows for safe, non-racy operations on ℓ. For concurrent reads, the assertion can be equipped with fractional permission, *i.e.*, $\ell \overset{q}{\mapsto} v$. Ownership of a fraction $q \in (0, 1]$ is sufficient to prevent concurrent writes. We would like to have these features for non-atomic accesses: concretely, the full ownership of a points-to $\ell \mapsto v$ should be sufficient to safely perform non-atomic writes (and thus also any atomic operations), while a fractional $\ell \overset{q}{\mapsto} v$ should be sufficient to safely perform non-atomic reads (and thus also any atomic reads). In this chapter, we give a model for iRC11's *non-atomic points-to* assertion that satisfies this interface, using the base logic local assertions defined in Chapter 7. In Chapter 10, we will also discuss iRC11's ability to switch between non-atomic and atomic points-to assertions.

9.1 The Interface of Non-Atomic Points-To

The interface of iRC11 non-atomic points-to is rather standard, as given in Figure 9.1. NA-FRAC, NA-FRAC-VALID, and NA-FRAC-AGREE together say that non-atomic points-to is fractional, and NA-EXCL says that full ownership of a non-atomic points-to is exclusive. NA-READ allows us to perform non-racy reads using *any* access mode o with a fraction $\ell \overset{q}{\mapsto} v$, and we are guaranteed the return value is v. NA-WRITE allows us to perform non-racy writes also using *any* access mode o with the full fraction $\ell \mapsto \_$, and we know afterwards ℓ has the value just written. The support for an arbitrary access mode o reflects the fact that if the points-to ownership is sufficient to safely perform the most demanding mode (non-atomic, **na**), then it should also be sufficient for less demanding ones.

Furthermore, NA-ALLOC says that an allocation gives us the full block ownership ($\dagger^n \ell$—lifted from the base logic[1] to vProp—and the full non-atomic points-to ownership ($\ast_{m\in[0,n)} \ell + m \mapsto \text{☣}$) for all locations of the newly allocated block, whose base location is ℓ. Conversely, NA-DEALLOC consumes the block ownership and the points-to ownership of all locations. We strengthen NA-DEALLOC slightly by only requiring a weaker points-to $\ell \mapsto ?$, which we call an *unsynchronized points-to*. Intuitively, the ownership of an unsynchronized points-to $\ell \mapsto ?$ only guarantees that the owning thread has observed the latest write to ℓ, but is not synchronized with that write, *i.e.*, it has not observed the write's message

[1] see Definition 7.1

NA-FRAC
$\ell \overset{q}{\mapsto} v * \ell \overset{q'}{\mapsto} v \dashv\vdash \ell \overset{q+q'}{\mapsto} v$

NA-FRAC-VALID
$\ell \overset{q}{\mapsto} v \vdash q \in (0, 1]$

NA-FRAC-AGREE
$\ell \overset{q}{\mapsto} v * \ell \overset{q'}{\mapsto} v' \vdash v = v'$

NA-EXCL
$\ell \mapsto v * \ell \mapsto v' \vdash \mathsf{False}$

NA-READ
$\{\ell \overset{q}{\mapsto} v\}\, {}^{*o}\ell\ \mathsf{in}\ \pi\ \{w.\, w = v * \ell \overset{q}{\mapsto} v\}_{\mathcal{E}}$

NA-WRITE
$\{\ell \mapsto \_\}\, \ell :=_o v\ \mathsf{in}\ \pi\ \{\text{☣}.\, \ell \mapsto v\}_{\mathcal{E}}$

NA-ALLOC
$$\frac{0 < n}{\{\mathsf{True}\}\ \ \mathbf{alloc}(n)\ \mathsf{in}\ \pi\ \left\{\ell.\ \dagger^n\ \ell * \mathop{\Large *}_{m\in[0,n)} \ell + m \mapsto \text{☣}\right\}_{\mathcal{E}}}$$

NA-DEALLOC
$$\frac{0 < n}{\left\{\dagger^n\, \ell * \mathop{\Large *}_{m\in[0,n)} \ell + m \mapsto ?\right\}\ \ \mathbf{free}(\ell, n)\ \mathsf{in}\ \pi\ \{\text{☣}.\, \mathsf{True}\}_{\mathcal{E}}}$$

NA-UNSYNC
$\ell \overset{q}{\mapsto} \_ \vdash \ell \mapsto ?$

where $\ell \overset{q}{\mapsto} \_ ::= \exists v.\, \ell \overset{q}{\mapsto} v$.

FIGURE 9.1: Rules for iRC11 non-atomic points-to

view. On the other hand, ownership of $\ell \mapsto v$ guarantees that the thread is synchronized with that latest write, which is the write of v. The rule NA-UNSYNC demonstrates that the latter is stronger than the former.

9.2 The Model of Non-Atomic Points-To

In order to define the non-atomic points-to assertion purely within vProp, we first lift the base logic local assertions (either in iProp or in the meta-level logic, see Definition 7.1) to vProp as follows.

Definition 9.1 (Lifting Local Assertions to vProp)**.**

$$\begin{aligned}
[\![\mathsf{Hist}(\ell, h)]\!] &::= \lambda_.\, \mathsf{Hist}(\ell, h)\\
[\![\mathsf{Write}^{\sqsupseteq \mathbf{rlx}}(\ell, \alpha)]\!] &::= \lambda_.\, \mathsf{Write}^{\sqsupseteq \mathbf{rlx}}(\ell, \alpha)\\
[\![\mathsf{Read}^{\mathbf{na}}(\ell, \alpha)]\!] &::= \lambda_.\, \mathsf{Read}^{\mathbf{na}}(\ell, \alpha)\\
[\![\mathsf{Read}^{\sqsupseteq \mathbf{rlx}}(\ell, \alpha)]\!] &::= \lambda_.\, \mathsf{Read}^{\sqsupseteq \mathbf{rlx}}(\ell, \alpha)\\
[\![\mathsf{Local}_{\mathtt{A}}(\ell, h)]\!] &::= \lambda V.\, \mathsf{Local}_{\mathtt{A}}(\ell, h, V)\\
[\![\mathsf{Local}_{\mathtt{W}}^{\sqsupseteq \mathbf{rlx}}(\ell, \alpha)]\!] &::= \lambda V.\, \mathsf{Local}_{\mathtt{W}}^{\sqsupseteq \mathbf{rlx}}(\ell, \alpha, V)\\
[\![\mathsf{Local}_{\mathtt{R}}^{\sqsupseteq \mathbf{rlx}}(\ell, \alpha)]\!] &::= \lambda V.\, \mathsf{Local}_{\mathtt{R}}^{\sqsupseteq \mathbf{rlx}}(\ell, \alpha, V)\\
[\![\mathsf{Local}_{\mathtt{R}}^{\mathbf{na}}(\ell, \alpha, V_{\mathbf{na}})]\!] &::= \lambda V.\, \mathsf{Local}_{\mathtt{R}}^{\mathbf{na}}(\ell, \alpha, V_{\mathbf{na}}) \wedge V_{\mathbf{na}} \sqsubseteq V
\end{aligned}$$

The lifting is straightforward. Recall that the various local ownership for parts of the race-detector state are purely ghost state,[2] so in lifting them to vProp we simply ignore the interpreting view. For local observations, we use the interpreting view V as the last argument to the meta-level assertions. Recall Property 7.5 that the local observations are all view-monotone.

[2] see Definition 7.14

Remark 9.2 (The non-atomic view $V_{\mathbf{na}}$)**.** Note that unlike the rest of iRC11 local observations, we do not hide the view $V_{\mathbf{na}}$ of the non-atomic local observation $\mathsf{Local}_{\mathtt{R}}^{\mathbf{na}}(\ell, \alpha, V_{\mathbf{na}})$. Instead, we require that the implicit interpreting view V includes $V_{\mathbf{na}}$. The view $V_{\mathbf{na}}$ is called the *non-atomic view*, and we expose it to record the view of the most recent *non-atomic*

access period. Intuitively, safe accesses to a location ℓ must alternate between periods of non-atomic accesses and periods of atomic accesses. Interestingly, the switch from a non-atomic access period to an atomic access period of ℓ can happen (logically) much later than the most recent physical non-atomic operation to ℓ. That is, *the end of a non-atomic access period may not logically coincide with the most recent non-atomic access*. Even so, any incoming atomic accesses of the new atomic access period must synchronize with not only the most recent non-atomic operation, but with the point of the switch itself. Therefore, we use the view V_{na} to track the view of the switch, so as to make more resources available to the incoming atomic accesses. In short, the non-atomic view V_{na} is needed to have strong reasoning principles for switching between non-atomic and atomic accesses, and its uses will be explained more clearly in Chapter 10. In this chapter, we can simply ignore this view.

Definition 9.3 (Model of $\ell \mapsto v$)**.** We define a *primitive* non-atomic points-to $\ell \overset{q}{\mapsto}_{\mathsf{na}} h$ which represents fractional ownership of ℓ with a history h, and then use it to define the unsynchronized non-atomic points-to $\ell \overset{q}{\mapsto} ?$ and the actual points-to $\ell \overset{q}{\mapsto} v$.

$$
\begin{aligned}
\ell \overset{q}{\mapsto}_{\mathsf{na}} h ::= \exists \alpha_w, \alpha_1, \alpha_2.\, & \mathsf{Local}_{\mathtt{A}}(\ell, h) * \mathsf{Local}_{\mathtt{W}}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha_w) * \\
& \mathsf{Local}_{\mathtt{R}}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha_2) * (\exists V_{\mathsf{na}}.\, \mathsf{Local}_{\mathtt{R}}^{\mathtt{na}}(\ell, \alpha_1, V_{\mathsf{na}})) * \\
& \mathsf{Hist}_q(\ell, h) * \mathsf{Write}_q^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha_w) * \\
& \mathsf{Read}_q^{\mathtt{na}}(\ell, \alpha_1) * \mathsf{Read}_q^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha_2) \\
\ell \overset{q}{\mapsto} ? ::= \exists t, v, V^?.\, & \ell \overset{q}{\mapsto}_{\mathsf{na}} \left[t \leftarrow (v, V^?)\right] \\
\ell \overset{q}{\mapsto} v ::= \exists t, V^?.\, & \ell \overset{q}{\mapsto}_{\mathsf{na}} \left[t \leftarrow (v, V^?)\right] * \sqsupseteq V^?
\end{aligned}
$$

A fraction q of the primitive non-atomic points-to for ℓ contains the corresponding fractions for ℓ's history ownership of h and the parts of race-detector state. By $\mathsf{Local}_{\mathtt{A}}(\ell, h)$, the owner of $\ell \overset{q}{\mapsto}_{\mathsf{na}} h$ has also observed the allocation of ℓ. The sets α_w, α_1, and α_2 of atomic writes, non-atomic and atomic reads, respectively, are existentially quantified, and, due to the local observations, all sets are also observed by the owner of $\ell \overset{q}{\mapsto}_{\mathsf{na}} h$.

The unsynchronized non-atomic points-to $\ell \overset{q}{\mapsto} ?$ then simply requires that the history be a singleton $\left[t \leftarrow (v, V^?)\right]$ for ℓ's latest write event $(t, v, V^?)$. The non-atomic points-to $\ell \overset{q}{\mapsto} v$ additionally fixes the value to be v, and requires that the owner has observed the message view $V^?$.

The definitions clearly show that NA-UNSYNC holds. We sketch the proofs for the remaining rules.

Proof sketch that $\ell \mapsto v$ is fractional. Proofs of NA-FRAC, NA-FRAC-VALID, and NA-FRAC-AGREE follow from the fact that the ownership history and the local assertions for parts of the race-detector state are all fractional—see Figure 7.2. In proving NA-FRAC, we will need BL-NAL-JOIN and BL-ATRL-JOIN to join the local observations for reads. □

Proof sketch of NA-ALLOC. We perform the proof in the base logic. Note that we do not have a base logic rule for allocation and deallocation, so we will need to prove both NA-ALLOC and NA-DEALLOC by unfolding

the both WP definitions of iRC11 (Definition 8.4) and of the base logic (Definition 6.14), and work directly with the state interpretation (Definition 7.18), like for other base logic WP rules.

Fortunately, the pre-condition and the race-free condition for allocation is trivial. As the newly allocated block—whose base location is ℓ—is fresh in the global memory,[3] we can update the global ghost state GlobalGhost to mirror the change in the global physical state, namely we allocate the block ownership $\dagger^n \ell$, and the history ownership as well as the local assertions for all locations in the newly allocated block, all of which are needed to construct the non-atomic points-to for them. Note that the allocated locations all have the allocated value $\dagger$, which is lifted to the poison value $\lightning$ in iRC11. □

[3] see OM-ALLOC, Figure 3.3, §3.3

Proof sketch of NA-DEALLOC. After unfolding the WP definitions of both iRC11 and the base logic, we first need to show that the step is safe (it reduces). With the full fraction block ownership and the global ghost ownership GlobalGhost, we can use BL-GHOST-BLOCK-FULL (Figure 7.8, §7.5) to know that we have collected the ownership of all locations in the block. Furthermore, the unsynchronized non-atomic points-to of all the locations guarantee that they are still alive, and that the deallocation is race-free for all of them, as the deallocation acts like a non-atomic write. Consequently, we satisfy both OM-FREE (§3.3) and DRF-DEALLOC (§3.4), so the deallocation reduces. Since we do not need the caller to have synchronized with all the message views of the locations' latest writes, the unsynchronized non-atomic points-to's $\ell + m \mapsto ?$ are sufficient.

After the step, we update the state interpretation to match the changed global state. Fortunately, the global ghost GlobalGhost are very loose on deallocated locations—we only need to update the ghost histories of the deallocated locations to None. □

Proof sketch of NA-READ. We only need to unfold the definitions of the non-atomic points-to (Definition 9.3) and the iRC11 Hoare triples and WPs (Definition 8.5 and Definition 8.4), and perform the proof in the base logic. Recall that by Definition 8.5, all of our resources are interpreted at the current component $\mathcal{V}$.cur of the thread-view $\mathcal{V}$.

- In case $o = \mathbf{na}$, we apply BL-HOARE-READ-NA (§7.3.2). Note that V_r is instantiated to $V_{\mathbf{na}}$. In the post-condition we only need to use the post-condition of BL-HOARE-READ-NA to address the only change by the read, which is the non-atomic reads set and its local observation.
- In case $o \sqsupseteq \mathbf{rlx}$, we apply BL-HOARE-READ-AT (§7.3.2). Note that the history in the non-atomic points-to is a singleton, and V_r is instantiated to $\mathcal{V}$.cur, so the proof is straightforward.

□

Proof sketch of NA-WRITE. The proof is similar to that of NA-READ. We use BL-HOARE-WRITE-NA in case $o = \mathbf{na}$. Otherwise, if $o \sqsupseteq \mathbf{rlx}$, we use BL-HOARE-WRITE-AT, and then use BL-HIST-DROP-SINGLETON (Figure 7.2, §7.2) to shrink the history back to a singleton. □

10

Atomic Points-To

The *atomic points-to* assertion plays the similar role as the non-atomic points-to assertion, but for atomic accesses. It is iRC11's first abstraction for the ownership needed to safely perform atomic accesses. It can be used directly to verify ORC11 code, but iRC11 also uses it to derive the higher-level GPS protocols[1] (Part III). Nevertheless, the atomic points-to assertion is more flexible than iRC11's version of GPS protocols because they work more explicitly with views. Consequently, it is used pervasively in Compass, in conjunction with logical atomic triples (Part IV).

[1] Turon et al., "GPS: navigating weak memory with ghosts, protocols, and separation" [TVD14]; Kaiser et al., "Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris" [Kai+17].

iRC11 atomic points-to assertion is inspired by Cosmo's atomic points-to assertion[2] to work with explicit views. However, since Cosmo is sound for the stronger Multicore OCaml memory model, its atomic points-to assertion is fairly simple: the (potentially fractional) assertion $\ell \mapsto_{\mathtt{at}} (v, V)$ represents Cosmo's ownership of an atomic location ℓ with the value v and view V of the latest write. That is, Cosmo's atomic points-to needs only to take care of the latest write, because atomic accesses in Multicore OCaml are much stronger than the different access modes supported by C11. In contrast, iRC11 atomic points-to assertion needs to carry around a history of multiple writes that are still visible to accessing threads, and to provide multiple rules for the different access modes.

[2] Mével et al., "Cosmo: a concurrent separation logic for multicore OCaml" [MJP20].

In the presence of concurrent writes to the same location ℓ, iRC11 rules for handling ℓ's history are rather cumbersome and hard to use. In practice, if a client performs arbitrary concurrent writes to a location ℓ, then the concurrent protocol for ℓ is often trivial. That is because there would be no clear order between the writes: in the ORC11 semantics, we will see that the writes arrive in the history randomly, with **holes** in the history.[3] More specifically, this is the result of adapting C11's support for *non-multi-copy-atomicity* (non-MCA), *i.e.*, the property where writes can arrive at different threads in different orders.

[3] see OM-WRITE, §3.3

Fortunately, algorithms tend to avoid concurrent writes where interesting protocols are needed: they either have a single writer and multiple concurrent readers, or have all participants purely perform compare-and-swap (CASes) operations to resolve potential contention. In such fashion, the history has no holes, and the mo order becomes more meaningful and can be used to support some well-ordered protocol.

Consequently, iRC11 provides *multiple modes* for the atomic points-to assertion to cater to these common cases. In §10.1, we present these modes for the atomic points-to, the relations among them and with the

non-atomic points-to, and iRC11 Hoare rules for atomic accesses with the atomic points-to. In §10.2, we give the model of the atomic points-to assertion, built atop the local assertions of the base logic (§7.2).

10.1 The Interface of the Atomic Points-To Assertion

We support 3 modes for the atomic points-to: arbitrarily concurrent (`con`), single-writer (`sw`), and CAS-only (`cas`).

$$\theta \in \mathit{AtomicMode} ::= \texttt{con} \mid \texttt{sw} \mid \texttt{cas}.$$

The atomic points-to with the arbitrarily concurrent mode `con` supports any access mode, but with a weak set of WP or Hoare rules. The other two modes enjoy stronger WP or Hoare rules. In the single-writer mode `sw`, all writes using the atomic points-to must be made sequential (synchronized), but reads can be arbitrarily concurrent. In the CAS-only mode `cas`, the atomic points-to only supports CASes to write, and reads can be arbitrarily concurrent.

Definition 10.1 (Atomic Points-To Assertion)**.** The atomic points-to assertion has the form $\ell \overset{t_x}{\longmapsto}{}^{\gamma}_{\theta}\ h$ where (1) θ is one of the 3 atomic modes; and (2) $\varnothing \neq h \in \mathit{History}$ is ℓ's current history which contains write events still visible to accessing threads; and (3) γ is a ghost location used to uniquely identity an atomic period of the atomic points-to and can be ignored for now; and (4) t_x is the timestamp of the latest exclusive *single-writer* write, which will be needed for GPS protocols and can also be ignored for now.

We write $\ell \mapsto_\theta h ::= \exists t.\, \ell \overset{t_x}{\longmapsto}_\theta h$ to ignore the exclusive single-writer timestamp.

Definition 10.2 (Atomic Local Ownership and Observations)**.** The atomic points-to assertion $\ell \mapsto_\theta h$ is needed for every atomic access, and thus will be put inside an invariant for shared concurrent access. Therefore, we need to define several local ownership and observations to represent what a thread knows about the shared history h of the atomic points-to.

- The history-seen observation $\ell \sqsupseteq_{\texttt{sn}} h$ asserts the observation of all ℓ's write events in the non-empty history h. This observation is the minimum requirement to perform an atomic read on ℓ.
- The history-sync observation $\ell \sqsupseteq_{\texttt{sy}} h$ asserts not only the observation of ℓ's write events of in h, but also the observation of those writes' message views.
- The single-writer ownership $\ell \sqsupseteq_{\texttt{sw}} h$ asserts the exclusive permission to write (the single-writer) to ℓ, and the history-sync observation of h (*i.e.*, $\ell \sqsupseteq_{\texttt{sy}} h$). The single-writer ownership guarantees that h is the current history of ℓ.
- The fractional CAS ownership $\ell \sqsupseteq^{t_x,q}_{\texttt{cas}} h$ asserts the shared permission to CAS to ℓ, and the history-seen observation of h (*i.e.*, $\ell \sqsupseteq_{\texttt{sn}} h$). A fraction q of the CAS ownership only guarantees that h

$\mathsf{persistent}(\ell \sqsupseteq_{\mathtt{sn}} h)$
$\mathsf{persistent}(\ell \sqsupseteq_{\mathtt{sy}} h)$
$\mathsf{timeless}(\ell \sqsupseteq_{\mathtt{sn}} h)$
$\mathsf{timeless}(\ell \sqsupseteq_{\mathtt{sy}} h)$

$\mathsf{timeless}(\ell \overset{t}{\mapsto}_{\theta} h)$
$\mathsf{timeless}(\ell \sqsupseteq_{\mathtt{sw}} h)$
$\mathsf{timeless}(\ell \sqsupseteq^{t,q}_{\mathtt{cas}} h)$

AT-EXCL
$\ell \overset{t}{\mapsto}_{\theta} h * \ell \overset{t'}{\mapsto}_{\theta'} h' \vdash \mathsf{False}$

AT-SW-EXCL
$\ell \sqsupseteq_{\mathtt{sw}} h * \ell \sqsupseteq_{\mathtt{sw}} h' \vdash \mathsf{False}$

AT-SW-CAS-EXCL
$\ell \sqsupseteq_{\mathtt{sw}} h * \ell \sqsupseteq^{q}_{\mathtt{cas}} h' \vdash \mathsf{False}$

AT-SW-AGREE
$\ell \mapsto_{\theta} h * \ell \sqsupseteq_{\mathtt{sw}} h' \vdash \theta = \mathtt{sw} \land h = h'$

AT-CAS-FRAC-AGREE
$\ell \overset{t}{\mapsto}_{\theta} h * \ell \sqsupseteq^{t',q'}_{\mathtt{cas}} h' \vdash \theta \neq \mathtt{con} \land t = t' \land h' \subseteq h$

AT-CAS-CAS-FRAC-AGREE
$\ell \sqsupseteq^{t,q}_{\mathtt{cas}} h * \ell \sqsupseteq^{t',q'}_{\mathtt{cas}} h' \vdash t = t' \land q + q' \in (0, 1]$

AT-CAS-JOIN
$\ell \sqsupseteq^{t,q}_{\mathtt{cas}} h * \ell \sqsupseteq^{t',q'}_{\mathtt{cas}} h' \vdash \ell \sqsupseteq^{t,q+q'}_{\mathtt{cas}} (h \cup h')$

AT-CAS-SPLIT
$\ell \sqsupseteq^{t,q+q'}_{\mathtt{cas}} h \vdash \ell \sqsupseteq^{t,q}_{\mathtt{cas}} h * \ell \sqsupseteq^{t,q'}_{\mathtt{cas}} h$

AT-SY
$\ell \mapsto_{\theta} h \vdash \ell \sqsupseteq_{\mathtt{sy}} h$

AT-SY-SN
$\ell \sqsupseteq_{\mathtt{sy}} h \vdash \ell \sqsupseteq_{\mathtt{sn}} h$

AT-SW-SY
$\ell \sqsupseteq_{\mathtt{sw}} h \vdash \ell \sqsupseteq_{\mathtt{sy}} h$

AT-CAS-SN
$\ell \sqsupseteq^{q}_{\mathtt{cas}} h \vdash \ell \sqsupseteq_{\mathtt{sn}} h$

AT-SY-MONO
$$\frac{h' \neq \varnothing \qquad h' \subseteq h}{\ell \sqsupseteq_{\mathtt{sy}} h \vdash \ell \sqsupseteq_{\mathtt{sy}} h'}$$

AT-SN-MONO
$$\frac{h' \neq \varnothing \qquad h' \subseteq h}{\ell \sqsupseteq_{\mathtt{sn}} h \vdash \ell \sqsupseteq_{\mathtt{sn}} h'}$$

AT-SY-UNFOLD
$$\frac{h(t) = (v, V)}{\ell \sqsupseteq_{\mathtt{sy}} h \vdash \exists V * \sqsupseteq[\ell \leftarrow \{\varnothing\,[\mathsf{w} := t]\}]}$$

AT-SN-UNFOLD
$$\frac{t \in \mathtt{dom}(h)}{\ell \sqsupseteq_{\mathtt{sn}} h \vdash \sqsupseteq[\ell \leftarrow \{\varnothing\,[\mathsf{w} := t]\}]}$$

AT-SY-JOIN
$\ell \sqsupseteq_{\mathtt{sn}} h * \ell \sqsupseteq_{\mathtt{sy}} h' \vdash \ell \sqsupseteq_{\mathtt{sy}} (h \cup h')$

AT-SN-JOIN
$\ell \sqsupseteq_{\mathtt{sy}} h * \ell \sqsupseteq_{\mathtt{sn}} h' \vdash \ell \sqsupseteq_{\mathtt{sn}} (h \cup h')$

AT-SN-VALID
$\ell \mapsto_{\theta} h * \ell \sqsupseteq_{\mathtt{sn}} h' \vdash h' \subseteq h$

FIGURE 10.1: Basic properties of assertions related to the atomic points-to

is the sub history of ℓ's current history. The timestamp t_x is of the latest exclusive single-writer write to ℓ. As usual, we write $\ell \sqsupseteq^{q}_{\mathtt{cas}} h$ to ignore this timestamp, and write $\ell \sqsupseteq_{\mathtt{cas}} h$ for the full ownership where $q = 1$.

Property 10.3 (Basic Properties of Assertions Related to Atomic Points-To)**.** Figure 10.1 presents several important basic properties of the atomic points-to assertions and its related assertions. All assertions are timeless, and the history-seen and history-sync observations are naturally persistent. The atomic points-to and the single-writer ownership are both exclusive (AT-EXCL and AT-SW-EXCL).

AT-SW-CAS-EXCL says that the single-writer ownership and the CAS ownership are incompatible, implying that the single-writer is indeed single. AT-SW-AGREE says that the atomic points-to and the single-writer ownership must agree on the history and the atomic mode. AT-CAS-FRAC-AGREE says that, on the other hand, the CAS ownership only guarantees that the history h' owned by the CAS ownership is a sub-history of the current history h. This is because CAS ownership are used for concurrent updates, so a fraction should not know the full history. AT-CAS-FRAC-AGREE additionally says that the CAS ownership guarantees that the latest exclusive single-writer timestamp is frozen in CAS-only mode

($t = t'$). Interestingly, AT-CAS-FRAC-AGREE says that the CAS ownership only guarantees that the atomic mode θ is not the concurrent mode $\texttt{con}$. We note that this is just a weakness in our model for atomic points-to, and this weakness does not affect us in practice. A better but slightly more complex model would give us $\theta = \texttt{cas}$.

AT-CAS-CAS-FRAC-AGREE, AT-CAS-JOIN, and AT-CAS-SPLIT encode the fractional nature of the CAS ownership.

AT-SY, AT-SY-SN, AT-SW-SY, and AT-CAS-SN together state the relations between the ownership assertions and the observations. AT-SY says that the atomic points-to naturally has observed and synchronized with all write events. AT-SW-SY says that this also applies to the single-writer ownership, because the writes are sequential. AT-CAS-SN on the other hand says that the CAS ownership does not guarantee synchronization. Recall that the history h in $\ell \sqsupseteq^{q}_{\texttt{cas}} h$ is only a sub-history of the current one, and CAS ownership are used for concurrent updates.

AT-SY-MONO and AT-SN-MONO say that the observations on histories are downward-closed. AT-SY-UNFOLD and AT-SN-UNFOLD clearly state the difference between observing a write and synchronizing with that write, using the view-seen observation (Definition 8.12). AT-SY-JOIN and AT-SN-JOIN allow us to join observations. Finally, AT-SN-VALID says that an observation of h' guarantees that h' is a snapshot (a sub-history) of the current one h.

Property 10.4 (Conversions Between Non-Atomic and Atomic Points-To)**.** The top three rules of Figure 10.2 present the rules for converting the non-atomic points-to ownership to the atomic one. The bottom of Figure 10.2 visualizes the possible conversions between the points-to assertions.

NA-AT-SW says that we can go from the non-atomic points-to assertion to the single-writer atomic one and the single-writer ownership with a singleton history of the latest write (t, v, V), knowing that we have observed the message view V ($\sqsupseteq V$).

NA-AT-SW-VIEW strengthens NA-AT-SW by (1) freezing the atomic points-to and the single-writer ownership at the latest write message view V using the view-at modality (Definition 8.13); and (2) allowing the user to also freeze arbitrary local resource P at the same view. NA-AT-SW-VIEW demonstrates that the view V in fact is not the message view of the latest write in ℓ's history, because ℓ's latest write message view would not be able to justify P. Instead the view V is the view at which the switch (from non-atomic points-to to atomic points-to) happens, and the singleton history $[t \leftarrow (v, V)]$ is not ℓ's actual history, but an abstraction of ℓ's actual history. This abstraction allows subsequent atomic accesses using the atomic points-to assertion to access the view V, and thus the resource P provided at the switch. In other words, NA-AT-SW-VIEW says that the atomic accesses to ℓ after the switch are synchronized not only with the latest write to ℓ before the switch, but also with the switch itself.

AT-NA allows us to go from an atomic points-to back to a non-atomic one, without knowing the atomic mode θ nor having any other ownership (single-writer or CAS ownership). This demonstrates that the atomic points-to itself contains sufficient resources, and the single-write or CAS

NA-AT-SW
$$\ell \mapsto v \dot{\Rrightarrow} \exists \gamma, t, V.\ \sqsupseteq V * \ell \sqsupseteq^{\gamma}_{\mathtt{sw}} [t \leftarrow (v, V)] * \ell \overset{t}{\mapsto}^{\gamma}_{\mathtt{sw}} [t \leftarrow (v, V)]$$

NA-AT-SW-VIEW
$$\ell \mapsto v * P \dot{\Rrightarrow} \exists \gamma, t, V.\ \sqsupseteq V * @_V\, (P * \ell \sqsupseteq^{\gamma}_{\mathtt{sw}} [t \leftarrow (v, V)] * \ell \overset{t}{\mapsto}^{\gamma}_{\mathtt{sw}} [t \leftarrow (v, V)])$$

AT-NA
$$\ell \overset{t_0}{\longmapsto}^{\gamma}_{\theta} h \Rrightarrow_{\mathcal{E}} \exists t \geq t_0, v, V.\ \sqsupseteq V * \ell \mapsto v * h(t) = (v, V) * t = \mathtt{max}(\mathtt{dom}(h))$$

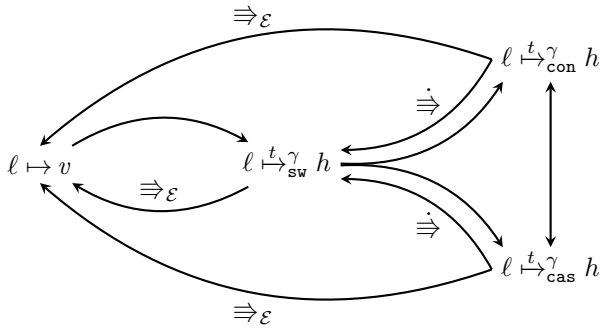
AT-CON-SW
$$\ell \overset{t'}{\longmapsto}^{\gamma}_{\mathtt{con}} h \dot{\Rrightarrow} \exists t = \mathtt{max}(\mathtt{dom}(h)).\ \ell \overset{t}{\mapsto}^{\gamma}_{\mathtt{sw}} h * \ell \sqsupseteq^{\gamma}_{\mathtt{sw}} h$$

AT-SW-CON
$$\ell \overset{t}{\mapsto}^{\gamma}_{\mathtt{sw}} h * \ell \sqsupseteq^{\gamma}_{\mathtt{sw}} h' \vdash \ell \overset{t}{\mapsto}^{\gamma}_{\mathtt{con}} h$$

AT-CAS-SW
$$\ell \overset{t_1}{\longmapsto}^{\gamma}_{\mathtt{cas}} h * \ell \sqsupseteq^{\gamma, t_2}_{\mathtt{cas}} h' \dot{\Rrightarrow} \exists t = \mathtt{max}(\mathtt{dom}(h)).\ \ell \overset{t}{\mapsto}^{\gamma}_{\mathtt{sw}} h * \ell \sqsupseteq^{\gamma}_{\mathtt{sw}} h$$

AT-SW-CAS
$$\ell \overset{t}{\mapsto}_{\mathtt{sw}} h * \ell \sqsupseteq_{\mathtt{sw}} h' \vdash \ell \overset{t}{\mapsto}_{\mathtt{cas}} h * \ell \sqsupseteq^{t}_{\mathtt{cas}} h$$

AT-CON-CAS
$$\ell \overset{t}{\mapsto}^{\gamma}_{\mathtt{con}} h \dashv\vdash \ell \overset{t}{\mapsto}^{\gamma}_{\mathtt{cas}} h * \ell \sqsupseteq^{\gamma, t}_{\mathtt{cas}} h$$



Visualization of the conversions.

FIGURE 10.2: Conversions between the non-atomic and atomic points-to assertion

ownership is purely needed to enforce an access protocol (single-writer or CAS-only). We note that NA-AT-SW and NA-AT-SW-VIEW only need a basic update to switch from non-atomic to atomic, while AT-NA requires a fancy update to go back. The reader may already have guessed correctly that the proof of AT-NA relies on BL-HIST-DROP-SINGLETON (§7.2), which justifies the fancy update. Consequently, the value v we get back for the non-atomic points-to is the latest write, regardless of how that write is made (with a CAS or a normal write using *any* access mode). Thanks to AT-SY and AT-SY-UNFOLD, we know that we have observed the latest write message view V ($\sqsupseteq V$).[4]

[4]Recall that V is not just the latest write message view, it also includes the view of the switch.

CYCLES OF ALTERNATING NON-ATOMIC AND ATOMIC PERIODS. We note that we have made the ghost location γ explicit in these rules, who signify its role. In the model of atomic points-to, γ is used to store the ghost state to define the protocols (concurrent, single-writer, or CAS-only) for the atomic points-to ownership assertions. But intuitively, the ghost location γ uniquely identifies an *atomic access period* of the location ℓ. When we use the rule NA-AT-SW (or NA-AT-SW-VIEW) to switch from non-atomic to atomic points-to, we receive a *fresh* location γ that identifies and enforces the atomic protocol for the *current* atomic period of ℓ. As such,

the atomic local ownership and observations (Definition 10.2) with the ghost location γ are only meaningful when we still have access to the atomic points-to $\ell \overset{t}{\mapsto}^{\gamma}_{\theta}$ with the same γ. Once we use AT-NA to turn $\ell \overset{t}{\mapsto}^{\gamma}_{\theta}$ back to a non-atomic points-to, we can see that the atomic local ownership and observations with ghost location γ are not needed, and in fact γ is simply forgotten, and afterwards the atomic local ownership and observations of γ become meaningless. Later, when the non-atomic points-to is used again to switch to an atomic one, a new atomic period will be started with another fresh ghost location γ'. This life-cycle can probably be understood better if we look at the visualization graph in Figure 10.2 as an automaton.

We further note that all assertions appearing in a single rule in this chapter should be read with the same ghost location γ. It may not make sense to have interactions between ownership from different atomic periods, and we do not have rules for those cases anyway.

Property 10.5 (Conversions Between Modes of Atomic Points-To)**.** The rest of Figure 10.2 presents the rules for switching between different modes of the atomic points-to. AT-CON-SW and AT-SW-CON allow conversions between the concurrent mode and the single-writer mode of the atomic points-to, while AT-CAS-SW and AT-SW-CAS allow conversions between the CAS-only mode and the single-writer mode. Both AT-CON-SW and AT-CAS-SW need a basic update simply to update the exclusive single-writer timestamp to the latest one. Finally AT-CON-CAS allows one to convert between the concurrent mode and the CAS-only mode.

RULES FOR CONCURRENT ATOMIC ACCESSES. We next look at the rules for atomic operations using the atomic points-to assertion. We note that these rules are meant for concurrent accesses. If we simply use atomic accesses sequentially,[5] we should locally own an atomic points-to $\ell \mapsto_\theta h$, which we can turn back to a non-atomic one using AT-NA, and then use the rules NA-READ and NA-WRITE (§9.1) which support atomic access modes.[6]

In the following, we look at atomic access rules where the atomic points-to $\ell \mapsto_\theta h$ is meant to be shared for concurrent accesses. The rules do not enforce exactly how the points-to is shared—that can be done orthogonally with invariants (Chapter 11). Therefore, they only assume ownership of the atomic points-to under a view-at modality (Definition 8.13), *i.e.*, $@_{V_b}(\ell \mapsto_\theta h)$ where the view V_b has no clear relation to what the executing thread has observed. In particular, the rules will have the following form.

$$\{\sqsupseteq V * P * @_{V_b}(\ell \mapsto_\theta h)\}\, e \,\mathsf{in}\, \pi\, \{v.\, \exists V', h'.\, \sqsupseteq V' * @_{V_b \sqcup V'}(\ell \mapsto_\theta h') * Q\}_{\mathcal{E}}$$

That is, the view V_b is meant to track all relaxed memory effects done by all participating threads to the shared points-to $\ell \mapsto_\theta h$, and hence the executing thread cannot know much about V_b, except that it also contributes to V_b. Most importantly, the view V_b will be used later to enable synchronized, safe switching back to non-atomic points-to (using AT-NA).

[5] which can be the case in C/C++ where mixing atomic and non-atomic accesses are forbidden. Instead, C/C++ make the distinction at the location level: there are non-atomic locations and atomic ones. Under this restriction, even though we know that there is no other thread racing with us, we may still have to use a `rlx` access for an atomic location.

[6] If one is to perform sequential accesses, then one would never need to use CASes.

AT-READ-SN

$$\frac{\mathbf{rlx} \sqsubseteq o}{\{\sqsupseteq V_0 * \ell \sqsupseteq_{\mathtt{sn}} h_0 * @_{V_b}(\ell \overset{t_x}{\longmapsto}_\theta h)\}\ {}^{*o}\ell \text{ in } \pi \left\{ \begin{array}{l} v.\, \exists h', t, V, V' \sqsupseteq V_0.\, h_0 \subseteq h' \subseteq h * \\ \quad h'(t) = (v, V) * t \geq \mathtt{max}(\mathtt{dom}(h_0)) * \\ \quad (o \sqsupseteq \mathbf{acq})\ ?\ V \sqsubseteq V' : \nabla_\pi\,(\sqsupseteq V) \\ \quad \sqsupseteq V' * @_{V'}(\ell \sqsupseteq_{\mathtt{sn}} h') * @_{V_b \sqcup V'}(\ell \overset{t_x}{\longmapsto}_\theta h) \end{array} \right\}_\mathcal{E}}$$

AT-READ-SN-ACQ

$$\{\sqsupseteq V_0 * \ell \sqsupseteq_{\mathtt{sn}} h_0 * @_{V_b}(\ell \overset{t_x}{\longmapsto}_\theta h)\}\ {}^{*\mathbf{acq}}\ell \text{ in } \pi \left\{ \begin{array}{l} v.\, \exists h', t, V, V' \sqsupseteq V_0 \sqcup V.\, h_0 \subseteq h' \subseteq h * \\ \quad h'(t) = (v, V) * t \geq \mathtt{max}(\mathtt{dom}(h_0)) * \\ \quad \sqsupseteq V' * @_{V'}(\ell \sqsupseteq_{\mathtt{sn}} h') * @_{V_b \sqcup V'}(\ell \overset{t_x}{\longmapsto}_\theta h) \end{array} \right\}_\mathcal{E}$$

AT-READ-CAS

$$\frac{\mathbf{rlx} \sqsubseteq o}{\{\sqsupseteq V_0 * \ell \sqsupseteq^q_{\mathtt{cas}} h_0 * @_{V_b}(\ell \overset{t_x}{\longmapsto}_\theta h)\}\ {}^{*o}\ell \text{ in } \pi \left\{ \begin{array}{l} v.\, \exists h', t, V, V' \sqsupseteq V_0.\, h_0 \subseteq h' \subseteq h * \\ \quad h'(t) = (v, V) * t \geq \mathtt{max}(\mathtt{dom}(h_0)) * \\ \quad (o \sqsupseteq \mathbf{acq})\ ?\ V \sqsubseteq V' : \nabla_\pi\,(\sqsupseteq V) \\ \quad \sqsupseteq V' * @_{V'}(\ell \sqsupseteq^q_{\mathtt{cas}} h') * @_{V_b \sqcup V'}(\ell \overset{t_x}{\longmapsto}_\theta h) \end{array} \right\}_\mathcal{E}}$$

AT-READ-SY

$$\frac{\mathbf{rlx} \sqsubseteq o}{\{\sqsupseteq V_0 * \ell \sqsupseteq_{\mathtt{sy}} h * @_{V_b}(\ell \overset{t_x}{\longmapsto}_\theta h)\}\ {}^{*o}\ell \text{ in } \pi \left\{ \begin{array}{l} v.\, \exists t, V, V' \sqsupseteq V_0 \sqcup V.\, h'(t) = (v, V) * t = \mathtt{max}(\mathtt{dom}(h)) * \\ \quad \sqsupseteq V' * @_{V_b \sqcup V'}(\ell \overset{t_x}{\longmapsto}_\theta h) \end{array} \right\}_\mathcal{E}}$$

AT-READ-SW

$$\frac{\mathbf{rlx} \sqsubseteq o}{\{\sqsupseteq V_0 * \ell \sqsupseteq_{\mathtt{sw}} h * @_{V_b}(\ell \overset{t_x}{\longmapsto}_\theta h)\}\ {}^{*o}\ell \text{ in } \pi \left\{ \begin{array}{l} v.\, \exists t, V, V' \sqsupseteq V_0 \sqcup V.\, h(t) = (v, V) * t = \mathtt{max}(\mathtt{dom}(h)) * \\ \quad \sqsupseteq V' * \ell \sqsupseteq_{\mathtt{sw}} h * @_{V_b \sqcup V'}(\ell \overset{t_x}{\longmapsto}_\theta h) \end{array} \right\}_\mathcal{E}}$$

FIGURE 10.3: iRC11 read rules with the atomic points-to assertion

10.1.1 *Atomic Read Rules*

Several rules for atomic reads are given in Figure 10.3. AT-READ-SN, AT-READ-SY, AT-READ-CAS, and AT-READ-SW allow reading with a history-seen observation, a history-sync observation, a fractional CAS ownership, and a single-writer ownership, respectively, in addition to the shared atomic points-to.

AT-READ-SN is the most fundamental read rule for atomic points-to, as all other rules in Figure 10.3 are derived from it. The rule assumes in the pre-condition a local history-seen observation $\ell \sqsupseteq_{\mathtt{sn}} h_0$ for some snapshot history h_0 of ℓ, and the shared atomic points-to $\ell \overset{t_x}{\longmapsto}_\theta h$ of the current history h at some view V_b. The pre-condition also includes a view-seen observation $\sqsupseteq V_0$ for some view V_0. The post-condition says that the executing thread π will read a message (t, v, V) which is no earlier than what it has observed ($t \geq \mathtt{max}(\mathtt{dom}(h_0))$), and afterwards the thread will have observed a bigger snapshot history h' that contains the read message

$(\ell \sqsupseteq_{\mathtt{sn}} h')$. After the read, the current view of π will be at least V' $(\sqsupseteq V')$, and if this is an acquire read then we know that the thread has observed the read message view V, due to $V \sqsubseteq V'$ and VS-MONO (§8.5). We note that $\sqsupseteq V' * @_{V'}(\ell \sqsupseteq_{\mathtt{sn}} h')$ is stronger than $\ell \sqsupseteq_{\mathtt{sn}} h'$, due to VA-ELIM (§8.5). If this is a relaxed read, then the message view V will only be available after an acquire fence, *i.e.*, its observation is under an acquire fence modality: $\nabla_\pi(\sqsupseteq V)$. Last but not least, the atomic points-to $\ell \overset{t_x}{\longmapsto}_\theta h$ is returned unchanged, but at the view V_b extended with the view V' (*i.e.*, $V_b \sqcup V'$)—which is π's current view after the read—to account for the observation of the read itself (the action id created by the read).

AT-READ-SN-ACQ is derived from AT-READ-SN simply by instantiating o with **acq**. Since it is an acquire read, we know that the thread's current view V' includes the view V of the read message, *i.e.*, $V \sqsubseteq V'$. AT-READ-CAS is derived from AT-READ-SN, simply by the rule AT-CAS-SN that the CAS ownership implies the history-seen observation. AT-READ-SY is derived from AT-READ-SN using AT-SY-SN: assuming that the thread has observed and synchronized with all write events in ℓ's current history h, the thread will read the latest write. AT-READ-SW is then derived from AT-READ-SY using AT-SW-SY.

10.1.2 *Atomic Write Rules*

Several rules for atomic writes are given in Figure 10.4. AT-WRITE-SN, AT-WRITE-CAS, and AT-WRITE-SW allow reading with a history-seen observation, a full fractional CAS ownership, and a single-writer ownership, respectively, in addition to the shared atomic points-to (in a corresponding atomic mode).

Again, AT-WRITE-SN is the basic rule from which the other rules are derived. In the pre-condition, it requires a view-seen observation $\sqsupseteq V_0$ for some view V_0, a history-seen observation $\ell \sqsupseteq_{\mathtt{sn}} h_0$ for some snapshot history h_0 of ℓ, and the atomic points-to in the concurrent mode $\ell \overset{t_x}{\longmapsto}_{\mathtt{con}} h$, shared at some view V_b. The pre-condition additionally requires, in case the write is a relaxed write, a view-seen observation $\sqsupseteq V_{\mathtt{rel}}$ of some view $V_{\mathtt{rel}}$ *under* the release modality, *i.e.*, $\Delta_\pi(\sqsupseteq V_{\mathtt{rel}})$. This assertion ensures that the view $V_{\mathtt{rel}}$ has been observed by the thread π at its most recent release fence, so the message view of the relaxed write to be performed is guaranteed to include $V_{\mathtt{rel}}$. In other words, $V_{\mathtt{rel}}$ is a lower bound for the message view of the relaxed write to be performed. If the write to be performed is at least a release one, π's current view is a lower bound.

The post-condition of AT-WRITE-SN says that a new write message (t, v, V) will be inserted into the history h. That is, after the write, the ownership $\ell \overset{t_x}{\longmapsto}_{\mathtt{con}} h[t \leftarrow (v, V)]$ is returned, at the extended view $V_b \sqcup V'$ where V' is the thread π's current view after the write. Note that the timestamp t for the new write message must be fresh in h $(t \notin \mathtt{dom}(h))$, and must be mo-later than the events that the thread has observed for ℓ $(\mathtt{max}(\mathtt{dom}(h_0)) < t)$. Since this is the write, we know that the thread's current view V' after the step strictly extends the view V_0 before: $V_0 \sqsubset V'$. Furthermore, the message view V cannot be smaller than the view V_0 before the step $(V_0 \not\sqsupseteq V)$, because V contains at least the new timestamp

AT-WRITE-SN

$$\frac{\mathtt{rlx} \sqsubseteq o}{\left\{\begin{array}{l}\sqsupseteq V_0 * \ell \sqsupseteq_{\mathtt{sn}} h_0 * @_{V_b}(\ell \overset{t_x}{\longmapsto}_{\mathtt{con}} h) * \\ (o = \mathtt{rlx})\ ?\ \Delta_\pi\,(\sqsupseteq V_{\mathtt{rel}}) : \mathsf{True}\end{array}\right\} \ell :=_o v \text{ in } \pi \left\{\begin{array}{l}\text{☣}.\ \exists t, V, V' \sqsupset V_0 \not\sqsupseteq V.\, \mathtt{max}(\mathtt{dom}(h_0)) < t \notin \mathtt{dom}(h) * \\ \quad (\mathtt{rel} \sqsubseteq o)\ ?\ V = V' : V_{\mathtt{rel}} \sqsubseteq V \sqsubseteq V' * \\ \quad \sqsupseteq V' * @_V(\ell \sqsupseteq_{\mathtt{sn}} [t \leftarrow (v, V)]) * \\ \quad @_{V'}(\ell \sqsupseteq_{\mathtt{sn}} h_0[t \leftarrow (v, V)]) * @_{V_b \sqcup V'}(\ell \overset{t_x}{\longmapsto}_{\mathtt{con}} h[t \leftarrow (v, V)])\end{array}\right\}\mathcal{E}}$$

AT-WRITE-CAS

$$\frac{\mathtt{rlx} \sqsubseteq o}{\left\{\begin{array}{l}\sqsupseteq V_0 * \ell \sqsupseteq^q_{\mathtt{cas}} h_0 * @_{V_b}(\ell \mapsto_{\mathtt{cas}} h) * \\ (o = \mathtt{rlx})\ ?\ \Delta_\pi\,(\sqsupseteq V_{\mathtt{rel}}) : \mathsf{True}\end{array}\right\} \ell :=_o v \text{ in } \pi \left\{\begin{array}{l}\text{☣}.\ \exists t, V, V' \sqsupset V_0 \not\sqsupseteq V.\, \mathtt{max}(\mathtt{dom}(h_0)) < t \notin \mathtt{dom}(h) * \\ \quad (\mathtt{rel} \sqsubseteq o)\ ?\ V = V' : V_{\mathtt{rel}} \sqsubseteq V \sqsubseteq V' * \\ \quad \sqsupseteq V' * @_V(\ell \sqsupseteq_{\mathtt{sn}} [t \leftarrow (v, V)]) * \\ \quad @_{V'}(\ell \sqsupseteq^q_{\mathtt{cas}} h_0[t \leftarrow (v, V)]) * @_{V_b \sqcup V'}(\ell \mapsto_{\mathtt{cas}} h[t \leftarrow (v, V)])\end{array}\right\}\mathcal{E}}$$

AT-WRITE-SW

$$\frac{\mathtt{rlx} \sqsubseteq o}{\left\{\begin{array}{l}\sqsupseteq V_0 * \ell \sqsupseteq_{\mathtt{sw}} h * @_{V_b}(\ell \overset{t_x}{\longmapsto}_{\mathtt{sw}} h) * \\ (o = \mathtt{rlx})\ ?\ \Delta_\pi\,(\sqsupseteq V_{\mathtt{rel}}) : \mathsf{True}\end{array}\right\} \ell :=_o v \text{ in } \pi \left\{\begin{array}{l}\text{☣}.\ \exists t, V, V' \sqsupset V_0 \not\sqsupseteq V.\, \mathtt{max}(\mathtt{dom}(h)) < t * \\ \quad (\mathtt{rel} \sqsubseteq o)\ ?\ V = V' : V_{\mathtt{rel}} \sqsubseteq V \sqsubseteq V' * \sqsupseteq V' * \\ \quad @_{V'}(\ell \sqsupseteq_{\mathtt{sw}} h[t \leftarrow (v, V)]) * @_{V_b \sqcup V'}(\ell \overset{t}{\mapsto}_{\mathtt{sw}} h[t \leftarrow (v, V)])\end{array}\right\}\mathcal{E}}$$

AT-WRITE-SW-RLX

$$\left\{\begin{array}{l}\sqsupseteq V_0 * \ell \sqsupseteq_{\mathtt{sw}} h * @_{V_b}(\ell \overset{t_x}{\longmapsto}_{\mathtt{sw}} h) * \\ @_{V_{\mathtt{rel}}} P * \Delta_\pi\,(\sqsupseteq V_{\mathtt{rel}})\end{array}\right\} \ell :=_{\mathtt{rlx}} v \text{ in } \pi \left\{\begin{array}{l}\text{☣}.\ \exists t, V, V' \sqsupset V_0 \not\sqsupseteq V.\, \mathtt{max}(\mathtt{dom}(h)) < t * \\ \quad V_{\mathtt{rel}} \sqsubseteq V \sqsubseteq V' * \sqsupseteq V' * @_V P * \\ \quad @_{V'}(\ell \sqsupseteq_{\mathtt{sw}} h[t \leftarrow (v, V)]) * @_{V_b \sqcup V'}(\ell \overset{t}{\mapsto}_{\mathtt{sw}} h[t \leftarrow (v, V)])\end{array}\right\}\mathcal{E}$$

AT-WRITE-SW-RLX-SIMPLE

$$\{\ell \sqsupseteq_{\mathtt{sw}} h * @_{V_b}(\ell \overset{t_x}{\longmapsto}_{\mathtt{sw}} h) * \Delta_\pi\, P\}\ \ell :=_{\mathtt{rlx}} v \text{ in } \pi \left\{\begin{array}{l}\text{☣}.\ \exists t, V, V' \sqsupseteq V.\, \mathtt{max}(\mathtt{dom}(h)) < t * \sqsupseteq V' * @_V P * \\ \quad \ell \sqsupseteq_{\mathtt{sw}} h[t \leftarrow (v, V)] * @_{V_b \sqcup V'}(\ell \overset{t}{\mapsto}_{\mathtt{sw}} h[t \leftarrow (v, V)])\end{array}\right\}\mathcal{E}$$

AT-WRITE-SW-REL

$$\{\sqsupseteq V_0 * \ell \sqsupseteq_{\mathtt{sw}} h * @_{V_b}(\ell \overset{t_x}{\longmapsto}_{\mathtt{sw}} h) * P\}\ \ell :=_{\mathtt{rel}} v \text{ in } \pi \left\{\begin{array}{l}\text{☣}.\ \exists t, V \sqsupset V_0.\, \mathtt{max}(\mathtt{dom}(h)) < t * \sqsupseteq V * @_V P * \\ \quad @_V(\ell \sqsupseteq_{\mathtt{sw}} h[t \leftarrow (v, V)]) * @_{V_b \sqcup V}(\ell \overset{t}{\mapsto}_{\mathtt{sw}} h[t \leftarrow (v, V)])\end{array}\right\}\mathcal{E}$$

FIGURE 10.4: iRC11 write rules with the atomic points-to assertion

t which V_0 cannot have. In case this is at least a release write, then the message view V is exactly the view V' after the write. If it is only a relaxed write, we know the $V_{\mathtt{rel}}$ is a lower bound for V ($V_{\mathtt{rel}} \sqsubseteq V$). In any case the message view is no more than the thread's current view after the step ($V \sqsubseteq V'$). Finally, after the step the thread extends its observation on ℓ's history by the write event it has just performed. That is, it owns $\ell \sqsupseteq_{\mathtt{sn}} h_0[t \leftarrow (v, V)]$, with the history h_0 it has observed before the step extended with (t, v, V). The observation is strengthened by being put at the view V' which the thread has observed. Additionally, the thread also gets a history-seen observation for the singleton history of the write message (t, v, V), at exactly its message view ($@_V(\ell \sqsupseteq_{\mathtt{sn}} [t \leftarrow (v, V)])$).

AT-WRITE-CAS is derived from AT-WRITE-SN, using AT-CAS-SN and an

extra ghost update to update the CAS ownership from h_0 to $h_0[t \leftarrow (v, V)]$. This is also the case for AT-WRITE-SW, but using AT-SW-SY and AT-SY-SN instead. However, AT-WRITE-SW does update the latest exclusive single-writer timestamp to the new timestamp t.

RESOURCE TRANSFER. AT-WRITE-SW-RLX, AT-WRITE-SW-RLX-SIMPLE, and AT-WRITE-SW-REL are all derived from AT-WRITE-SW, and demonstrate the support for transferring the resource P with a write, a typical pattern in GPS, RSL, FSL, and Cosmo.[7] AT-WRITE-SW-RLX specializes AT-WRITE-SW for the relaxed write case, and assumes P at the view $V_{\texttt{rel}}$ which we know will be a lower bound the new write's message view. Consequently, after the write, by view-monotonicity P holds at the new write's message view V, *i.e.*, we have $@_V P$. As such, we have released the resource P with the write, and we can attach it to the message (t, v, V) with the help of an invariant (see Chapter 11). Then, when another thread performs an acquire read (or a relaxed read and then an acquire fence) from (t, v, V), by the rule AT-READ-SN-ACQ, the reading thread will obtain $\sqsupseteq V$, which it then can combine with $@_V P$ (taken from the invariant) and apply the rule VA-ELIM (§8.5) to acquire the resource P locally, and thus conclude the resource transfer.

[7] [VN13; TVD14; DV16; DV17; Kai+17; MJP20].

AT-WRITE-SW-RLX-SIMPLE simplifies AT-WRITE-SW-RLX further by dropping most of the views. It is derived from AT-WRITE-SW-RLX with the help of the rule RELMOD-VA-REVERT (§8.5) concerning the relation between the release modality and the view-at modality.

For a release write, AT-WRITE-SW-REL simply says that any local resource P before the step can be released with the write, by putting P at the new write's message view V after the step.

10.1.3 *Atomic CAS Rules*

The general rule AT-CAS-SN-GEN in Figure 10.5 allows us to perform CASes with the seen-history observation and the shared atomic points-to assertion in the CAS-only mode. The pre-condition is not so different from the pre-condition needed to perform a write, *i.e.*, that of AT-WRITE-CAS. The extra premise $\forall v_0, t_0 \geq \mathtt{max}(\mathtt{dom}(h_0)).\, h(t_0) = (v_0, \_) \Rightarrow \vdash v_0 =^? v_r$ is required to guarantee safe comparison between any *readable* value v_0 and the expected value v_r, and the resources concern P_{cmp} are needed for deterministic pointer comparison. Please see also the explanation of the base logic rule BL-HOARE-CAS in §7.3.3.

In particular, if the expected value v_r is a location value ℓ_r, to guarantee deterministic pointer comparison, P_{cmp} is required to simultaneously imply (with a basic update) $\Phi_{\mathrm{cmp}}(\ell_r, h)$ that (1) some primitive non-atomic points-to ownership of ℓ_r at an arbitrary view, sufficient to show that ℓ_r is alive; and (2) for any location value ℓ' that the thread may read from h, P_{cmp} is also sufficient to show that ℓ' is also alive.

In the post-condition, a boolean value b signaling the success or failure of the CAS instruction is returned, and a message (t', v', V_r) will be read by the instruction. The timestamp t' cannot be earlier than what the thread has already observed through h_0. Regardless of success or failure,

AT-CAS-SN-GEN

$$\frac{\mathbf{rlx} \sqsubseteq o_f, o_r, o_w \qquad \forall v_0, t_0 \geq \mathtt{max}(\mathtt{dom}(h_0)).\, h(t_0) = (v_0, \_) \Rightarrow \vdash v_0 =^? v_r}{\left\{\begin{array}{l} \sqsupseteq V_0 * \ell \sqsupseteq_{\mathtt{sn}} h_0 * @_{V_b}(\ell \overset{t_x}{\longmapsto}_{\mathtt{cas}} h) * (o = \mathbf{rlx})\ ?\ \Delta_\pi\,(\sqsupseteq V_{\mathbf{rel}}) : \mathsf{True} * \\ P_{\mathrm{cmp}} * \square\left((v_r = \ell_r)\ ?\ (P_{\mathrm{cmp}} \dot{\Rrightarrow} \Phi_{\mathrm{cmp}}(\ell_r, h)) : \mathsf{True}\right) \end{array}\right\}}$$

$$\mathbf{CAS}^{o_f, o_r, o_w}(\ell, v_r, v_w) \text{ in } \pi$$

$$\left\{\begin{array}{l} b.\, \exists t', v', V_r, V' \sqsupseteq V_0, h'_0, h'.\, h_0 \subseteq h'_0 \subseteq h' * h'_0(t') = (v', V_r) * \mathtt{max}(\mathtt{dom}(h_0)) \leq t' * \\ \quad P_{\mathrm{cmp}} * \sqsupseteq V' * @_{V'}(\ell \sqsupseteq_{\mathtt{sn}} h'_0) * @_{V_b \sqcup V'}(\ell \overset{t_x}{\longmapsto}_{\mathtt{cas}} h') * \\ \quad \vee \left\{\begin{array}{ll} b = \mathbf{false} & * v_r \neq v' * h' = h * (o_f \sqsupseteq \mathbf{acq})\ ?\ V_r \sqsubseteq V' : \nabla_\pi\,(\sqsupseteq V_r) \\ b = \mathbf{true} & * v_r = v' * \exists t \notin \mathtt{dom}(h), V_w \sqsupseteq V_r \not\sqsupseteq V' \sqsupseteq V_0.\, t = t' + 1 \\ & \quad * h' = h[t \leftarrow (v_w, V_w)] * h'_0(t) = (v_w, V_w) \\ & \quad * (o_r \sqsupseteq \mathbf{acq})\ ?\ V_w \sqsubseteq V' : \nabla_\pi\,(\sqsupseteq V_w) * (o_w \sqsupseteq \mathbf{rel})\ ?\ V' \sqsubseteq V_w : V_{\mathbf{rel}} \sqsubseteq V_w \end{array}\right. \end{array}\right\}_{\mathcal{E}}$$

where

$$\Phi_{\mathrm{cmp}}(\ell_r, h) ::= (\exists q_r, h_r, V.\, \triangleright @_V(\ell_r \overset{q_r}{\longmapsto} h_r)) \wedge (\forall t' \geq \mathtt{dom}(h), \ell'.\, h(t') = (\ell', \_) \twoheadrightarrow \exists q', h', V.\, \triangleright @_V(\ell' \overset{q'}{\longmapsto} h'))$$

FIGURE 10.5: An iRC11 CAS rule with the atomic points-to assertion

P_{cmp} is returned unchanged, and we know that the thread's current view after the step is V'.

In case of failure, *i.e.*, $b = \mathbf{false}$, we know that the read value v' is definitely not equal to the expected value v_r, and that the atomic points-to ownership is returned unchanged but at the extended view $V_b \sqcup V'$ ($@_{V_b \sqcup V'}(\ell \overset{t_x}{\longmapsto}_{\mathtt{cas}} h)$), and that the thread will have observed the new snapshot history h'_0 ($@_{V'}(\ell \sqsupseteq_{\mathtt{sn}} h'_0)$). Furthermore, the read message view V_r is observed according to the failure read access mode o_f. If o_f is at least an acquire mode, then V_r is included in the view V' after the step ($V_r \sqsubseteq V'$). Otherwise, the observation of V_r is only available after the next acquire fence ($\nabla_\pi \sqsupseteq V_r$).

If the CAS succeeds, *i.e.*, $b = \mathbf{true}$, then $v' = v_r$ and a new write message (t, v_w, V_w) will be inserted into the history h next to the read message ($t = t' + 1$) where t is fresh in h ($t \notin \mathtt{dom}(h)$). V_w strictly extends V_r, and V' strictly extends V_0 and cannot be smaller than V_r, because they contain the new write of the timestamp t. The thread will have observed the new snapshot history h'_0 that contains the new write event. The write message view V_w is observed according to the read access mode o_r: if o_r is at least an acquire mode, then $V_w \sqsubseteq V'$, otherwise the thread only has $\nabla_\pi\,(\sqsupseteq V_w)$. If the write access mode o_w is at least release, then the write message view includes the thread's current view after the step ($V' \sqsubseteq V_w$), otherwise $V_{\mathbf{rel}}$ is a lower bound of V_w. Note that if this is a release-acquire CAS, then the write message view V_w is exactly the thread's current view after the step V', *i.e.*, $o_w \sqsupseteq \mathbf{rel} \wedge o_r \sqsupseteq \mathbf{acq} \Rightarrow V' = V_w$.

Finally, we note that AT-CAS-SN-GEN can be used to derive CAS rules that use other kinds of ownership. Even though the rule requires the atomic points-to to be in the CAS-only mode (`cas`), we can apply AT-CON-CAS (Figure 10.2) to support CASes with the atomic points-to in the

AT-CAS-SW-GEN

$$\frac{\texttt{rlx} \sqsubseteq o_f, o_r, o_w \qquad \forall v_0, t_0 \geq \texttt{max}(\texttt{dom}(h_0)).\, h(t_0) = (v_0, \_) \Rightarrow \vdash v_0 =^? v_r}{\left\{\begin{array}{l} \sqsupseteq V_0 * \ell \sqsupseteq_{\texttt{sn}} h_0 * @_{V_c}(\ell \sqsupseteq_{\texttt{sw}} h) * @_{V_b}(\ell \mapsto_{\texttt{sw}} h) * (o = \texttt{rlx})\,?\,\Delta_\pi\,(\sqsupseteq V_{\texttt{rel}}) : \mathsf{True} \\ P_{\mathrm{cmp}} * \square\left((v_r = \ell_r)\,?\,(P_{\mathrm{cmp}} \dot{\Rrightarrow} \Phi_{\mathrm{cmp}}(\ell_r, h)) : \mathsf{True}\right) \end{array}\right\}}$$

$$\texttt{CAS}^{o_f, o_r, o_w}(\ell, v_r, v_w) \text{ in } \pi$$

$$\left\{\begin{array}{l} b.\, \exists t', v', V_r, V' \sqsupseteq V_0, h_0', h'.\, h_0 \subseteq h_0' \subseteq h' * h_0'(t') = (v', V_r) * \texttt{max}(\texttt{dom}(h_0)) \leq t' * \\ \quad P_{\mathrm{cmp}} * \sqsupseteq V' * @_{V'}(\ell \sqsupseteq_{\texttt{sn}} h_0') * @_{V_b \sqcup V'}(\ell \sqsupseteq_{\texttt{sw}} h' * \ell \mapsto_{\texttt{sw}} h') * \\ \quad \vee \left\{\begin{array}{ll} b = \texttt{false} & * v_r \neq v' * h' = h * (o_f \sqsupseteq \texttt{acq})\,?\,V_r \sqsubseteq V' : \nabla_\pi\,(\sqsupseteq V_r) \\ b = \texttt{true} & * v_r = v' * \exists t \notin \texttt{dom}(h), V_w \sqsupset V_r \not\sqsupseteq V' \sqsupset V_0.\, t = t' + 1 \\ & \quad * h' = h[t \leftarrow (v_w, V_w)] * h_0'(t) = (v_w, V_w) \\ & \quad * (o_r \sqsupseteq \texttt{acq})\,?\,V_w \sqsubseteq V' : \nabla_\pi\,(\sqsupseteq V_w) * (o_w \sqsupseteq \texttt{rel})\,?\,V' \sqsubseteq V_w : V_{\texttt{rel}} \sqsubseteq V_w \end{array}\right. \end{array}\right\}_{\mathcal{E}}$$

where $\Phi_{\mathrm{cmp}}(\ell_r, h)$ is defined as in Figure 10.5.

FIGURE 10.6: An iRC11 CAS rule with the atomic points-to in single-writer mode

concurrent mode `con`. With a fractional CAS ownership, we can also apply AT-CAS-SN-GEN, thanks to AT-CAS-SN.

If the atomic points-to is in the single-writer mode (`sw`), then, together with the single-writer ownership $\ell \sqsupseteq_{\texttt{sw}} h$, we can get to the atomic points-to in CAS-only mode thanks to AT-SW-CAS (also in Figure 10.2), then apply AT-CAS-SN-GEN, and then AT-CAS-SW to go back to the single-writer mode. The result is the CAS rule AT-CAS-SW-GEN for the single-writer mode, in Figure 10.6. Naturally, a single-writer owner never needs to perform a CAS, because it is not racing in writing with anyone. Furthermore, the rule is basically useless, as it requires and returns the single-writer ownership at some view V_c and $V_b \sqcup V'$, respectively. Nevertheless, the rule is a sanity check that shows that, in the single-writer mode, only the single-writer owner can actually perform a write (in this case, a CAS).

10.2 The Model of the Atomic Points-To Assertion

To give a model for the atomic points-to assertion, we rely on the base logic local assertions (§7.4), in a similar way to the model of non-atomic points-to assertion (§9.2). However, we need extra ghost state to manage the "switching" protocols among atomic modes and between the atomic points-to to the non-atomic points-to. The ghost location γ of the assertion $\ell \overset{t_x}{\longmapsto}{}^{\gamma}_{\theta}$, which uniquely identifies an atomic period for ℓ, will store this extra ghost state.

Definition 10.6 (Extra RAs for Atomic Points-To)**.** We need 3 RAs: one to allow creating snapshots of histories, one to store the latest non-atomic view—needed to switch between non-atomic and atomic points-to, and one to store the timestamp of the latest exclusive single-writer write—

needed to switch from other modes to single-writer mode.

$$\begin{aligned}
\textsc{atHistR} &::= \textsc{Auth}(\textsc{Map}(\mathit{Time}, \textsc{Ag}(\mathit{Val} \times \mathit{View}))) \\
\textsc{naWriteR} &::= \textsc{Option}(\textsc{Ag}(\mathit{View})) \\
\textsc{exWriteR} &::= \textsc{Auth}(\textsc{Option}(\textsc{Frac} \times \textsc{Ag}(\mathit{Time}))) \\
\textsc{atomicR} &::= \textsc{atHistR} \times \textsc{exWriteR} \times \textsc{naWriteR}
\end{aligned}$$

The RA atHistR supports making snapshots of histories: the authoritative element is used to store the up-to-date history, while fragmentary elements are snapshots of that history. The authoritative element can only grow.

The RA naWriteR allows storing permanently a view which is meant to be a thread's current view V_{na} at which the switch from non-atomic to atomic points-to is performed. All subsequent atomic accesses using the same atomic points-to identified by the ghost location γ will have synchronized with V_{na}.

The RA exWriteR allows for fractional fragmentary elements that agree on the timestamp of the latest exclusive single-writer write, and only allows updating using the full fraction, together with the authoritative element.

Finally, the RA atomicR for the atomic points-to is just a product of the 3 RAs above.

Definition 10.7 (Ghost Ownership Abstraction for Atomic RAs)**.** We define the following abstractions for ghost ownership of the atomic RAs in vProp. All are timeless and objective.

$$\begin{aligned}
\mathsf{atLastNA}^{\gamma}(V_{\mathsf{na}}) &::= \boxed{(\varepsilon, \varepsilon, \mathsf{ag}(V_{\mathsf{na}}))}^{\gamma} \\
\mathsf{atExclTime}^{\gamma}_{q}(t_x) &::= \boxed{(\varepsilon, \circ\, (\mathsf{Some}(q, \mathsf{ag}(t_x))), \varepsilon)}^{\gamma} \\
\mathsf{atReader}^{\gamma}(h) &::= \boxed{(\circ\, h, \varepsilon, \varepsilon)}^{\gamma} \\
\mathsf{atWriter}^{\gamma}(h) &::= \boxed{(\bullet_{3/4} h \cdot \circ\, h, \varepsilon, \varepsilon)}^{\gamma} \\
\mathsf{atAuth}^{\gamma}(h, t_x, V_{\mathsf{na}}) &::= \boxed{(\bullet_{1/4} h \cdot \circ\, h, \bullet\, (\mathsf{Some}(1, \mathsf{ag}(t_x))), \mathsf{ag}(V_{\mathsf{na}}))}^{\gamma}
\end{aligned}$$

- $\mathsf{atLastNA}^{\gamma}(V_{\mathsf{na}})$ is persistent, and records the view at the point of the switch from a non-atomic points-to to the atomic period identified by γ.
- $\mathsf{atExclTime}^{\gamma}_{q}(t_x)$ is fractional and records the timestamp t_x of the latest exclusive single-writer write in the atomic period identified by γ. The full fraction $\mathsf{atExclTime}^{\gamma}(t_x)$, which a single-writer would own, is the *exclusive write permission* needed to update t_x.
- $\mathsf{atReader}^{\gamma}(h)$ is persistent and records a snapshot history h, *i.e.*, in the atomic period of γ, h is a lower bound of the current history.
- $\mathsf{atWriter}^{\gamma}(h)$ is exclusive and records the current history h. It is the (ghost) *writer permission* needed to perform a write (a change) to the history.
- $\mathsf{atAuth}^{\gamma}(h, t_x, V_{\mathsf{na}})$ is the authoritative state of the atomic points-to protocol. The other ghost ownership abstractions defined above

$$\mathsf{persistent}(\mathsf{atLastNA}^\gamma(V_{\mathsf{na}})) \qquad \mathsf{atLastNA}^\gamma(V_{\mathsf{na}}) * \mathsf{atLastNA}^\gamma(V'_{\mathsf{na}}) \vdash V_{\mathsf{na}} = V'_{\mathsf{na}}$$
$$\mathsf{persistent}(\mathsf{atReader}^\gamma(h))$$

$$\mathsf{atExclTime}^\gamma_q(t_x) * \mathsf{atExclTime}^\gamma_{q'}(t'_x) \vdash t_x = t'_x \land q + q' \in (0, 1]$$

$$\mathsf{atExclTime}^\gamma_q(t_x) * \mathsf{atExclTime}^\gamma_{q'}(t_x) \dashv\vdash \mathsf{atExclTime}^\gamma_{q+q'}(t_x)$$

$$\mathsf{atWriter}^\gamma(h) * \mathsf{atReader}^\gamma(h') \vdash h' \subseteq h \qquad \mathsf{atWriter}^\gamma(h) * \mathsf{atWriter}^\gamma(h') \vdash \mathsf{False} \qquad \mathsf{atWriter}^\gamma(h) \vdash \mathsf{atReader}^\gamma(h)$$

$$\mathsf{atAuth}^\gamma(h, t_x, V_{\mathsf{na}}) * \mathsf{atLastNA}^\gamma(V'_{\mathsf{na}}) \vdash V_{\mathsf{na}} = V'_{\mathsf{na}} \qquad \mathsf{atAuth}^\gamma(h, t_x, V_{\mathsf{na}}) * \mathsf{atExclTime}^\gamma_q(t'_x) \vdash t_x = t'_x$$

$$\mathsf{atAuth}^\gamma(h, t_x, V_{\mathsf{na}}) * \mathsf{atReader}^\gamma(h') \vdash h' \subseteq h \qquad \mathsf{atAuth}^\gamma(h, t_x, V_{\mathsf{na}}) * \mathsf{atWriter}^\gamma(h') \vdash h' = h$$

FIGURE 10.7: Several properties of ghost abstractions for the atomic RA

must agree or be included in this authoritative state. We note that we use a setup where the authoritative element is also fractional ($\bullet_{1/4}$ in atAuth^γ, or $\bullet_{3/4}$ in $\mathsf{atWriter}^\gamma$). Fractions of the authoritative element enjoy agreement, and that is how we establish agreement between $\mathsf{atWriter}^\gamma$ and atAuth^γ.

Several properties of these ghost abstractions are given in Figure 10.7.

We can now give the model of the atomic points-to assertion as well as its local ownership and observation assertions.

Definition 10.8 (Model of Atomic Local Ownership and Observations)**.** We first define what it means to locally observe a history h of ℓ, and to locally synchronize with that history.

$$\mathsf{Local}_{\mathtt{sn}}(\ell, h) ::= \forall t, v, V^?.\, h(t) = (v, V^?) \Rightarrow \sqsupseteq[\ell \leftarrow \{\varnothing\,[\mathsf{w} := t]\}]$$
$$\mathsf{Local}_{\mathtt{sy}}(\ell, h) ::= \forall t, v, V^?.\, h(t) = (v, V^?) \Rightarrow \sqsupseteq V^? * \sqsupseteq[\ell \leftarrow \{\varnothing\,[\mathsf{w} := t]\}]$$

That is, the owner of $\mathsf{Local}_{\mathtt{sn}}(\ell, h)$ should have observed the timestamps of the writes in h, while the owner of $\mathsf{Local}_{\mathtt{sy}}(\ell, h)$ should additionally have observed the message views of those writes. The model of the atomic local ownership and observations is then given within vProp, as follows.

$$\ell \sqsupseteq^\gamma_{\mathtt{sn}} h ::= \mathsf{Local}_{\mathtt{sn}}(\ell, h) * \mathsf{atReader}^\gamma(h) * \exists V_{\mathsf{na}}.\, \mathsf{atLastNA}^\gamma(V_{\mathsf{na}}) * \sqsupseteq V_{\mathsf{na}}$$
$$\ell \sqsupseteq^\gamma_{\mathtt{sy}} h ::= \mathsf{Local}_{\mathtt{sy}}(\ell, h) * \mathsf{atReader}^\gamma(h) * \exists V_{\mathsf{na}}.\, \mathsf{atLastNA}^\gamma(V_{\mathsf{na}}) * \sqsupseteq V_{\mathsf{na}}$$
$$\ell \sqsupseteq^{\gamma, t_x}_{\mathtt{sw}} h ::= \ell \sqsupseteq^\gamma_{\mathtt{sy}} h * \mathsf{atWriter}^\gamma(h) * \mathsf{atExclTime}^\gamma(t_x) * t_x = \mathtt{max}(\mathtt{dom}(h))$$
$$\ell \sqsupseteq^{\gamma, t_x, q}_{\mathtt{cas}} h ::= \ell \sqsupseteq^\gamma_{\mathtt{sn}} h * \mathsf{atExclTime}^\gamma_q(t_x) * t_x \in \mathtt{dom}(h)$$

- The seen-history observation $\ell \sqsupseteq^\gamma_{\mathtt{sn}} h$ requires that the h is indeed a snapshot history of ℓ's atomic points-to identified by γ ($\mathsf{atReader}^\gamma(h)$), and that the owner has observed the writes in h ($\mathsf{Local}_{\mathtt{sn}}(\ell, h)$). The remaining part $\exists V_{\mathsf{na}}.\, \mathsf{atLastNA}^\gamma(V_{\mathsf{na}}) * \sqsupseteq V_{\mathsf{na}}$ says that the owner has observed the view V_{na} of the switch from the non-atomic points-to to the atomic points-to identified by γ.
- The sync-history observation $\ell \sqsupseteq^\gamma_{\mathtt{sn}} h$ is similar to the seen-history observation, but additionally requires the synchronization with all the message views in h ($\mathsf{Local}_{\mathtt{sy}}(\ell, h)$).

- The single-writer ownership $\ell \sqsupseteq^{\gamma,t_x}_{\mathtt{sw}} h$ requires the sync-history observation $\ell \sqsupseteq^{\gamma}_{\mathtt{sn}} h$, and holds the writer permission $\mathsf{atWriter}^{\gamma}(h)$ for the current history h as well as the exclusive writer permission $\mathsf{atExclTime}^{\gamma}(t_x)$ for the timestamp t_x of the latest exclusive single-writer write. That is, the single-writer ownership holds both the permissions to update h and t_x. Additionally, we know that t_x is the maximum timestamp in the current history h.
- The CAS ownership $\ell \sqsupseteq^{\gamma,t_x,q}_{\mathtt{cas}} h$ only requires the seen-history observation, and its fraction q is the fraction for exclusive single-writer timestamp $\mathsf{atExclTime}^{\gamma}_{q}(t_x)$, which is sufficient to prevent others from updating t_x (and thus prevent any single-writer writes). Another requirement is that the owner has observed t_x ($t_x \in \mathtt{dom}(h)$).

Definition 10.9 (Model of the Atomic Points-To)**.** We now give the model of the atomic points-to assertions, also within vProp. It relies on a "lift-view" function $\mathtt{liftV}(h, V_{\mathsf{na}})$ that lifts all h's message views to include the view V_{na} of the "non-atomic to atomic" switch.

$$\mathtt{liftV}(h, V_{\mathsf{na}}) ::= \left[t \leftarrow (v, V^? \sqcup V_{\mathsf{na}}) \;\middle|\; h(t) = (v, V^?)\right]$$

$$\begin{aligned}
\ell \overset{t_x}{\longmapsto}^{\gamma}_{\theta} h ::= \exists h', \alpha_w, \alpha_1, \alpha_2, V_{\mathsf{na}}.\ & h = \mathtt{liftV}(h', V_{\mathsf{na}}) * t_x \in \mathtt{dom}(h) \\
& * \mathsf{Local}_{\mathtt{sy}}(\ell, h) * \mathsf{Local}^{\sqsupseteq \mathtt{rlx}}_{\mathtt{W}}(\ell, \alpha_w) \\
& * \mathsf{Local}^{\sqsupseteq \mathtt{rlx}}_{\mathtt{R}}(\ell, \alpha_2) * \mathsf{Local}^{\mathsf{na}}_{\mathtt{R}}(\ell, \alpha_1, V_{\mathsf{na}}) \\
& * \mathsf{Hist}(\ell, h') * \mathsf{Write}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha_w) \\
& * \mathsf{Read}^{\mathsf{na}}(\ell, \alpha_1) * \mathsf{Read}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha_2) \\
& * \mathsf{atAuth}^{\gamma}(h, t_x, V_{\mathsf{na}}) \\
& * \begin{cases} \theta = \mathtt{sw} & * \mathsf{True} \\ \theta = \mathtt{cas} & * \mathsf{atWriter}^{\gamma}(h) \\ \theta = \mathtt{con} & * \mathsf{atWriter}^{\gamma}(h) * \mathsf{atExclTime}^{\gamma}(t_x) \end{cases}
\end{aligned}$$

The model of the atomic points-to $\ell \overset{t_x}{\longmapsto}^{\gamma}_{\theta} h$ is very similar to that of the non-atomic points-to:[8] it requires *full* ownership of (1) the base logic assertions for history ownership ($\mathsf{Hist}(\ell, h')$) of the unlifted history h' ($h = \mathtt{liftV}(h', V_{\mathsf{na}})$) and of (2) the difference parts of the race detector state (the atomic writes set $\mathsf{Write}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha_w)$, and the non-atomic and atomic reads sets $\mathsf{Read}^{\mathsf{na}}(\ell, \alpha_1)$ and $\mathsf{Read}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha_2)$).[9] Similarly, it requires the base logic's local observations for all of those sets.[10] It also requires the observation of the non-atomic view V_{na} of the switch as well as that V_{na} has observed the non-atomic reads set, *i.e.*, $\mathsf{Local}^{\mathsf{na}}_{\mathtt{R}}(\ell, \alpha_1, V_{\mathsf{na}})$.

[8] see Definition 9.3

[9] see also Definition 7.1 and Definition 9.1

[10] Note that $\mathsf{Local}_{\mathtt{sy}}(\ell, h)$ implies the allocation observation $\mathsf{Local}_{\mathtt{A}}(\ell, h)$.

The main difference with the non-atomic points-to is the ghost ownership. The atomic points-to owns the authoritative state $\mathsf{atAuth}^{\gamma}(h, t_x, V_{\mathsf{na}})$ to hold the authority over the local ownership and observations (given model in Definition 10.8). It further owns the remaining ghost permissions for different atomic modes: (i) for the single-writer mode `sw` it owns nothing more, because all permissions are owned by the single-writer ownership; (ii) for the CAS-only mode `cas` it owns the writer permission $\mathsf{atWriter}^{\gamma}(h)$ so as to allow concurrent CASes to update the history, but it does not own the exclusive write permission, which is owned by

the fractions of CAS ownership themselves; and (iii) for the arbitrarily concurrent mode con it owns all the ghost permissions, as the clients of the con mode only have the seen-history or sync-history observations to work with ℓ. Recall that the atomic points-to assertion is meant to be shared for concurrent accesses, and participants rely on their local atomic ownership and observations to relate themselves to the shared history h and thus to strengthen the behaviors of their own instructions.

We now sketch the proofs of several important rules.

10.2.1 *Proof Sketches for Conversions between Non-Atomic and Atomic Points-To*

Proof sketch of NA-AT-SW-VIEW. This proof is done entirely within vProp. We start by first freezing $\ell \mapsto v * P$ at some view V_0 using VA-INTRO (§8.5). Then we unfold the model of non-atomic points-to (Definition 9.3). We note that the local observations of the non-atomic points-to are not objective, while the history ownership and the race detector state ownership are objective, so the view-at modality can be easily eliminated using VA-OBJ (§8.5). Our goal then looks as follows.

Context:	Goal:
$\sqsupseteq V_0 * @_{V_0} P * @_{V_0} \sqsupseteq V$	
$@_{V_0}(\mathsf{Local}_{\mathtt{A}}(\ell, [t \leftarrow (v, V)]) * \mathsf{Local}_{\mathtt{W}}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha_w))$	
$@_{V_0}(\mathsf{Local}_{\mathtt{R}}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha_2) * \mathsf{Local}_{\mathtt{R}}^{\mathtt{na}}(\ell, \alpha_1, V_{\mathtt{na}}))$	
$\mathsf{Hist}(\ell, [t \leftarrow (v, V)]) * \mathsf{Write}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha_w)$	
$\mathsf{Read}^{\mathtt{na}}(\ell, \alpha_1) * \mathsf{Read}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha_2)$	
	$\exists \gamma, t, V.\ \sqsupseteq V * @_V (P * \ell \sqsupseteq^{\gamma}_{\mathtt{sw}} [t \leftarrow (v, V)] * \ell \overset{t}{\mapsto}^{\gamma}_{\mathtt{sw}} [t \leftarrow (v, V)])$

From $@_{V_0}\mathsf{Local}_{\mathtt{R}}^{\mathtt{na}}(\ell, \alpha_1, V_{\mathtt{na}})$ and the vProp definition of $\mathsf{Local}_{\mathtt{R}}^{\mathtt{na}}$ (Definition 9.1), we have $V_{\mathtt{na}} \sqsubseteq V_0$.

We then allocate a new ghost location γ for the RA ATOMICR, using GHOST-ALLOC (§6.2), with the initial history $h = [t \leftarrow (v, V \sqcup V_0)]$, the exclusive write timestamp t, and the new non-atomic view V_0. This allocation will give us these following extra ownership $\mathsf{atAuth}^{\gamma}(h, t, V_0) * \mathsf{atLastNA}^{\gamma}(V_0) * \mathsf{atExclTime}^{\gamma}(t) * \mathsf{atWriter}^{\gamma}(h)$.

We note that from $@_{V_0} \sqsupseteq V$, by VA-VS (§8.5), we have $V \sqsubseteq V_0$. Consequently, $V \sqcup V_0 = V_0$. We then instantiate the existential quantifiers respectively with γ, t, and V_0. Note how V_0, the view of the switch, indeed becomes the new message view for t. We can easily discharge $\sqsupseteq V$ and $@_{V_0} P$, and then arrive at the following goal.

Context:	Goal:
$\sqsupseteq V_0 * @_{V_0} \sqsupseteq V$	
$@_{V_0}(\mathsf{Local}_{\mathtt{A}}(\ell, [t \leftarrow (v, V)]) * \mathsf{Local}_{\mathtt{W}}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha_w))$	
$@_{V_0}(\mathsf{Local}_{\mathtt{R}}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha_2) * \mathsf{Local}_{\mathtt{R}}^{\mathtt{na}}(\ell, \alpha_1, V_{\mathtt{na}}))$	
$\mathsf{Hist}(\ell, [t \leftarrow (v, V)]) * \mathsf{Write}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha_w)$	
$\mathsf{Read}^{\mathtt{na}}(\ell, \alpha_1) * \mathsf{Read}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha_2)$	
$\mathsf{atAuth}^{\gamma}(h, t, V_0) * \mathsf{atLastNA}^{\gamma}(V_0)$	
$\mathsf{atExclTime}^{\gamma}(t) * \mathsf{atWriter}^{\gamma}(h)$	$@_{V_0}(\ell \sqsupseteq^{\gamma}_{\mathtt{sw}} h * \ell \overset{t}{\mapsto}^{\gamma}_{\mathtt{sw}} h)$

By unfolding the definitions of the atomic points-to and the single-writer ownership, and then discharge all available assumptions, we arrive at the goal:

Context:	Goal:
$V \sqsubseteq V_0 * V_{\mathsf{na}} \sqsubseteq V_0 * @_{V_0} \mathsf{Local}^{\mathsf{na}}_{\mathtt{R}}(\ell, \alpha_1, V_{\mathsf{na}})$	$@_{V_0}(\ell \sqsupseteq^{\gamma}_{\mathtt{sy}} h * \mathsf{Local}_{\mathtt{sy}}(\ell, h) * \mathsf{Local}^{\mathsf{na}}_{\mathtt{R}}(\ell, \alpha_1, V_0))$

This is easily done because h is the singleton $[t \leftarrow (v, V_0)]$, and $\mathsf{Local}^{\mathsf{na}}_{\mathtt{R}}$ is view monotone, so $\mathsf{Local}^{\mathsf{na}}_{\mathtt{R}}(\ell, \alpha_1, V_{\mathsf{na}})$ implies $\mathsf{Local}^{\mathsf{na}}_{\mathtt{R}}(\ell, \alpha_1, V_0)$ knowing that $V_{\mathsf{na}} \sqsubseteq V_0$. □

Proof sketch of AT-NA. The proof is rather straightforward: the model of the atomic points-to, after dropping the atomic ghost state abstractions, is almost the same as the model of the non-atomic points-to, except for the history h'. We then use BL-HIST-DROP-SINGLETON (which needs the fancy update, see Figure 7.2) to truncate h' to the singleton of its latest write. After sorting out the local observations, we are done. □

10.2.2 *Proof Sketches for Conversions among Atomic Modes*

Proof sketch of AT-SW-CAS. The proof is straightforward—its main proof step is to move the write permission $\mathsf{atWriter}^{\gamma}(h)$ from the single-writer ownership into the CAS-only atomic points-to. □

Proof sketch of AT-CAS-SW. The proof is also straightforward—it has 2 main proof steps. First, we move the write permission $\mathsf{atWriter}^{\gamma}(h)$ from the CAS-only atomic points-to out to construct the single-writer ownership. Second, we update the full fraction $\mathsf{atExclTime}^{\gamma}(t_2)$ from the CAS ownership, together with the authoritative ghost state in the atomic points-to, to the latest write timestamp t to complete the single-writer ownership. □

10.2.3 *Proof Sketches for Atomic Operations*

The proofs for atomic operations are done in the base logic, after unfolding iRC11 Hoare triple and WP definitions (Definition 8.5 and Definition 8.4). We then rely on the base logic rules for atomic operations (§7.3) to proceed.

Proof sketch of AT-READ-SN. The basis of the proof is to apply BL-HOARE-READ-AT (Figure 7.5). In the pre-condition we have $\ell \sqsupseteq_{\mathtt{sn}} h_0 * @_{V_b}(\ell \overset{t_x}{\longmapsto}_{\theta} h)$, and we need to satisfy the pre-condition of BL-HOARE-READ-AT. From $\ell \sqsupseteq_{\mathtt{sn}} h_0$ we get $\mathsf{Local}_{\mathtt{A}}(\ell, h, \mathcal{V}.\mathsf{cur})$ (after unfolding the models of Hoare triples and WPs, in the base logic). From $@_{V_b}(\ell \overset{t_x}{\longmapsto}_{\theta} h)$ we get (1) $\mathsf{Hist}(\ell, h) * \mathsf{Read}^{\sqsupseteq \mathtt{rlx}}(\ell, \alpha_2)$, because they are objective, and (2) $\mathsf{Local}^{\sqsupseteq \mathtt{rlx}}_{\mathtt{R}}(\ell, \alpha_2, V_b)$. Therefore we can call BL-HOARE-READ-AT.

Afterwards, we prove the post-condition of AT-READ-SN from that of BL-HOARE-READ-AT. The most important new observation that we get is $\mathsf{Local}^{\sqsupseteq \mathtt{rlx}}_{\mathtt{R}}(\ell, \alpha_2 \cup \{r\}, V_b \sqcup \mathcal{V}'.\mathsf{cur})$, which fits perfectly in AT-READ-SN's requirement of $@_{V_b \sqcup V'}(\ell \overset{t_x}{\longmapsto}_{\theta} h)$, where it is the case that $V' = \mathcal{V}'.\mathsf{cur}$.

We are then left with the observations and facts about the read message. Note that due to $\ell \sqsupseteq_{\mathtt{sn}} h_0$, the current view $\mathcal{V}.\mathsf{cur}$ before the step has observed all events in h_0, so by the post-condition $\mathcal{V} \xrightarrow{\mathtt{R}:o,\ell,t,V^?,r} \mathcal{V}'$ from BL-HOARE-READ-AT, we know that the read timestamp t is not earlier than the writes in h_0, *i.e.*, $t \geq \mathtt{max}(\mathtt{dom}(h_0))$. Furthermore inspection of $\mathcal{V} \xrightarrow{\mathtt{R}:o,\ell,t,V^?,r} \mathcal{V}'$ is needed to prove the remaining observations, and while that is not trivial, it is rather routine so we elide it here. □

Proof sketch of AT-WRITE-SN. The proof is similar to that of AT-READ-SN, but relies on the rule BL-HOARE-WRITE-AT. The main difference is in the update of the local atomic writes observation, from $\mathsf{Local}^{\sqsupseteq \mathtt{rlx}}_{\mathtt{W}}(\ell, \alpha_w, V_b)$ to $\mathsf{Local}^{\sqsupseteq \mathtt{rlx}}_{\mathtt{W}}(\ell, \alpha_w \cup \{t\}, V_b \sqcup V')$. Naturally, a careful inspection of the condition $\mathcal{V} \xrightarrow{\mathtt{W}:o,\ell,t,\bot,V} \mathcal{V}'$ is needed to establish the remaining observations. □

Proof sketch of AT-CAS-SN-GEN. The proof is similar to those of AT-READ-SN and AT-WRITE-SN, but relies on the rule BL-HOARE-CAS. The 3 main tasks are:

- show that the premise $\forall v_0, t_0 \geq \mathtt{max}(\mathtt{dom}(h_0)).\, h(t_0) = (v_0, \_) \Rightarrow \vdash v_0 =^? v_r$ implies $\forall v_0 \in \mathsf{Readable}(h, \mathcal{V}).\, \vdash v_0 =^? v_r$; and
- show that AT-CAS-SN-GEN's $(P_{\mathrm{cmp}} \dot{\Rrightarrow} \Phi_{\mathrm{cmp}}(\ell_r, h))$ implies BL-HOARE-CAS's $(P_{\mathrm{cmp}} \mathrel{-\!\!*} \Phi_{\mathrm{cmp}}(\ell_r))$; and
- show the observations of AT-CAS-SN-GEN's post-condition from that of BL-HOARE-CAS.

The first two tasks are straightforward, mostly by unfolding definitions. Again, the last task requires careful inspection, but is routine and therefore elided here. □

CHAPTER SUMMARY. In this chapter, we presented the interface and the model of the atomic points-to assertion and its related local ownership and observations. The atomic points-to construction is designed to not only support the naturally desired concurrent atomic accesses, but also support strong reasoning principles in typical usage modes. It also supports switching between the different modes, as well as alternating between phases of non-atomic access and of atomic accesses. In the next chapters, we will see the flexibility of the atomic points-to assertion when combined with *invariants* in verifications.

11

Invariants in Relaxed Memory

In §6.3, we have reviewed Iris invariants as the key tool to share resources for concurrent accesses. In the Iris program logic (and also Iris-derived SC logics), invariants can be used to build concurrent protocols of data structures, by putting *all* shared ownership into a data structure's invariant. The invariant enforces a user-defined relation on the shared resources, so as to constrain how operations can access and change them. For example, in verifying a linked-list based concurrent queue, one can put the points-to ownership of the queue's head and tail pointers, as well as all the nodes of the queue, into a single invariant, and state the FIFO protocol on those points-to assertions, within the same invariant.

In RMC separation logics, we also want the facilities of invariants to support concurrent resource sharing. The situation is a bit different, because when moving resources locally owned by a thread—and thus are being interpreted by that thread's local views—into the "public domain" of invariants, we have to know the views used to interpret those resources, now that they are no longer tied to a thread. The SC-logic idea of putting all resources in a single invariant then appears intractable in RMC logics: those resources may be accessed separately and concurrently by multiple threads, so they may hold at separately different views. If we look at the concurrent queue example again, we see that an enqueueing thread would mostly work with the tail pointer, while a dequeuing one would separately work with the head pointer, so the ownership of the two pointers are likely to hold at different views. In other words, there is no single coherent history of updates to all locations shared in an invariant, and concurrent threads accessing the invariant cannot hope to agree on a consistent view of the *whole* invariant content.

As such, RSL, FSL, GPS, and their descendants[1] opted to restrict invariants to *single-location* invariants (or protocols) which restrict the evolution of a single, shared location. Intuitively, this is sound because in C11 all threads always share a consistent view on the single-location modification order mo_ℓ: writes to a single location is always totally ordered. Single-location invariants have proved useful in practice, but they become cumbersome to use when one works with a set of closely related locations, such as in the concurrent queue example. Roughly speaking, one would have a single-location invariant for each of the queue's head, tail, and nodes, and if one wants to enforce a property that spans multiple of those locations, one would have to invent mechanisms

[1] Vafeiadis and Narayan, "Relaxed separation logic: a program logic for C11 concurrency" [VN13]; Turon et al., "GPS: navigating weak memory with ghosts, protocols, and separation" [TVD14]; Tassarotti et al., "Verifying read-copy-update in a logic for weak memory" [TDV15]; Doko and Vafeiadis, "A Program Logic for C11 Memory Fences" [DV16]; Doko and Vafeiadis, "Tackling Real-Life Relaxed Concurrency with FSL++" [DV17]; Kaiser et al., "Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris" [Kai+17]; He et al., "GPS+: Reasoning About Fences and Relaxed Atomics" [He+18].

to tie those single-location invariants together.

In particular, the verifications done in GPS [TVD14] involve designing multiple extra ghost state to create permissions that can be owned by the protocol participants, and thus allow them to restrict interference by others. For example, the GPS verification for the linked-list based Michael-Scott concurrent queue[2] sets up the head's single-location invariant to always hold a unique permission, say $\diamond_0$ to access the first (0) element in the queue, and every node i holds also a unique permission $\diamond_{i+1}$ to access the next node after it. A dequeue requires a CAS to update the head pointer, and if that is successful, the dequeue caller acquires $\diamond_0$, which then can be used to access the first (0) element resources from the first element's single-location invariant, including the permission $\diamond_1$. The caller needs to put back $\diamond_1$ into the head's invariant, because that is the permission for the next dequeuer to access the element 1, which is now the next element to be dequeued. *This contrive setup with extra ghost state would not be needed—and in fact is not needed in SC logics—had we have all resources stored inside a single invariant*, because then we would have the relation between the nodes clearly stated in one place, instead of having the relation broken up in form of multiple permissions.

[2] Michael and Scott, "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms" [MS96].

This long discussion is to motivate *general invariants for multiple locations* in RMC, whose variants will be presented in this chapter. In Part III, we will show how these invariants are used to derive GPS single-location invariants.

The key challenge of general invariants is to identify the views that justify the different parts of the invariant content. One solution is simple: let the clients (of invariants) pick those views, explicitly using the view-at modality (Definition 8.13, §8.5), and then the invariant itself has no extra work to do. This gives rise to iRC11 *objective* invariants, a direct lifting of Iris invariants from iProp to vProp, which we present in §11.1.

However, we also would like to support *cancelable invariants*, those that allow reclaiming the invariant content once the invariant is no longer in use. Instead of the clients, cancelable invariants themselves take care of the one view that justifies all parts of the invariant content, and thus guarantee that *the cancelation of the invariant is synchronized with all accesses to the invariant*. We present iRC11 cancelable invariants in §11.2.

In §11.3, we provide the interface of another invariant form, called *non-atomic invariants*. They are needed for RustBelt Relaxed in the model of Rust's type system (Part III).

11.1 Objective Invariants

Definition 11.1 (vProp Objective Invariants)**.** The vProp objective invariants directly lift Iris iProp invariants as follows.

$$[\![\boxed{I}^{\mathcal{N}}]\!] ::= \lambda\_. \boxed{\forall V.\, [\![I]\!] V}^{\mathcal{N}}$$

That is, the invariant content $I : \mathsf{vProp}$ is put inside an Iris invariant at an arbitrary (universally quantified) view V. This resembles the model of the objective modality (Definition 8.10), hence the name.

$$\mathsf{objective}(\boxed{I}^{\mathcal{N}}) \qquad \textsc{OInv-alloc}\quad \triangleright \langle \mathsf{obj} \rangle\, I \vdash \Rrightarrow_{\mathcal{E}} \boxed{I}^{\mathcal{N}} \qquad \textsc{OInv-acc}\quad \frac{\mathcal{N} \subseteq \mathcal{E}}{\boxed{I}^{\mathcal{N}} \vdash {}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}} \left(\triangleright \langle \mathsf{obj} \rangle\, I * (\triangleright \langle \mathsf{obj} \rangle\, I \;{}^{\mathcal{E}\setminus\mathcal{N}}\!\!\Rrightarrow^{\mathcal{E}} \mathsf{True})\right)}$$
$$\mathsf{persistent}(\boxed{I}^{\mathcal{N}})$$

$$\textsc{OInv-alloc-obj}\quad \frac{\mathsf{objective}(I)}{\triangleright I \vdash \Rrightarrow_{\mathcal{E}} \boxed{I}^{\mathcal{N}}} \qquad \textsc{OInv-acc-obj}\quad \frac{\mathsf{objective}(I) \qquad \mathcal{N} \subseteq \mathcal{E}}{\boxed{I}^{\mathcal{N}} \vdash {}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}} \left(\triangleright I * (\triangleright I \;{}^{\mathcal{E}\setminus\mathcal{N}}\!\!\Rrightarrow^{\mathcal{E}} \mathsf{True})\right)}$$

FIGURE 11.1: iRC11 rules for objective invariants

Objective invariants satisfy the rules in Figure 11.1. That is, the rules are similar to those of Iris iProp invariants (Figure 6.4), except that they require the invariant content I to be objective, or be placed under an objective modality. More specifically, with OINV-ALLOC, we can allocate a vProp objective invariant if we have the invariant content I under the objective modality, and with OINV-ACC we can access the invariant content atomically, but we only get I also under the objective modality. The rules OINV-ALLOC-OBJ and OINV-ACC-OBJ are derived from OINV-ALLOC and OINV-ACC, respectively, using the rules OBJMOD-INTRO and OBJMOD-ELIM (§8.4).

Intuitively, if a client only stores objective resources, which include pure facts (ϕ) and ghost state ($\boxed{a}^{\gamma}$),[3] then objective invariants work the same as Iris traditional invariants. If the client, however, wishes to put points-to assertions into an objective invariants, then they should make those resources objective, by making their interpreting views explicit using the view-at modality (Definition 8.13). In particular, we can apply VA-INTRO (Figure 8.3) to turn an atomic points-to $\ell \overset{t}{\mapsto}{}^{\gamma}_{\theta}\ h$ can be turn into the form $@_V(\ell \overset{t}{\mapsto}{}^{\gamma}_{\theta}\ h)$ (knowing that $\sqsupseteq V$) which is objective (see Figure 8.3) and thus can be put inside objective invariants using OINV-ALLOC. Fortunately, we have established in sections §10.1.1 to §10.1.3 that the atomic access rules only require and return the atomic points-to ownership at some arbitrary views V_b and $V_b \sqcup V'$, respectively.

[3] see PURE-OBJ and GHOST-OBJ, §8.4

In other words, the atomic access rules using atomic points-to (§10.1) are compatible with objective invariants, and are sufficient to verify algorithms with concurrent protocols on shared atomic points-to assertions. We will see such examples in Chapter 12 and in Part IV.

We note that we can also use VA-INTRO to make *non-atomic* points-to assertions objective, and put them in invariants to transfer them to other threads. However, the non-atomic points-to can only be used without the view-at modality. Fortunately, the atomic access rules allow us to acquire the view-seen observations needed to remove the view-at modality from the non-atomic points-to, using VA-ELIM. For more details, please recall the discussion about resource transfer at the end of §10.1.2, which uses the rules AT-WRITE-SW-REL and AT-READ-SN-ACQ. Nevertheless, we will also see examples for resource transfer in Chapter 12.

11.2 Cancelable Invariants

If we put some resources in objective invariants, the resources are in the public domain forever. But there are situations where we want to reclaim the resources in the invariants, after which point we know that the invariants are no longer needed any more. For example, we would like to have a per-object invariant to govern the protocol of a data structure, but when the data structure object is deallocated, the invariant should become obsolete.

Iris supports a kind of invariants called *cancelable invariants* where an invariant can be canceled to reclaim the invariant. We would like to support this kind of invariants for vProp, but we need some extra work to make the returned resources useable: we need to track more carefully the view at which the resources are put inside the invariant, so that after the cancelation we can eliminate the view-at modality protecting those resources, and thus the client can own them locally.

11.2.1 *The Interface of* vProp *Cancelable Invariants*

We present the interface of vProp cancelable invariants in Figure 11.2.[4] Note that we did not present Iris iProp cancelable invariants, but they only differ from the vProp ones in the parts concerning views.[5]

The interface of cancelable invariants involves two kinds of assertions: (1) a persistent, objective assertion $\boxed{I}^{\gamma,\mathcal{N}}$ that an invariant with content I exists in the namespace $\mathcal{N}$ with an identifier γ; and (2) a fractional and timeless *invariant token* assertion $\heartsuit_q^\gamma$ (also identified by γ) that is needed to know that the invariant is not yet canceled.

[4] These are called *raw* cancelable invariants in the RustBelt Relaxed paper [Dan+20a].

[5] Please see Figure 14.1 for a few key rules of Iris traditional cancelable invariants. See also [Iri22, §10.1].

Invariant Allocation. With the invariant content I, we can allocate a cancelable invariant using CInv-alloc. Afterwards, we know that the invariant exists ($\boxed{I}^{\gamma,\mathcal{N}}$), and we own the full fraction $\heartsuit_1^\gamma$ of its invariant token. The invariant token is fractional, as shown in CInv-tok-frac-valid and CInv-tok-frac, so that we can split the full fraction into pieces and give them to multiple threads and they can access the invariant concurrently. CInv-tok-obj-split allows us to split a fraction of a token identified by γ into two parts, one of which is objective, which in turn can be easily put inside an invariant (which could possibly be the invariant with the same identifier γ as the token).

Invariant Access. A fraction of the invariant token is needed to open the invariant with the same identifier γ, as can be seen in CInv-acc. The opening is a mask-changing fancy update, after which we receive the invariant content I under a later. Once we are done, we must return I and close the invariant with a reverse mask-changing fancy update. Recall from §6.7 that mask-changing fancy updates are needed to prevent opening an invariant twice, and to limit the invariant accesses to a single, atomic step of computation. More concretely, recall that the rules WP-Inv and Hoare-Inv for opening traditional Iris invariants around an atomic step are derived from WP-atomic and Inv-acc. We apply the same method

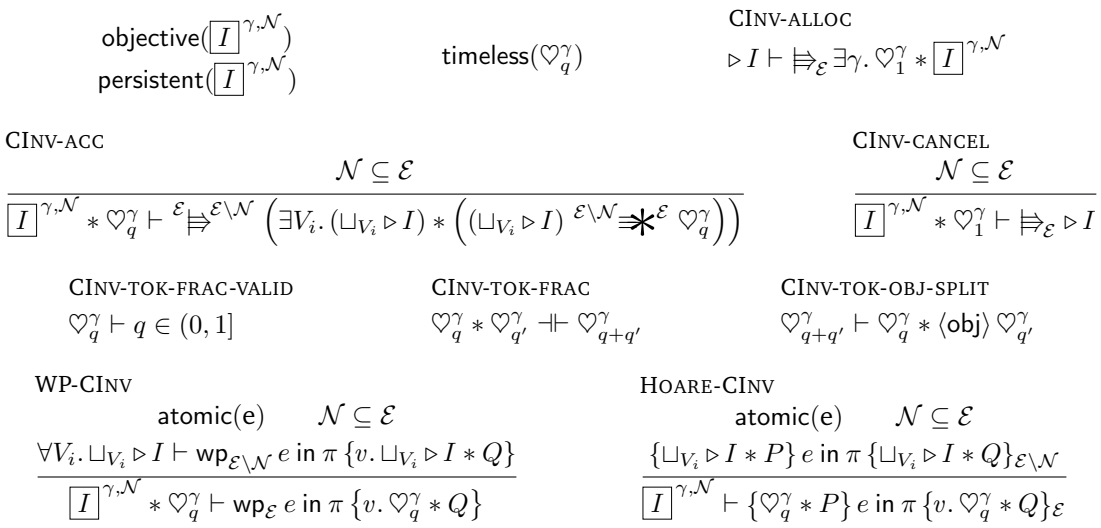


FIGURE 11.2: iRC11 rules for cancelable invariants

with WP-ATOMIC and CINV-ACC to derive the explicit rules WP-CINV and HOARE-CINV for opening vProp cancelable invariants.

The invariant content I we receive from CINV-ACC, however, unlike that from the rule INV-ACC for traditional Iris invariants, are protected under a *view-join* modality (Definition 8.14), *i.e.*, what we get from opening the invariant and what we have to return to close the invariant are both $\sqcup_{V_i} \triangleright I$. Recall the intuition of $\sqcup_{V_i} P$: it asserts the ownership of the resource P which hold at a view whose difference with the implicit interpreting view V is V_i. That is, P holds at $V_i \sqcup V$. This means that a client of the cancelable invariant only gains access to the invariant content I at an *arbitrary larger* view $V_i \sqcup V$ ($@_{V_i \sqcup V} \triangleright I$), where in practice V_i represents the view at which I is currently justified. During its access to I, the client can update the current view from V to some larger view V' ($V \sqsubseteq V'$), so long as it returns the invariant content I at the view $V_i \sqcup V'$ ($@_{V_i \sqcup V'} \triangleright I$), *i.e.*, it returns $\sqcup_{V_i} \triangleright I$.

The use of the view-join modality in CINV-ACC therefore enforces two requirements on the clients of cancelable invariants. The first requirement is that clients of cancelable invariants should be able to work with the content I justified at some arbitrary view $V_i \sqcup V$ (where V is the client's current view at the opening of the invariant). This requirement is the same as that for clients of objective invariants, and therefore can be mitigated in the same ways as discussed in the previous section (§11.1): objective resources do not care about views, and atomic access rules can work with the atomic points-to ownership at some arbitrary view, while views for other resources like non-atomic points-to need to be tracked more carefully and rely on the seen-view observations from the atomic operations. Again, we will see this point worked out more concretely in the example verifications in Chapter 12.

The second requirement is that the client cannot return I at a too big view: the view $V_i \sqcup V'$ must be sufficient to justify I (where V' is the client's current view at the closing of the invariant).[6] We note that this requirement is only enforced on non-objective resources in I.

[6] Note that if the client returns I at a view smaller than $V_i \sqcup V'$, then it can always return I at $V_i \sqcup V'$, thanks to view monotonicity of I.

Fortunately, the atomic access rules using atomic points-to (in §10.1.1 to §10.1.3) are also designed to be compatible with this requirement: if we provides an atomic points-to $@_{V_i \sqcup V} \ell \overset{t}{\mapsto}_\theta h$ to the pre-condition of one of the rules, we will receive some $@_{V_i \sqcup V \sqcup V'} \ell \overset{t}{\mapsto}_\theta h'$ in the post-condition, which is indeed $@_{V_i \sqcup V'} \ell \overset{t}{\mapsto}_\theta h$ because $V \sqsubseteq V'$. We will see this point more concretely also in Chapter 12.

Note that we can switch between the view-join and view-at modalities easily using VJ-VA-ACC. In fact, with VJ-VA-ACC and the rules in §10.1.1 to §10.1.3, we can derive atomic access rules that take an atomic points-to in the form $\sqcup_{V_i} \ell \overset{t}{\mapsto}_\theta h$ in the pre-condition, and return an updated atomic points-to in the form $\sqcup_{V_i} \ell \overset{t}{\mapsto}_\theta h'$ in the post-condition. We will see the derivations of those rules also in §12.2 (see Figure 12.5).

INVARIANT CANCELATION. The second requirement by the access rule CINV-ACC is what guarantees the soundness of the cancelation rule CINV-CANCEL. Cancelation needs to maintain the following safety guarantee.

Property 11.2 (Cancelation Safety)**.**

An invariant's cancellation must happen-after all accesses to it.
(CANCEL-SAFE)

Nevertheless, CINV-CANCEL simply says that with we can trade in the full fraction $\heartsuit_1^\gamma$ of the invariant token for the invariant $\boxed{I}^{\gamma,\mathcal{N}}$ to cancel it and get back the invariant content I *locally without any view-explicit modality* (albeit under a later, as usual). As such, *except* for the uses of the view-join modality in CINV-ACC, the core interface of iRC11 cancelable invariants (CINV-ALLOC, CINV-ACC, and CINV-CANCEL) is exactly the same as that of traditional Iris cancel invariants that are sound only for SC logics.[7] The reason why the cancelation rule maintains CANCEL-SAFE (*i.e.*, race-free and safe) for RMC, and why the relaxed memory effects can be localized in just the view-join modality used in CINV-ACC, is rather hard to explain intuitively, without looking into the model of invariant tokens. We therefore delay this explanation until §11.2.2.

[7] see Figure 14.1 and [Iri22, §10.1]

STRONGER ALLOCATION RULES. Figure 11.3 provides several stronger rules for cancelable allocation and access. CINV-ALLOC-OPEN strengthens CINV-ALLOC by not requiring the invariant content I upfront. Instead the client is first given a fresh identifier γ for the invariant token, and so the client can pick the invariant content I that may depend on γ. The client then receives the the invariant assertion $\boxed{I}^{\gamma,\mathcal{N}}$ but the invariant does not hold yet (hence the mask does not include $\mathcal{N}$). Once the client provides $\triangleright I$, the invariant is established and the client receives the full invariant token $\heartsuit_1^\gamma$.

CINV-ALLOC-FRAC strengthens CINV-ALLOC in a slightly different way. The client first receives some fraction q of the invariant token with a fresh identifier γ, the the client can pick and provide the invariant content I that may contain $\heartsuit_q^\gamma$ itself. After the invariant is established the client receives the remaining fraction $\heartsuit_{q'}^\gamma$, *i.e.*, $q + q' = 1$. Note that q and q' are picked by the client.

CINV-ALLOC-OPEN

$$\frac{\mathcal{N} \subseteq \mathcal{E}}{\vdash \Rrightarrow_{\mathcal{E}} \exists \gamma.\, \forall I.\, {}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}} \boxed{I}^{\gamma,\mathcal{N}} * \left(\triangleright I \;{}^{\mathcal{E}\setminus\mathcal{N}}\!\!\Rrightarrow^{\mathcal{E}} \heartsuit_1^\gamma\right)}$$

CINV-ALLOC-FRAC

$$\frac{q + q' = 1}{\vdash \Rrightarrow_{\mathcal{E}} \exists \gamma.\, \heartsuit_q^\gamma * \left(\forall I.\, \triangleright I \Rrightarrow_{\mathcal{E}} \heartsuit_{q'}^\gamma * \boxed{I}^{\gamma,\mathcal{N}}\right)}$$

CINV-ACC-GEN

$$\frac{\mathcal{N} \subseteq \mathcal{E}}{\boxed{I}^{\gamma,\mathcal{N}} * \heartsuit_q^\gamma \vdash {}^{\mathcal{E}}\Rrightarrow^{\mathcal{E}\setminus\mathcal{N}} \heartsuit_q^\gamma * \exists V_i.\, (\sqcup_{V_i} \triangleright I) * \wedge \begin{cases} \left(\heartsuit_q^\gamma * (\sqcup_{V_i} \triangleright I)\right) \;{}^{\mathcal{E}\setminus\mathcal{N}}\!\!\Rrightarrow^{\mathcal{E}} \heartsuit_q^\gamma \\ \forall V', P.\, \left(@_{V'} \heartsuit_q^\gamma * \left(@_{V'} \heartsuit_q^\gamma \Rrightarrow_{\mathcal{E}\setminus\mathcal{N}} (\sqcup_{V_i} \triangleright I) * P\right)\right) \;{}^{\mathcal{E}\setminus\mathcal{N}}\!\!\Rrightarrow^{\mathcal{E}} P \\ \heartsuit_1^\gamma \mathrel{-\!\!*} \left(\sqsupseteq V_i * {}^{\mathcal{E}\setminus\mathcal{N}}\!\!\Rrightarrow^{\mathcal{E}} \mathsf{True}\right) \end{cases}$$

FIGURE 11.3: Stronger iRC11 rules for cancelable invariants

A STRONGER ACCESS RULE. CINV-ACC-GEN (Figure 11.3) generalizes the access rule CINV-ACC in several non-trivial ways. First, one does not need to trade in the invariant token $\heartsuit_q^\gamma$ to access the invariant: one does need to provide it, but then receives it back immediately together with the invariant content $\sqcup_{V_i} \triangleright I$. Second, there are 3 options to close the invariant.

The first option is similar to the closing viewshift of CINV-ACC: to close the invariant, the client gives back the invariant content $\sqcup_{V_i} \triangleright I$. As the client did not trade in the invariant token, they need to provide the token at the closing, which they will get back immediately afterwards.[8]

The second closing option strengthens the first one further: to close the invariant, the client provides (1) the invariant token $\heartsuit_q^\gamma$ at some view V' of its choice, and (2) a wand viewshift $\left(@_{V'} \heartsuit_q^\gamma \Rrightarrow_{\mathcal{E}\setminus\mathcal{N}} (\sqcup_{V_i} \triangleright I) * P\right)$. Intuitively, the invariant needs (1) to update its internal tracking of the current invariant view. After that it feeds (1) into (2), a logical *continuation* picked by the client that, when receiving the same token, will produce the invariant content $\sqcup_{V_i} \triangleright I$ and some remaining resource P. The invariant then consumes $\sqcup_{V_i} \triangleright I$ to re-establish and close the invariant, and returns the remaining resource P to the client.

The last closing option embeds the cancelation rule CINV-CANCEL: instead of returning the invariant content I, the client can trade the full fraction $\heartsuit_1^\gamma$ of the invariant token for (1) a seen-view observation $\sqsupseteq V_i$ of the same view V_i that the client has received at the invariant opening, so that they can eliminate the view-join modality from I; and (2) a reverse mask-changing fancy update ${}^{\mathcal{E}\setminus\mathcal{N}}\!\!\Rrightarrow^{\mathcal{E}} \mathsf{True}$ without additional requirement to close the invariant. We note that CINV-CANCEL is derivable from CINV-ACC-GEN.

Finally, we note that the stronger rules (CINV-ALLOC-OPEN, CINV-ALLOC-FRAC, and CINV-ACC-GEN) *without* the view-join modality are all sound in traditional Iris SC logics. In those logics, the second closing option of CINV-ACC-GEN also coincides with the rule's first option.

[8] Another possible generalization is that the fraction q needed at closing does not need to be the same fraction used for opening the invariant.

11.2.2 *The Model of Cancelable Invariants*

To model iRC11 cancelable invariants, we need more ghost state to encode the access and cancelation protocol, which will be stored in the

ghost location γ—the invariant identifier.

Definition 11.3 (RA for iRC11 Cancelable Invariants)**.** We need the fractional view-lattice RA FRACVIEWR $= \text{AUTH}(\text{OPTION}(\text{FRAC} \times \text{LAT}(\mathit{View})))$. We define notations for two kinds of elements of the RA.

$$\mathsf{PartialV}_q(V_p) ::= \circ\,(\mathsf{Some}(q, V_p))$$
$$\mathsf{FullV}(V_f) ::= \bullet\,(\mathsf{Some}(1, V_f))$$

Definition 11.4 (Model of iRC11 Cancelable Invariants)**.** The model of invariant tokens and cancelable invariants is given directly in vProp, using the RA and objective invariants, as follows.

$$\heartsuit^\gamma_q ::= \exists V_{\mathtt{tok}}.\, \boxed{\mathsf{PartialV}_q(V_{\mathtt{tok}}) : \text{FRACVIEWR}}^\gamma * \sqsupseteq V_{\mathtt{tok}} \qquad \text{(CINV-MODEL-TOK)}$$

$$\boxed{I}^{\gamma,\mathcal{N}} ::= \boxed{\exists V_i.\, @_{V_i}\left(\boxed{\mathsf{PartialV}_1(V_{\mathtt{tok}})}^\gamma \vee \boxed{\mathsf{FullV}(V_i)}^\gamma * I\right)}^{\mathcal{N}} \qquad \text{(CINV-MODEL)}$$

INVARIANT TOKENS. First of all, invariant tokens $\heartsuit^\gamma_q$ are *view-dependent* assertions: even though owning a token $\heartsuit^\gamma_q$ means owning only the ghost element $\mathsf{PartialV}_q(V_{\mathtt{tok}})$, this ghost ownership is tied to the current implicit view V at which the assertion is interpreted through the *token view* $V_{\mathtt{tok}}$. In particular, the ghost element $\mathsf{PartialV}_q(V_{\mathtt{tok}})$ records both the fraction q, which represents how much of the invariant this token owns, and the token view $V_{\mathtt{tok}}$, which represents what this particular fractional token has *observed*, *i.e.*, what invariant accesses this fractional token has participated in. The model requires that V—the current implicit view at which the token is interpreted—has also at least observed what $\heartsuit^\gamma_q$ has observed: $\sqsupseteq V_{\mathtt{tok}}$ (the seen-view observation, see Definition 8.12).

INVARIANT ASSERTIONS. The model of invariant assertions $\boxed{I}^{\gamma,\mathcal{N}}$ simply encodes the two possible states of the invariant: "active" or "canceled". Thus it is an *objective invariant* (§11.1) of a disjunction (see CINV-MODEL). The right-hand side of the disjunction encodes the active state, where the content I is still available in the invariant at some *content view* V_i. In the active state the underlying invariant also owns the authoritative ghost element $\mathsf{FullV}(V_i)$ that records the view V_i in the ghost location γ. The left-hand side of the disjunction encodes the canceled state, which asserts ownership of the full fractional element $\mathsf{PartialV}_1(V_i)$. Recall that the invariant assertion $\boxed{I}^{\gamma,\mathcal{N}}$ itself is objective.[9] The underlying invariant's content is also objective: it is wrapped under a view-at modality of the content view V_i.[10] The relation between the content view V_i and the token views $V_{\mathtt{tok}}$'s is managed entirely by the ghost elements $\mathsf{FullV}(V_i)$ and $\mathsf{PartialV}_q(V_{\mathtt{tok}})$.

[9] see Figure 11.1

[10] see Figure 8.3

Concept 11.5 (Synchronized Ghost State)**.** In essence, by its model in CINV-MODEL-TOK, cancelable invariant tokens are just ghost state. However, unlike the vanilla ghost state ownership which is objective,[11] invariant tokens are not objective as they are tied to their owner's observations. We generally call these "synchronized ghost state". The RA

[11] see GHOST-OBJ, §8.4

FRACVIEWR has two interesting kinds of elements that help us implement the idea of "synchronicity": (1) the unique element $\mathsf{FullV}(V_f)$ that is used to record the *full view* V_f, and (2) fractional elements $\mathsf{PartialV}_q(V_p)$ that are used to associate some *partial view* V_p with some fraction q. FRACVIEWR is built to maintain the following property:

> *The join of all partial views (the V_p's from all $\mathsf{PartialV}_q(V_p)$'s) is always equal to the full view V_f in $\mathsf{FullV}(V_f)$.* (SYNC-GHOST)

This property guarantees that the partial view V_p of the full fractional element $\mathsf{PartialV}_1(V_p)$ is actually equal to the full view V_f of $\mathsf{FullV}(V_f)$: $V_p = V_f$. *The* SYNC-GHOST *property is what we require for view-dependent ghost state to be synchronized ghost state.* By synchronized ghost state we mean any ghost construction that is built on the notion of *fractional observations*. That is, the ghost state has fractional elements that track the subjective observations of the threads the elements are tied to, and, most importantly, the full fractional element is guaranteed to have tracked all observations.

In the case of cancelable invariants, the observations are the views around which threads access and update the invariant content I. Intuitively, we record the view V_i of the invariant content I as the full view in $\mathsf{FullV}(V_i)$ (see CINV-MODEL). The token view $V_{\mathtt{tok}}$ in the ghost element $\mathsf{PartialV}_q(V_{\mathtt{tok}})$ of some token $\heartsuit_q^\gamma$ tracks the changes to I made by each access that $\heartsuit_q^\gamma$ participated in. By SYNC-GHOST, the full token view $V_{\mathtt{tok\_full}}$ of the full token $\heartsuit_1^\gamma$ will thus be equal to the content view V_i. Consequently a thread owning $\heartsuit_1^\gamma$ must have observed all changes to the invariant content I, *i.e.*, it must have $\sqsupseteq V_i$. Effectively, CANCEL-SAFE is maintained and CINV-CANCEL can safely eliminate the view-at modality protecting I (using VA-ELIM, §8.5) and return $\triangleright I$ at the canceling thread's current view.

FORMAL PROPERTIES OF FRACVIEWR. To maintain SYNC-GHOST, the RA FRACVIEWR admits the rules in Figure 11.4. CINV-MODEL-SYNC says that any token view $V_{\mathtt{tok}}$ is included in the content view V_i, and the full token view $V_{\mathtt{tok\_full}}$ of $\heartsuit_1^\gamma$ is exactly V_i. CINV-MODEL-JOIN requires that the fractions consistently cannot sum up to more than 1, and also allows us to join together partial token views of the fractions when we are recollecting them. CINV-MODEL-UPDATE formalizes a restriction on how the ghost state can grow: we can update a token view $V_{\mathtt{tok}}$ by extending it with some V' *only if* we simultaneously update the content view V_i in the same way. This makes sure that every change in the full view V_i is accounted for by some token view $V_{\mathtt{tok}}$, and thus SYNC-GHOST is maintained.

Formally, CINV-MODEL-SYNC comes from validity of FRACVIEWR. If we own both $\mathsf{PartialV}_1(V_p)$ and $\mathsf{FullV}(V_f)$, by GHOST-OP and GHOST-VALID (§6.2), we have $\mathsf{valid}(\mathsf{FullV}(V_f) \cdot \mathsf{PartialV}_1(V_p)) = \mathsf{valid}(\bullet\,(\mathsf{Some}(1, V_f)) \cdot \circ\,(\mathsf{Some}(1, V_p)))$. By AUTH-BOTH-VALID (§6.9), we have that $\mathsf{Some}(1, V_p) \preccurlyeq \mathsf{Some}(1, V_f)$. By the definition of RA inclusion (RA-INCL, §6.2), it must be the case that $\mathsf{Some}(1, V_p) = \mathsf{Some}(1, V_f)$, *i.e.*, $V_p = V_f$.

CINV-MODEL-SYNC
$$\boxed{\mathsf{FullV}(V_i)}^{\gamma} * \boxed{\mathsf{PartialV}_q(V_{\mathtt{tok}})}^{\gamma} \vdash V_{\mathtt{tok}} \sqsubseteq V_i \land (q = 1 \Rightarrow V_{\mathtt{tok}} = V_i)$$

CINV-MODEL-UPDATE
$$\boxed{\mathsf{FullV}(V_i)}^{\gamma} * \boxed{\mathsf{PartialV}_q(V_{\mathtt{tok}})}^{\gamma} \Rrightarrow \boxed{\mathsf{FullV}(V_i \sqcup V')}^{\gamma} * \boxed{\mathsf{PartialV}_q(V_{\mathtt{tok}} \sqcup V')}^{\gamma}$$

CINV-MODEL-JOIN
$$\boxed{\mathsf{PartialV}_q(V)}^{\gamma} * \boxed{\mathsf{PartialV}_{q'}(V')}^{\gamma} \vdash \boxed{\mathsf{PartialV}_{q+q'}(V \sqcup V')}^{\gamma} * q + q' \in (0, 1]$$

FIGURE 11.4: Properties of the RA FRACVIEWR for cancelable invariants

PROOF SKETCHES. To understand how the model works, we briefly present the proofs of CINV-CANCEL and CINV-ACC.

Proof sketch of CINV-CANCEL. We prove the rule in vProp. After unfolding the model (Definition 11.4), we have the following goal.

Context:	Goal:
$\boxed{\mathsf{PartialV}_1(V_{\mathtt{tok}})}^{\gamma} * \sqsupseteq V_{\mathtt{tok}}$ $\boxed{\exists V_i.\, @_{V_i}\left(\boxed{\mathsf{PartialV}_1(V_{\mathtt{tok}})}^{\gamma} \lor \boxed{\mathsf{FullV}(V_i)}^{\gamma} * I\right)}^{\mathcal{N}}$	$\Rrightarrow_{\mathcal{E}} \triangleright I$

We then open the underlying objective invariant using OINV-ACC-OBJ and find a content view V_i and the two possibilities for the invariant state. If the invariant were in the cancelled state (the left disjunct), we would have *two* full fractional $\mathsf{PartialV}_1(\_)$ and CINV-MODEL-JOIN would give us contradiction from $1 + 1 \leq 1$. Thus the underlying invariant must be in the active state (the right disjunct).

By owning the full fraction, with CINV-MODEL-SYNC we know that $V_{\mathtt{tok}} = V_i$, so by owning $\sqsupseteq V_{\mathtt{tok}}$, the thread must have observed all changes to the invariant content: $\sqsupseteq V_i$. With that, we now can take the content $@_{V_i} V_i$ out of the invariant and eliminate the view-at modality with VA-ELIM, and return $\triangleright I$ for the user. To finish the proof, we put $\boxed{\mathsf{PartialV}_1(V_{\mathtt{tok}})}^{\gamma}$ in to switch the underlying invariant to the cancelled state and close it. □

Proof sketch of CINV-ACC. We also prove the rule in vProp. As in cancelation, we unfold the model, then open the underlying invariant with OINV-ACC-OBJ and deduce that it must be in the active state. Our goal then looks as follows.

Context:	Goal:
$(\triangleright \exists V_i.\ \ldots)\ {}^{\mathcal{E}\setminus\mathcal{N}}\!\!\Rrightarrow\!\!\!\!\ast^{\mathcal{E}}\ \mathsf{True}$ $\boxed{\mathsf{PartialV}_q(V_{\mathtt{tok}})}^{\gamma} * \sqsupseteq V_{\mathtt{tok}}$ $\boxed{\mathsf{FullV}(V_i)}^{\gamma} * @_{V_i} \triangleright I$	$\exists V_i.\, \sqcup_{V_i} \triangleright I * \left((\sqcup_{V_i} \triangleright I)\ {}^{\mathcal{E}\setminus\mathcal{N}}\!\!\Rrightarrow\!\!\!\!\ast^{\mathcal{E}}\ \heartsuit_q^{\gamma}\right)$

We instantiate the existential quantification with V_i, and then use VA-TO-VJ (Figure 8.3) to upgrade the invariant content I from the view-at modality to the view-join modality, so that we can discharge the left-hand side of the goal. We are then left to prove the closing wand viewshift.

After introduction, our goal looks as follows.

Context:	Goal:
$(\triangleright \exists V_i.\ \ldots)\ {}^{\mathcal{E}\setminus\mathcal{N}}\!\!\Rrightarrow\!\!\ast^{\mathcal{E}}\ \mathsf{True}$	
$\boxed{\mathsf{PartialV}_q(V_{\mathtt{tok}})}^{\gamma} * \sqsupseteq V_{\mathtt{tok}}$	
$\boxed{\mathsf{FullV}(V_i)}^{\gamma} * \sqcup_{V_i} \triangleright I$	${}^{\mathcal{E}\setminus\mathcal{N}}\!\!\Rrightarrow^{\mathcal{E}} \exists V'_{\mathtt{tok}}.\ \boxed{\mathsf{PartialV}_q(V'_{\mathtt{tok}})}^{\gamma} * \sqsupseteq V'_{\mathtt{tok}}$

We now use VJ-ELIM-VA (also Figure 8.3) to turn $\sqcup_{V_i} \triangleright I$ and $\sqsupseteq V_{\mathtt{tok}}$ into $@_{V_i \sqcup V'} \triangleright I$ for some $V' \sqsupseteq V_{\mathtt{tok}}$ that we know $\sqsupseteq V'$. We then use CInv-Model-Update to update $\boxed{\mathsf{FullV}(V_i)}^{\gamma} * \boxed{\mathsf{PartialV}_q(V_{\mathtt{tok}})}^{\gamma}$ to $\boxed{\mathsf{FullV}(V_i \sqcup V')}^{\gamma} * \boxed{\mathsf{PartialV}_q(V_{\mathtt{tok}} \sqcup V')}^{\gamma}$.

We then use the closing viewshift $(\triangleright \exists V_i.\ \ldots)\ {}^{\mathcal{E}\setminus\mathcal{N}}\!\!\Rrightarrow\!\!\ast^{\mathcal{E}}\ \mathsf{True}$ with the view $V_i \sqcup V'$ and the resources $\boxed{\mathsf{FullV}(V_i \sqcup V')}^{\gamma}$ and $@_{V_i \sqcup V'} \triangleright I$ to re-establish and close the invariant. We are left with the goal for the invariant token.

Context:	Goal:
$\boxed{\mathsf{PartialV}_q(V_{\mathtt{tok}} \sqcup V')}^{\gamma} * \sqsupseteq V'$	$\exists V'_{\mathtt{tok}}.\ \boxed{\mathsf{PartialV}_q(V'_{\mathtt{tok}})}^{\gamma} * \sqsupseteq V'_{\mathtt{tok}}$

From $V' \sqsupseteq V_{\mathtt{tok}}$, we know that $V_{\mathtt{tok}} \sqcup V' = V'$, so this is easily done. □

11.3 Non-Atomic Invariants

Iris additionally provides a derived form of invariants where the access can be non-atomic, *i.e.*, it can span multiple steps of execution. The catch is that each such access can only be done by one thread at a time. This form of invariants, called non-atomic invariants, is needed to model unique reference types in RustBelt.[12] We will thus also need non-atomic invariants for our RustBelt Relaxed work (Part III).

Fortunately, Iris non-atomic invariants can be proven sound in relaxed memory without any change in the interface! Naturally, the model of iRC11 non-atomic invariants still needs to handle relaxed memory effects, but it manages to encapsulate them within the interface. This is sound, intuitively because non-atomic invariants are meant to be thread-local—i.e. being accessed by only the current thread—so the thread is always synchronized with the invariant content. The model carefully tracks the view of the invariant content, employing an RA similar to that of cancelable invariants. The model and the proofs of iRC11 non-atomic invariants were constructed by Jacques-Henri Jourdan, as thus are not considered part of this dissertation and will not be presented here. The exact definitions are available in the Coq development of iRC11.[13]

Nevertheless, we present the interface of iRC11 non-atomic invariants in Figure 11.5, which, again, is exactly the same as that of Iris-SC.[14] Like cancelable invariants, non-atomic invariants also have two kinds of assertions: (1) a persistent, objective assertion $\mathsf{NaInv}^{p.\mathcal{N}}(I)$ that a non-atomic invariant with content I exists in the namespace $\mathcal{N}$ with of invariant pool p; and (2) a timeless *invariant token* assertion $[\mathsf{Na} : p.\mathcal{E}]$ that is needed to access the invariants in the set $\mathcal{E}$ under the pool p.

Invariant pools allow us to have separate pools of invariants with their own namespaces and tokens. Intuitively, we can think of pools as

[12] Jung et al., "RustBelt: Securing the Foundations of the Rust Programming Language" [Jun+18a].

[13] https://gitlab.mpi-sws.org/iris/gpfsl/-/blob/master/theories/logic/na_invariants.v

[14] Iris Team (The), *The Iris 3.6 Technical Appendix* [Iri22], §10.3.

$$\mathsf{objective}(\mathsf{NaInv}^{p.\mathcal{N}}(I)) \quad \mathsf{persistent}(\mathsf{NaInv}^{p.\mathcal{N}}(I)) \qquad \mathsf{timeless}([\mathrm{Na} : p.\mathcal{E}]) \qquad \textsc{NAInv-new-pool}\ \vdash \dot{\Rightarrow} \exists p.\, [\mathrm{Na} : p.\top]$$

$$\textsc{NAInv-tok-split}\ [\mathrm{Na} : p.\mathcal{E}_1 \uplus \mathcal{E}_2] \dashv\vdash [\mathrm{Na} : p.\mathcal{E}_1] * [\mathrm{Na} : p.\mathcal{E}_2] \qquad \textsc{NAInv-alloc}\ \triangleright I \vdash \Rightarrow_{\mathcal{E}} \mathsf{NaInv}^{p.\mathcal{N}}(I)$$

$$\textsc{NAInv-acc}\ \frac{\mathcal{N} \subseteq \mathcal{E} \qquad \mathcal{N} \subseteq \mathcal{E}'}{\mathsf{NaInv}^{p.\mathcal{N}}(I) * [\mathrm{Na} : p.\mathcal{E}'] \vdash \Rightarrow_{\mathcal{E}} \triangleright I * [\mathrm{Na} : p.\mathcal{E}' \setminus \mathcal{N}] * (\triangleright I * [\mathrm{Na} : p.\mathcal{E}' \setminus \mathcal{N}] \mathrel{-\!\!*}_{\mathcal{E}} [\mathrm{Na} : p.\mathcal{E}'])}$$

FIGURE 11.5: The interface of non-atomic invariants

threads, and non-atomic invariants as *thread-local* invariants, where each thread has its own, local pool of invariants. Every thread also has its own invariant tokens $[\mathrm{Na} : p.\mathcal{E}]$, which can be "threaded through" its execution to access its own invariant pools, without having to worry about other threads' interference. Thus the accesses are really sequential, and can span multiple (non-atomic) instructions.

A fresh invariant pool p can be allocated with NAINV-NEW-POOL, where we obtain the invariant token $[\mathrm{Na} : p.\top]$ for p with the full set of invariant names ($p.\top$). The token can be split and joined using NAINV-TOK-SPLIT, which supports accessing disjoint sets of invariants. With some invariant content I, we can allocate a non-atomic invariant in some namespace $\mathcal{N}$ of the pool p using NAINV-ALLOC.

Finally, the most important rule is the access rule NAINV-ACC: with a token $[\mathrm{Na} : p.\mathcal{E}']$ for some mask $p.\mathcal{E}'$ that includes $p.\mathcal{N}$, we can open the invariant $\mathsf{NaInv}^{p.\mathcal{N}}(I)$ to access the content $\triangleright I$. During the access, we lose the ownership of the invariant names in $p.\mathcal{N}$, so we only have the remaining token $[\mathrm{Na} : p.\mathcal{E}' \setminus \mathcal{N}]$ that allows us to open more invariants *except* those that have already been opened. Once we return the invariant content I, we can regain the original token $[\mathrm{Na} : p.\mathcal{E}']$. We note that the access is non-atomic because we are not force to have *mask-changing* fancy updates/viewshifts: through out the access the usual *atomic* invariants in $\mathcal{E}$ still hold. Recall that the masks in fancy updates ($\Rightarrow_{\mathcal{E}}$) are used to maintain non-reentrancy for *atomic* invariants, while the masks in non-atomic invariant tokens ($[\mathrm{Na} : p.\mathcal{E}']$) are used to maintain non-reentrancy for *non-atomic* invariants.

CHAPTER SUMMARY. In this chapter we introduce 3 forms of invariants: (1) objective invariants that are useful to share general, multi-location resources for concurrent accesses; and (2) cancelable invariants that support reclaiming concurrently shared resources; and (3) non-atomic thread-local invariants that allow non-atomic accesses to invariant contents. We will see example uses of (1) and (2) in Chapter 12, and more of them in the rest of this dissertation. We will see the application of (3) in Part III, where it is used for the lifetime logic's non-atomic persistent borrows (§16.2.3).

12

Example Verifications with iRC11

In this chapter we demonstrate various features of iRC11 that have been presented so far, using several simple example verifications concerning the message-passing idiom. In §12.1 we sketch some verifications of the message-passing examples that we have seen in Chapter 2 (Figure 2.1) and demonstrate the uses of non-atomic (Chapter 9) and atomic points-to (Chapter 10), objective invariants (§11.1), view-explicit modalities (§8.5) and fence modalities (§8.3). In §12.2 we verify the message-passing but with resource reclamation (deallocation), demonstrating cancelable invariants (§11.2) and the switching from atomic back to non-atomic points-to (§10.5).[1] In §12.3, we verify a slightly more complex *spawn-and-join* library, which allows spawning a computation as a child thread and then waiting for its completion to receive the computation result. The transfer of the result is implemented using message-passing.[2] Finally, in §12.4, we verify a release-acquire implementation of the linked-list based Treiber stack[3] against a simple "bag" specification, which demonstrates the atomic points-to CAS rule with pointer comparison.[4] We will revisit the Treiber stack with a stronger specification in Compass (Part IV).

[1] Coq proofs of these MP examples are in https://gitlab.mpi-sws.org/iris/gpfsl/-/blob/master/gpfsl-examples/mp/proof_gen_inv.v

[2] Coq proof in https://gitlab.mpi-sws.org/iris/lambda-rust/-/blob/masters/weak_mem/theories/lang/spawn.v

[3] Treiber, *Systems Programming: Coping with Parallelism* [Tre86].

[4] Coq proof in https://gitlab.mpi-sws.org/iris/gpfsl/-/blob/master/gpfsl-examples/stack/proof_treiber_at.v

12.1 Release-Acquire Message-Passing

In Figure 12.1(b), we provide two λ_{Rust} + ORC11 implementations of the message passing example, together with the desired specification. Figure 12.1(a) presents the implementation using release-acquire accesses, and corresponds to Example 2.1(c). The **mp** program runs on the main thread π. It allocates a block of size 2 with the base location ℓ, and non-atomically initializes both locations to 0. It then forks a child thread ρ which will *non-atomically* write the message 42 to $\ell + 1$, and signal the message by *atomically* writing 1 to ℓ. The main thread π waits for the signal by a loop of acquire reads of ℓ. **repeat** (e) is implemented as a recursive function, which keeps executing e until it returns **true**. Once the loop ends, π should be able to get the message 42 from ρ, safely using a non-atomic read on $\ell + 1$. The program **mp_acq_fence** (Figure 12.1(b)) optimizes **mp** by using only relaxed reads in the loop, and then an acquire fence after the loop finishes.

Both programs should satisfy the simple specification in MP-SPEC: the returned value is the message 42. We note that both programs should satisfy the stronger specification MP-SPEC-STRONG, where the ownership

mp ::=

$\pi 1$: **let** $\ell :=$ **alloc**(2) **in**

$\pi 2$: $\ell :=_{\mathsf{na}} 0; (\ell + 1) :=_{\mathsf{na}} 0;$

$\pi 3$: **fork**$\{\ \rho 1$: $(\ell + 1) :=_{\mathsf{na}} 42; \rho 2$: $\ell :=_{\mathsf{rel}} 1;\ \}$

$\pi 4$: **repeat** $({}^{*\mathsf{acq}}\ell \mathrel{!}= 0);$

$\pi 5$: ${}^{*\mathsf{na}}(\ell + 1)$ // 42

((a)) MP with release-acquire accesses.

mp_acq_fence ::=

$\pi 1$: **let** $\ell :=$ **alloc**(2) **in**

$\pi 2$: $\ell :=_{\mathsf{na}} 0; (\ell + 1) :=_{\mathsf{na}} 0;$

$\pi 3$: **fork**$\{\ \rho 1$: $(\ell + 1) :=_{\mathsf{na}} 42; \rho 2$: $\ell :=_{\mathsf{rel}} 1;\ \}$

$\pi 4$: **repeat** $({}^{*\mathsf{rlx}}\ell \mathrel{!}= 0);$

$\pi 5$: $\mathsf{fence}_{\mathsf{acq}};$

$\pi 6$: ${}^{*\mathsf{na}}(\ell + 1)$ // 42

((b)) MP with an acquire fence.

repeat $(e) ::= (\mathbf{rec}\, f([]) := \mathbf{let}\, v := e\, \mathbf{in}$
$\mathbf{if}\, v == \mathbf{false}\, \mathbf{then}\, f([])\, \mathbf{else}\, v)([])$

MP-SPEC
$\{\mathsf{True}\}\ \mathbf{mp}\ \mathsf{in}\ \pi\ \{v.\, v = 42\}_\top$

MP-SPEC-STRONG
$\{\mathsf{True}\}\ \mathbf{mp}\ \mathsf{in}\ \pi\ \{v.\, v = 42 * \exists \ell.\, \ell \mapsto [1, 42] * \dagger^2 \ell\}_\top$

FIGURE 12.1: Message-Passing with Loops

of the allocated block is returned, to prevent memory leaks. Note that the notation $\ell \mapsto [1, 42]$ stands for $\ell \mapsto 1 * \ell + 1 \mapsto 42$. We will look at a similar proof to that of MP-SPEC-STRONG in §12.2.

A high-level proof sketch of **mp** . We start in line $\pi 1$ using NA-ALLOC (§9.1), from which we get the block ownership $\dagger^2 \ell$ and two non-atomic points-to assertions for ℓ and $\ell+1$. The two points-to are sufficient for initialization in line $\pi 2$, using NA-WRITE. We then use NA-AT-SW (§10.1) to turn the non-atomic points-to $\ell \mapsto 0$ of ℓ to an atomic one with a single-writer permission $(\ell \mapsto^{\gamma_\ell}_{\mathsf{sw}} \_ * \ell \sqsupseteq^{\gamma_\ell}_{\mathsf{sw}} \_)$. We then use OINV-ALLOC-OBJ (§11.1) to allocate an objective invariant that contains that atomic points-to $\ell \mapsto^{\gamma_\ell}_{\mathsf{sw}} \_$. In line $\pi 3$ when forking ρ, we give the non-atomic points-to $\ell + 1 \mapsto 0$ of $\ell + 1$ and the single-writer ownership $\ell \sqsupseteq^{\gamma_\ell}_{\mathsf{sw}} \_$ of ℓ to ρ, as well as the fact that the invariant has been established.

In thread ρ: in line $\rho 1$, with the non-atomic points-to of $\ell + 1$, we can write the message 42 using NA-WRITE, and then get $\ell + 1 \mapsto 42$. In line $\rho 2$, we can open the invariant with OINV-ACC to access the atomic points-to $\ell \mapsto^{\gamma_\ell}_{\mathsf{sw}} \_$ of ℓ so that we can perform the release write of 1 to it, using AT-WRITE-SW-REL (§10.1.2) and the single-writer ownership $\ell \sqsupseteq^{\gamma_\ell}_{\mathsf{sw}} \_$ of ℓ. When closing the invariant we return to the invariant not only ℓ's atomic points-to but also $\ell + 1 \mapsto 42$ at the view of the release write, so that π can regain it.

Back to thread π: in line $\pi 4$, in the **repeat** loop we can open the invariant and access ℓ's atomic points-to and keep reading ℓ, using AT-READ-SN-ACQ (§10.1.1). One we read that ℓ is non-zero and the loop ends, we know that $\ell + 1 \mapsto 42$ must be inside the invariant, and thread π has observed the view of $\ell + 1 \mapsto 42$. We need a unique token $\diamond$ to say that π is the only one who can acquire $\ell + 1 \mapsto 42$. This must be prepared in the beginning before allocating the invariant, and the token $\diamond$ is given to

thread π. At this point, we trade $\diamond$ for $\ell + 1 \mapsto 42$ in the invariant, and use π's view observation of the release write to acquire $\ell + 1 \mapsto 42$ locally. With that, in line $\pi 5$, we use NA-READ to read and return 42. □

To write out the proof more formally, we need to define the exclusive ghost token $\diamond$ and the objective invariant for **mp**.

Definition 12.1 (Exclusive Tokens)**.** We use the exclusive RA of unit $\mathrm{Ex}(1)$ to define exclusive tokens.

$$\diamond^\gamma ::= \boxed{\mathrm{ex}() : \mathrm{Ex}(1)}^\gamma$$

Exclusive tokens satisfy the following rules.

$$\mathsf{timeless}(\diamond^\gamma) \quad \mathsf{objective}(\diamond^\gamma) \qquad \text{EXCL-TOK-ALLOC}\ \vdash \dot{\Rightarrow} \exists \gamma.\, \diamond^\gamma \qquad \text{EXCL-TOK-EXCL}\ \diamond^\gamma * \diamond^\gamma \vdash \mathsf{False}$$

Definition 12.2 (Invariant for **mp**)**.** The invariant of **mp** needs to be objective, and contains the atomic points-to ownership of ℓ in single-writer mode, since only the thread ρ is writing.

$$\begin{aligned}
&\mathsf{mpl}(\ell, \gamma_l, \gamma) : \mathsf{vProp} ::= \\
&\quad \exists h, b, t_0, V_0, V_\ell.\, @_{V_\ell}(\ell \mapsto^{\gamma_\ell}_{\mathtt{sw}} h) * \mathbf{let}\, h_0 := [t_0 \leftarrow (0, V_0)]\ \mathbf{in} \\
&\quad \mathbf{if}\, b = \mathbf{false}\, \mathbf{then}\, h = h_0\, \mathbf{else} \\
&\quad \exists t_1 > t_0, V_1.\, h = [t_0 \leftarrow (0, V_0)][t_1 \leftarrow (1, V_1)] * (\diamond^\gamma \vee @_{V_1}(\ell + 1 \mapsto 42))
\end{aligned}$$

More concretely, the invariant mpl owns $@_{V_\ell}(\ell \mapsto^{\gamma_\ell}_{\mathtt{sw}} h)$ for some atomic period identifier γ_ℓ and some history h, at some view V_ℓ. The history h of ℓ can be in two states, dictated by the existential quantified boolean b. If b is **false**, then ℓ is still in its initialized state, *i.e.*, it has a singleton history h_0 with the write message $(t_0, 0, V_0)$. Once b is **true**, ℓ is in its "signaled" state, where its history h has one extra write message $(t_1, 1, V_1)$. When ℓ is in the signaled state, mpl either owns $\diamond^\gamma$, or owns $\ell + 1 \mapsto 42$ at the view V_1 (of the signaling write). If mpl owns $\ell + 1 \mapsto 42$, it means that ρ has released the non-atomic of $\ell + 1$ but π has not acquired it yet. Once π has acquired $\ell + 1$'s non-atomic points-to, mpl will own $\diamond^\gamma$.

mpl is clearly objective, because it only contains pure facts, ghost state, and points-to assertions that are under the view-at modality.

Proof sketch of **mp** . We present the detailed proof sketch of **mp** using *Hoare proof outlines*, in Figure 12.2. Note that the post-condition of the proof is almost satisfying the stronger specification MP-SPEC-STRONG: it is only missing the points-to $\ell \mapsto 1$. □

Proof sketch of **mp_acq_fence** . The proof of **mp** uses the same invariant mpl, and follows the proof of **mp** closely. The main difference is that in thread ρ's read of ℓ, which is a relaxed read instead of an acquire read, we use the rule AT-READ-SN. Consequently, in the case where ρ reads 1 from ℓ, we will acquire $\nabla_\pi \sqsupseteq V' * @_{V_1} \ell + 1 \mapsto 42$, where $V' \sqsupseteq V_1$. Then, with the acquire fence, we apply HOARE-ACQ-FENCE (§8.3) to get $\sqsupseteq V'$.

$\{\mathsf{True}\}$
$\pi 1$: $\mathbf{let}\,\ell := \mathbf{alloc}(2)\,\mathbf{in}\ \{\ell \mapsto \textcircled{\scriptsize ✓} * \ell+1 \mapsto \textcircled{\scriptsize ✓} * \dagger^2\ell\}$ // NA-ALLOC
$\pi 2$: $\ell :=_{\mathsf{na}} 0; (\ell+1) :=_{\mathsf{na}} 0; \{\ell \mapsto 0 * \ell+1 \mapsto 0 * \dagger^2\ell\}$ // NA-WRITE
$\{\ell+1 \mapsto 0 * \dagger^2\ell * \exists\gamma, \gamma_\ell, t_0, V_0.\, \diamond^\gamma * \sqsupseteq V_0 * \ell \mapsto^{\gamma_\ell}_{\mathtt{sw}} [t_0 \leftarrow (0, V_0)] * \ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sw}} [t_0 \leftarrow (0, V_0)]\}$
Context: $\sqsupseteq V_0$ // EXCL-TOK-ALLOC and NA-AT-SW
$\{\ell+1 \mapsto 0 * \dagger^2\ell * \diamond^\gamma * \ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sw}} [t_0 \leftarrow (0, V_0)] * \exists V_\ell.\, \sqsupseteq V_\ell * @_{V_\ell} \ell \mapsto^{\gamma_\ell}_{\mathtt{sw}} [t_0 \leftarrow (0, V_0)]\}$ // VA-INTRO
$\{\ell+1 \mapsto 0 * \dagger^2\ell * \diamond^\gamma * \ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sw}} [t_0 \leftarrow (0, V_0)] * \ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sn}} [t_0 \leftarrow (0, V_0)] * @_{V_\ell} \ell \mapsto^{\gamma_\ell}_{\mathtt{sw}} [t_0 \leftarrow (0, V_0)]\}$
Context: $\ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sn}} [t_0 \leftarrow (0, V_0)]$ // AT-SW-SY and AT-SY-SN
$\{\ell+1 \mapsto 0 * \dagger^2\ell * \diamond^\gamma * \ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sw}} [t_0 \leftarrow (0, V_0)] * \mathsf{mpl}(\ell, \gamma_\ell, \gamma)\}$
$\left\{\ell+1 \mapsto 0 * \dagger^2\ell * \diamond^\gamma * \ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sw}} [t_0 \leftarrow (0, V_0)] * \boxed{\mathsf{mpl}(\ell, \gamma_\ell, \gamma)}^{\mathcal{N}}\right\}$ // OINV-ALLOC-OBJ
Context: $\boxed{\mathsf{mpl}(\ell, \gamma_\ell, \gamma)}^{\mathcal{N}}$
$\pi 3$: $\mathbf{fork}\{\ldots\}$ // HOARE-FORK

Thread ρ:
$\{\ell+1 \mapsto 0 * \ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sw}} [t_0 \leftarrow (0, V_0)]\}_\top$
$\rho 1$: $(\ell+1) :=_{\mathsf{na}} 42; \{\ell+1 \mapsto 42 * \ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sw}} [t_0 \leftarrow (0, V_0)]\}_\top$ // NA-WRITE

Accessing mpl:
$\{\ell+1 \mapsto 42 * \ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sw}} [t_0 \leftarrow (0, V_0)] * \triangleright\, \mathsf{mpl}(\ell, \gamma_\ell, \gamma)\}_{\top\setminus\mathcal{N}}$ // OINV-ACC-OBJ
$\left\{\ell+1 \mapsto 42 * \ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sw}} [t_0 \leftarrow (0, V_0)] * @_{V'_\ell}(\ell \mapsto^{\gamma_\ell}_{\mathtt{sw}} [t_0 \leftarrow (0, V_0)])\right\}_{\top\setminus\mathcal{N}}$
// By unfolding mpl and using a stronger version of AT-SW-AGREE
$\rho 2$: $\ell :=_{\mathtt{rel}} 1;$
$\left\{\begin{array}{l}\exists t_1, V_1 \sqsupset V_0.\, \sqsupseteq V_1 * @_{V_1}(\ell+1 \mapsto 42) * \\ @_{V_1} \ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sw}} [t_0 \leftarrow (0, V_0)][t_1 \leftarrow (1, V_1)] * @_{V'_\ell \sqcup V_1}(\ell \mapsto^{\gamma_\ell}_{\mathtt{sw}} [t_0 \leftarrow (0, V_0)][t_1 \leftarrow (1, V_1)])\end{array}\right\}_{\top\setminus\mathcal{N}}$
// By applying AT-WRITE-SW-REL with $\sqsupseteq V_0$
$\{\sqsupseteq V_1 * @_{V_1} \ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sw}} \_ * \mathsf{mpl}(\ell, \gamma_\ell, \gamma)\}_{\top\setminus\mathcal{N}}$ // By picking the "signaled" state for mpl

$\{\ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sw}} \_\}_\top$ // VA-ELIM
$\{\mathsf{True}\}_\top$
$\{\dagger^2\ell * \diamond^\gamma\}$
$\pi 4$: $\mathbf{repeat}\,(^{*\mathsf{acq}}\ell\ \mathtt{!=}\ 0);$

Loop body:
$\{\diamond^\gamma\}_\top$

Accessing mpl:
$\{\diamond^\gamma * \triangleright\, \mathsf{mpl}(\ell, \gamma_\ell, \gamma)\}_{\top\setminus\mathcal{N}}$ // OINV-ACC-OBJ
$\left\{\diamond^\gamma * \sqsupseteq V_0 * \ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sn}} [t_0 \leftarrow (0, V_0)] * @_{V'_\ell}(\ell \mapsto^{\gamma_\ell}_{\mathtt{sw}} h) * \triangleright\, \mathbf{if}\ldots\mathbf{then}\ldots\mathbf{else}\ldots\right\}_{\top\setminus\mathcal{N}}$
$^{*\mathsf{acq}}\ell$
$\left\{\begin{array}{l}v.\, \exists h' \subseteq h, t, V, V' \sqsupseteq V_0 \sqcup V.\, h'(t) = (v, V) * \sqsupseteq V' * @_{V'_\ell \sqcup V'}(\ell \mapsto^{\gamma_\ell}_{\mathtt{sw}} h) \\ \quad * \diamond^\gamma * \mathbf{if}\, b = \mathbf{false}\, \mathbf{then}\ldots\mathbf{else}\ (\diamond^\gamma \vee @_{V_1}(\ell+1 \mapsto 42))\end{array}\right\}_{\top\setminus\mathcal{N}}$ // AT-READ-SN-ACQ
We have $v \neq 0 \Rightarrow b = \mathbf{true} \wedge V_1 = V$. If $v = 0$, we return the invariant content unchanged.
If $v \neq 0 \wedge b = \mathbf{true}$, by EXCL-TOK-EXCL we have $@_{V_1}(\ell+1 \mapsto 42)$. With $V' \sqsupseteq V_0 \sqcup V$ and $\sqsupseteq V'$, by VS-MONO we have $\sqsupseteq V_1$. With VA-ELIM, we get $\ell+1 \mapsto 42$. We put back $\diamond^\gamma$ to re-establish mpl.
$\{((v = 0 * \diamond^\gamma) \vee (v \neq 0 * \ell+1 \mapsto 42)) * \mathsf{mpl}(\ell, \gamma_\ell, \gamma)\}_{\top\setminus\mathcal{N}}$

$\{(v = 0 * \diamond^\gamma) \vee (v \neq 0 * \ell+1 \mapsto 42)\}_\top$ // "loop invariant", we use LÖB induction before the loop
$\{\dagger^2\ell * \ell+1 \mapsto 42\}$ // $\dagger^2\ell$ was framed
$\pi 5$: $^{*\mathsf{na}}(\ell+1)\ \{v.\, v = 42 * \dagger^2\ell * \ell+1 \mapsto 42\}$ // NA-READ
$\{v.\, v = 42\}$

FIGURE 12.2: Hoare proof outlines for **mp**

Then we use VS-MONO to get $\sqsupseteq V_1$, which allows us to use VA-ELIM and get $\ell + 1 \mapsto 42$.

We give the Hoare proof outlines for the part that changes below.

$\{\dagger^2 \ell * \diamond^\gamma\}$
$\pi 4$: **repeat** $(*^{\mathbf{acq}}\ell \mathrel{!}= 0)$;
Loop body:
  $\{\diamond^\gamma\}_\top$
  $*^{\mathbf{rlx}}\ell$ // OINV-ACC-OBJ and AT-READ-SN
  $\{(v = 0 * \diamond^\gamma) \vee (v \neq 0 * \exists V' \sqsupseteq V_1. \nabla_\pi \sqsupseteq V' * @_{V_1}(\ell + 1 \mapsto 42))\}_\top$
$\{\dagger^2 \ell * \nabla_\pi \sqsupseteq V' * @_{V_1}(\ell + 1 \mapsto 42)\}$
$\pi 5$: $\mathbf{fence}_{\mathbf{acq}}$; $\{\dagger^2 \ell * \sqsupseteq V' * @_{V_1}(\ell + 1 \mapsto 42)\}$ // HOARE-ACQ-FENCE
$\{\dagger^2 \ell * \ell + 1 \mapsto 42\}$ // VS-MONO then VA-ELIM
$\pi 6$: $*^{\mathbf{na}}(\ell + 1)$ $\{v. v = 42 * \dagger^2 \ell * \ell + 1 \mapsto 42\}$ // NA-READ
$\{v. v = 42\}$

□

COMPARISON WITH IGPS PROOFS. The iGPS paper[5] also presents similar proofs of the Message-Passing example, once in its base logic and once in its surface logic. The proof presented in Figure 12.2 is slightly less complicated than the iGPS base-level proof where views are fully explicit, but is still more complicated than the iGPS surface-level proof, where views are fully hidden. The proof in Figure 12.2 only hides views around sequential (non-atomic) steps, but works with explicit views around atomic access steps, because the proof employs objective invariants and atomic points-to assertions. The iGPS surface-level proof for **mp** employs GPS single-location protocols in place of objective invariants and atomic points-to, and so views are hidden completely around atomic accesses. That proof is thus simpler than our proof here, because the invariant mpl is indeed a single-location invariant that governs the atomic accesses of ℓ.

[5] Kaiser et al., "Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris" [Kai+17], §3.2.2, §4.1.

This demonstrates that working with explicit views using atomic points-to in situations where we only have a single atomic location (or multiple atomic locations that are unrelated) are counterproductive, compared to single-location protocols. In §17 (Part III), we will show how to derive the higher-level abstraction of GPS single-location protocols from our atomic points-to and general invariants. Atomic points-to and general invariants, however, significantly simplify the process of stating a relation that spans multiple atomic locations, as we will see in Compass (Part IV). Furthermore, as we will see, working with explicit views is not just a trade-off (for multi-location invariants), but becomes a necessity to achieve the stronger specifications of Compass.

12.2 Release-Acquire Message-Passing with Reclamation

We now look at the verification of the **mp_reclaim**, whose implementation and specification are given in Figure 12.3. The **mp_reclaim** program extends **mp** simply by cleaning up the memory needed to perform the message passing, in line $\pi 6$ after reading the message and before returning it. The main difference of the verification is now to show that the

mp_reclaim ::=

$\pi 1$: **let** $\ell :=$ **alloc**(2) **in**

$\pi 2$: $\ell :=_{\mathsf{na}} 0; (\ell + 1) :=_{\mathsf{na}} 0;$

$\pi 3$: **fork**$\{$ $\rho 1$: $(\ell + 1) :=_{\mathsf{na}} 42;$ $\rho 2$: $\ell :=_{\mathsf{rel}} 1;$ $\}$

$\pi 4$: **repeat** $(*^{\mathsf{acq}}\ell \mathrel{!}= 0);$

$\pi 5$: **let** $v := *^{\mathsf{na}}(\ell + 1)$ **in**

$\pi 6$: **free**$(\ell, 2);$

$\pi 7$: v // 42

MP-RECLAIM-SPEC

$\{\mathsf{True}\}$

mp_reclaim in π

$\{v.\, v = 42\}_{\top}$

FIGURE 12.3: Message-Passing with Reclamation

call of **free** in line $\pi 6$ is safe. We note that verifying **mp_reclaim** against MP-RECLAIM-SPEC is the same as verifying **mp** against MP-SPEC-STRONG.

Recall that by the end the proof for **mp** (Figure 12.2), we are missing only the non-atomic points-to $\ell \mapsto 1$ of the location ℓ, which has been put in the invariant for shared atomic accesses, in the form of an atomic points-to. To reclaim the atomic points-to of ℓ from the invariant, we need to be able to cancel the invariant once thread π receives the message after line $\pi 4$. Interestingly, at the point the invariant does not hold anymore, so we do not need the exclusive token $\diamond^{\gamma}$ and the disjunction as in mpl (see Definition 12.2). On the other hand, we now need to deal with the invariant token $\heartsuit^{\gamma}_{q}$ of cancelable invariants (§11.2).

Definition 12.3 (Invariant for **mp_reclaim**)**.** The invariant of **mp_reclaim** does not need to be objective, so it contains the atomic points-to ownership of ℓ in single-writer mode, without being under a view-at modality.

$\mathsf{mpcl}(\ell, \gamma_l, \gamma) : \mathsf{vProp} ::=$

$\exists h, b, t_0, V_0.\, \ell \mapsto^{\gamma_\ell}_{\mathtt{sw}} h * \mathbf{let}\, h_0 := [t_0 \leftarrow (0, V_0)]\ \mathbf{in}$

$\mathbf{if}\, b = \mathbf{false}\, \mathbf{then}\, h = h_0\, \mathbf{else}$

$\exists t_1 > t_0, V_1.\, h = [t_0 \leftarrow (0, V_0)][t_1 \leftarrow (1, V_1)] * @_{V_1}(\heartsuit^{\gamma}_{1/2} * \ell + 1 \mapsto 42)$

The invariant also has two states ("initialized" and "signaled") like the invariant of **mp**. However, in the signaled state, instead of having a disjunction between the exclusive token $\diamond^{\gamma}$ and the non-atomic points-to of $\ell + 1$, the invariant just holds the points-to together with half of the cancelable invariant token $\heartsuit^{\gamma}_{1/2}$, at the message view V_1.

Intuitively, at the beginning the invariant token $\heartsuit^{\gamma}_{1/2}$ is split into two halves, and each participant (threads π and ρ) will keep one half to access the invariant. Once thread ρ is done, it sends back its half $\heartsuit^{\gamma}_{1/2}$ together with $\ell + 1 \mapsto 42$ (by putting them in mpcl so that thread π can acquire both, reconstruct the full token $\heartsuit^{\gamma}_{1}$, and cancel the invariant to get back $\ell \mapsto^{\gamma_\ell}_{\mathtt{sw}} h$.

Proof sketch for **mp_reclaim** . The Hoare proof outlines for **mp_reclaim** are given in Figure 12.4. We note several important points.

- Between line $\pi 2$ and $\pi 3$, we allocate a cancelable invariant for mpcl. But instead of using CINV-ALLOC, we use the stronger rule

$\{\mathsf{True}\}$

$\pi 1$: $\texttt{let}\,\ell := \texttt{alloc}(2)\,\texttt{in}$

$\pi 2$: $\ell :=_{\texttt{na}} 0; (\ell + 1) :=_{\texttt{na}} 0;\ \{\ell \mapsto 0 * \ell + 1 \mapsto 0 * \dagger^2 \ell\}$ // NA-ALLOC and NA-WRITE

$\{\ell + 1 \mapsto 0 * \dagger^2 \ell * \exists \gamma_\ell, t_0, V_0.\ \sqsupseteq V_0 * \ell \mapsto^{\gamma_\ell}_{\texttt{sw}} [t_0 \leftarrow (0, V_0)] * \ell \sqsupseteq^{\gamma_\ell}_{\texttt{sw}} [t_0 \leftarrow (0, V_0)]\}$ // NA-AT-SW

Context: $\sqsupseteq V_0 * \ell \sqsupseteq^{\gamma_\ell}_{\texttt{sn}} [t_0 \leftarrow (0, V_0)]$ // AT-SW-SY and AT-SY-SN

$\left\{\ell + 1 \mapsto 0 * \dagger^2 \ell * \ell \sqsupseteq^{\gamma_\ell}_{\texttt{sw}} [t_0 \leftarrow (0, V_0)] * \ell \mapsto^{\gamma_\ell}_{\texttt{sw}} [t_0 \leftarrow (0, V_0)] * \exists \gamma. \forall I.\ {}^{\top}\!\Rrightarrow^{\top\setminus\mathcal{N}} \ldots\right\}$ // CINV-ALLOC-OPEN

$\left\{\ell + 1 \mapsto 0 * \dagger^2 \ell * \ell \sqsupseteq^{\gamma_\ell}_{\texttt{sw}} [t_0 \leftarrow (0, V_0)] * \heartsuit^\gamma_1 * \boxed{\mathsf{mpcl}(\ell, \gamma_\ell, \gamma)}^{\gamma, \mathcal{N}}\right\}$

Context: $\boxed{\mathsf{mpcl}(\ell, \gamma_\ell, \gamma)}^{\gamma, \mathcal{N}}$

$\pi 3$: $\texttt{fork}\,\{\ldots\}$ // HOARE-FORK

Thread ρ

$\left\{\ell + 1 \mapsto 0 * \ell \sqsupseteq^{\gamma_\ell}_{\texttt{sw}} [t_0 \leftarrow (0, V_0)] * \heartsuit^\gamma_{1/2}\right\}_\top$

$\rho 1$: $(\ell + 1) :=_{\texttt{na}} 42;\ \left\{\ell + 1 \mapsto 42 * \ell \sqsupseteq^{\gamma_\ell}_{\texttt{sw}} [t_0 \leftarrow (0, V_0)] * \heartsuit^\gamma_{1/2}\right\}_\top$ // NA-WRITE

Accessing mpcl

$\left\{\begin{array}{l}\ell + 1 \mapsto 42 * \ell \sqsupseteq^{\gamma_\ell}_{\texttt{sw}} [t_0 \leftarrow (0, V_0)] * \heartsuit^\gamma_{1/2} * \exists V_i. \sqcup_{V_i} (\triangleright \mathsf{mpcl}(\ell, \gamma_\ell, \gamma)) \\ * \ldots \text{(* closing viewshifts *)}\end{array}\right\}_{\top\setminus\mathcal{N}}$ // CINV-ACC-GEN

$\left\{\ell + 1 \mapsto 42 * \ell \sqsupseteq^{\gamma_\ell}_{\texttt{sw}} [t_0 \leftarrow (0, V_0)] * \heartsuit^\gamma_{1/2} * \sqcup_{V_i} (\ell \mapsto^{\gamma_\ell}_{\texttt{sw}} [t_0 \leftarrow (0, V_0)]) * \ldots\right\}_{\top\setminus\mathcal{N}}$

$\rho 2$: $\ell :=_{\texttt{rel}} 1;$

$\left\{\begin{array}{l}\exists t_1, V_1 \sqsupset V_0.\ \sqsupseteq V_1 * @_{V_1} (\heartsuit^\gamma_{1/2} * \ell + 1 \mapsto 42) \\ * @_{V_1} \ell \sqsupseteq^{\gamma_\ell}_{\texttt{sw}} [t_0 \leftarrow (0, V_0)][t_1 \leftarrow (1, V_1)] * \sqcup_{V_i} (\ell \mapsto^{\gamma_\ell}_{\texttt{sw}} [t_0 \leftarrow (0, V_0)][t_1 \leftarrow (1, V_1)]) * \ldots\end{array}\right\}_{\top\setminus\mathcal{N}}$

// By applying AT-WRITE-SW-REL-VJ with $\sqsupseteq V_0$

We pick the "signaled" state for mpcl. We use the second closing viewshift option to close and release $@_{V_1} \heartsuit^\gamma_{1/2}$ to the invariant at the same time, in addition to $@_{V_1} (\ell + 1 \mapsto 42)$.

$\{\ell \sqsupseteq^{\gamma_\ell}_{\texttt{sw}} \_\}_\top$ // VA-ELIM

$\{\mathsf{True}\}_\top$

$\left\{\dagger^2 \ell * \heartsuit^\gamma_{1/2}\right\}$

$\pi 4$: $\texttt{repeat}\,({*}^{\texttt{acq}}\ell\ {!}{=}\ 0);$

Loop body

$\left\{\heartsuit^\gamma_{1/2}\right\}_\top$

Accessing mpcl

$\left\{\heartsuit^\gamma_{1/2} * \exists V_i. \sqcup_{V_i} \triangleright \mathsf{mpl}(\ell, \gamma_\ell, \gamma) * \ldots \text{(* closing viewshifts *)}\right\}_{\top\setminus\mathcal{N}}$ // CINV-ACC-GEN

$\left\{\heartsuit^\gamma_{1/2} * \sqsupseteq V_0 * \ell \sqsupseteq^{\gamma_\ell}_{\texttt{sn}} [t_0 \leftarrow (0, V_0)] * \sqcup_{V_i} (\ell \mapsto^{\gamma_\ell}_{\texttt{sw}} h) * \triangleright(\texttt{if} \ldots \texttt{then} \ldots \texttt{else} \ldots) * \ldots\right\}_{\top\setminus\mathcal{N}}$

${*}^{\texttt{acq}}\ell \left\{\begin{array}{l}v. \exists h' \subseteq h, t, V, V' \sqsupseteq V_0 \sqcup V. h'(t) = (v, V) * \sqsupseteq V' * \sqcup_{V_i} (\ell \mapsto^{\gamma_\ell}_{\texttt{sw}} h) \\ \quad * \heartsuit^\gamma_{1/2} * (\texttt{if}\, b = \texttt{false}\,\texttt{then} \ldots \texttt{else}\, @_{V_1} (\heartsuit^\gamma_{1/2} * \ell + 1 \mapsto 42)) * \ldots\end{array}\right\}_{\top\setminus\mathcal{N}}$

// AT-READ-SN-ACQ-VJ

We have $(v \neq 0 \Rightarrow b = \texttt{true} \wedge V_1 = V)$. If $v = 0$, we return the invariant content unchanged. If $b = \texttt{true}$, we have $@_{V_1} (\heartsuit^\gamma_{1/2} * \ell + 1 \mapsto 42)$. With VS-MONO and VA-ELIM, we get $\heartsuit^\gamma_{1/2} * \ell + 1 \mapsto 42$. Combining with π's own $\heartsuit^\gamma_{1/2}$ we get $\heartsuit^\gamma_1$.

$\left\{\begin{array}{l}(v = 0 * \heartsuit^\gamma_{1/2} * \sqcup_{V_i} \mathsf{mpcl}(\ell, \gamma_\ell, \gamma)) \vee (v \neq 0 * \heartsuit^\gamma_1 * \ell + 1 \mapsto 42 * \sqcup_{V_i} (\ell \mapsto^{\gamma_\ell}_{\texttt{sw}} h)) \\ * \ldots \text{(* closing viewshift *)}\end{array}\right\}_{\top\setminus\mathcal{N}}$

If $v = 0$, we use the first closing viewshift option to close the invariant. If $v \neq 0$, we use the *third* closing viewshift option to cancel the invariant. By the cancelation (CINV-ACC-GEN), we get $\sqsupseteq V_i$, which can be used with VJ-ELIM to get $\ell \mapsto^{\gamma_\ell}_{\texttt{sw}} h$. We then use AT-NA to get $\ell \mapsto 1$.

$\left\{(v = 0 * {}^{\top\setminus\mathcal{N}}\!\Rrightarrow^{\top} \heartsuit^\gamma_{1/2}) \vee (v \neq 0 * \ell \mapsto [1, 42] * {}^{\top\setminus\mathcal{N}}\!\Rrightarrow^{\top} \mathsf{True})\right\}_{\top\setminus\mathcal{N}}$

$\left\{(v = 0 * \heartsuit^\gamma_{1/2}) \vee (v \neq 0 * \ell \mapsto [1, 42])\right\}_\top$ // "loop invariant", we use LÖB induction before the loop

$\{\dagger^2 \ell * \ell \mapsto [1, 42]\}$ // $\dagger^2 \ell$ was framed

$\pi 5$: $\texttt{let}\,v := {*}^{\texttt{na}}(\ell + 1)\,\texttt{in}\ \{v = 42 * \dagger^2 \ell * \ell \mapsto [1, 42]\}$ // NA-READ

$\pi 6$: $\texttt{free}(\ell, 2);\ \{v = 42\}$ // NA-UNSYNC and NA-DEALLOC

$\pi 7$: $v\ \{v. v = 42\}$

FIGURE 12.4: Hoare proof outlines for **mp_reclaim**

CINV-ALLOC-OPEN (Figure 11.3), because our invariant mpcl depends on the invariant identifier γ itself. CINV-ALLOC-OPEN allows us to get the identifier γ before picking the invariant content which can depend on γ. In our case, that is $\mathsf{mpcl}(\ell, \gamma_\ell, \gamma)$.

- After the allocation of $\boxed{\mathsf{mpcl}(\ell, \gamma_\ell, \gamma)}^{\gamma, \mathcal{N}}$, we also get the full invariant token $\heartsuit^\gamma_1$, which we split into two halves of $\heartsuit^\gamma_{1/2}$, using CINV-TOK-FRAC. We then give one half to the thread ρ, using HOARE-FORK. The thread π retains the other half.

- In the proof of thread ρ, around the atomic access in line $\rho 2$, we also do not use the simple rule CINV-ACC, but instead use CINV-ACC-GEN to access the invariant. The latter gives us several options when closing the invariant, of which we will use the second option (see Figure 11.3). The second option allows us to use the half token $\heartsuit^\gamma_{1/2}$ (which we have used to open the invariant) to close the invariant and simultaneously release it into the invariant content mpcl. CINV-ACC-GEN indeed allows us to keep $\heartsuit^\gamma_{1/2}$ around after opening (and does not consume $\heartsuit^\gamma_{1/2}$ like CINV-ACC), so that when performing the release write of 1 to ℓ, we also release $\heartsuit^\gamma_{1/2}$ together with $\ell{+}1 \mapsto 42$. That is, after the write, we have $@_{V_1}(\heartsuit^\gamma_{1/2} * \ell + 1 \mapsto 42)$ where V_1 is the message view of the write.

  The second closing option looks as follows.

$$\forall V', P.\ \left(@_{V'}\heartsuit^\gamma_{1/2} * \left(@_{V'}\heartsuit^\gamma_{1/2} \Rrightarrow\!\!\!\!\!\!\ast^{\mathcal{E}\setminus\mathcal{N}}\ (\sqcup_{V_i} \triangleright \mathsf{mpcl}(\ell, \gamma_\ell, \gamma)) * P\right)\right) \overset{\mathcal{E}\setminus\mathcal{N}}{\Rrightarrow\!\!\!\!\!\!\ast}{}^{\mathcal{E}}\ P$$

  With $@_{V_1}(\heartsuit^\gamma_{1/2} * \ell + 1 \mapsto 42)$ and the atomic points-to

$$\sqcup_{V_i}(\ell \mapsto^{\gamma_\ell}_{\mathtt{sw}} [t_0 \leftarrow (0, V_0)][t_1 \leftarrow (1, V_1)])$$

  after the write, we instantiate the second closing option with $V' := V_1$ and $P := \mathsf{True}$. We give up $@_{V_1}\heartsuit^\gamma_{1/2}$ for the left-hand side of the separating conjunction before the wand viewshift ($\overset{\mathcal{E}\setminus\mathcal{N}}{\Rrightarrow\!\!\!\!\!\!\ast}{}^{\mathcal{E}}$). For the right-hand side, it is easy to show that

$$@_{V_1}(\ell + 1 \mapsto 42) * \sqcup_{V_i}(\ell \mapsto^{\gamma_\ell}_{\mathtt{sw}} [t_0 \leftarrow (0, V_0)][t_1 \leftarrow (1, V_1)]) \vdash @_{V_1}\heartsuit^\gamma_{1/2} \Rrightarrow\!\!\!\!\!\!\ast^{\mathcal{E}\setminus\mathcal{N}} \sqcup_{V_i} \triangleright \mathsf{mpcl}(\ell, \gamma_\ell, \gamma)$$

  by picking the signaled state ($b = \mathtt{true}$) for $\mathsf{mpcl}(\ell, \gamma_\ell, \gamma)$. We note that $@_{V_1}(\heartsuit^\gamma_{1/2} * \ell + 1 \mapsto 42) \dashv\vdash \sqcup_{V_i} @_{V_1}(\heartsuit^\gamma_{1/2} * \ell + 1 \mapsto 42)$, due to VJ-VA (Figure 8.3).

  After the instantiation, we get ${}^{\mathcal{E}\setminus\mathcal{N}}\!\Rrightarrow^{\mathcal{E}}$ True that we use to close the invariant and complete the access.

  Note that AT-WRITE-SW-REL allows us the option to also release the single-writer ownership to the invariant, because we have $@_{V_1}\ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sw}} \_$. We do not need this feature here, but it can be useful elsewhere (see §17).

- The same situation applies for thread π's atomic access in line $\pi 4$: we need to use CINV-ACC-GEN to open the invariant, and in case the

AT-WRITE-SW-REL-VJ

$$\{\sqsupseteq V_0 * \ell \sqsupseteq_{\mathtt{sw}} h * \sqcup_{V_b}(\ell \overset{t_x}{\longmapsto}_{\mathtt{sw}} h) * P\}\, \ell :=_{\mathtt{rel}} v \text{ in } \pi \left\{ \begin{array}{l} \text{✓}.\, \exists t, V \sqsupset V_0.\, \mathtt{max}(\mathtt{dom}(h)) < t * \sqsupseteq V * @_V P * \\ \quad @_V(\ell \sqsupseteq_{\mathtt{sw}} h[t \leftarrow (v, V)]) * \sqcup_{V_b}(\ell \overset{t}{\mapsto}_{\mathtt{sw}} h[t \leftarrow (v, V)]) \end{array} \right\}_{\mathcal{E}}$$

AT-READ-SN-ACQ-VJ

$$\{\sqsupseteq V_0 * \ell \sqsupseteq_{\mathtt{sn}} h_0 * \sqcup_{V_b}(\ell \overset{t_x}{\mapsto}_{\theta} h)\}\, {}^{*\mathtt{acq}}\ell \text{ in } \pi \left\{ \begin{array}{l} v.\, \exists h', t, V, V' \sqsupseteq V_0 \sqcup V.\, h_0 \subseteq h' \subseteq h * \\ \quad h'(t) = (v, V) * t \geq \mathtt{max}(\mathtt{dom}(h_0)) * \\ \quad \sqsupseteq V' * @_{V'}(\ell \sqsupseteq_{\mathtt{sn}} h') * \sqcup_{V_b}(\ell \overset{t_x}{\mapsto}_{\theta} h) \end{array} \right\}_{\mathcal{E}}$$

FIGURE 12.5: Derived iRC11 atomic access rules with the view-join modality

invariant is in the signaled state ($b = \mathtt{true}$), we need to use VJ-VA to get $@_{V_1}(\heartsuit^{\gamma}_{1/2} * \ell + 1 \mapsto 42)$. If the invariant is in the signaled state, we acquire $@_{V_1}(\heartsuit^{\gamma}_{1/2} * \ell + 1 \mapsto 42)$ from the invariant, which we can use VA-ELIM to get $\heartsuit^{\gamma}_{1/2} * \ell + 1 \mapsto 42$, because we also have $\sqsupseteq V_1$ thanks to the acquire read. Note that we also have π's half token $\heartsuit^{\gamma}_{1/2}$ locally, so together we have the full token $\heartsuit^{\gamma}_{1}$ (using CINV-TOK-FRAC). We then use the third closing option from CINV-ACC-GEN to cancel the invariant, from which we receive $\sqsupseteq V_i * {}^{\top \setminus \mathcal{N}}\!\!\Rightarrow^{\top} \mathsf{True}$. We use the fancy update to conclude the access. We combine $\sqsupseteq V_i$ with $\sqcup_{V_i}(\ell \mapsto^{\gamma_\ell}_{\mathtt{sw}} h)$ to get $\ell \mapsto^{\gamma_\ell}_{\mathtt{sw}} h$, using VJ-ELIM. We then use AT-NA to turn ℓ's atomic points-to to the non-atomic points-to $\ell \mapsto 1$, knowing that 1 is the latest write in h.

- Last but not least, we note that the atomic access rules AT-WRITE-SW-REL and AT-READ-SN-ACQ are not directly applicable to this proof, because the rules require an atomic points-to under a view-at modality, *i.e.*, $@_{V_b}\ell \mapsto_\theta h$, while we get a points-to under a view-join modality from the cancelable invariant access rule, *i.e.*, $\sqcup_{V_b}\ell \mapsto_\theta h$. Fortunately, in general, rules with the view-join modality can be derived from those with the view-at modality. We show the derived versions AT-WRITE-SW-REL-VJ and AT-READ-SN-ACQ-VJ in Figure 12.5. We demonstrate the derivation of AT-WRITE-SW-REL-VJ below.

□

Proof sketch of AT-WRITE-SW-REL-VJ. With $\sqsupseteq V_0$ and $\sqcup_{V_b}(\ell \overset{t_x}{\mapsto}_{\mathtt{sw}} h)$, we use VJ-ELIM-VA (Figure 8.3) to get

$$\sqsupseteq V' * @_{(V' \sqcup V_b)}(\ell \overset{t_x}{\mapsto}_{\mathtt{sw}} h)$$

for some $V' \sqsupseteq V_0$. We apply AT-WRITE-SW-REL with $@_{(V' \sqcup V_b)}(\ell \overset{t_x}{\mapsto}_{\mathtt{sw}} h)$. In the post-condition we get back the atomic points-to in the form

$$@_{V' \sqcup V_b \sqcup V}(\ell \overset{t}{\mapsto}_{\mathtt{sw}} h[t \leftarrow (v, V)])$$

for some $V \sqsupset V_0$ and $\sqsupseteq V$. We can rewrite it using VA-VJ to the form

$$@_{V' \sqcup V}(\sqcup_{V_b}(\ell \overset{t}{\mapsto}_{\mathtt{sw}} h[t \leftarrow (v, V)]))$$

We now use VA-ELIM to get back $\sqcup_{V_b}(\ell \overset{t}{\mapsto}_{\mathtt{sw}} h[t \leftarrow (v, V)])$. Note that we have $\sqsupseteq(V' \sqcup V)$ from $\sqsupseteq V'$ and $\sqsupseteq V$ using VS-JOIN. □

spawn ::=
$\lambda[f].$
1: **let** $\ell :=$ **alloc**(2) **in**
2: $\ell :=_{\mathsf{na}} 0$; // init to 0
3: **fork** $\{ f([\ell]) \}$; // spawn f
4: ℓ

finish ::=
$\lambda[\ell, v].$
1: $(\ell + 1) :=_{\mathsf{na}} v$; // write result
2: $\ell :=_{\mathsf{rel}} 1$ // signal done

join ::=
$\lambda[\ell].$
1: **repeat** $(^{*\mathsf{acq}}\ell \mathrel{!}= 0)$;
2: **let** $v := {}^{*\mathsf{na}}(\ell + 1)$ **in**
3: **free**$(\ell, 2)$; // clean up
4: v // return result

SPAWN-SPEC
$\left\{\left(\forall \ell, \rho.\, \mathsf{FinishHandle}^{\mathcal{N}}(\ell, \Phi) \twoheadrightarrow \mathsf{wp}\, f([\ell])\, \mathsf{in}\, \rho\, \{\_.\, \mathsf{True}\}[\top]\right)\right\}$ **spawn**$([f])$ in π $\{\ell.\, \exists \gamma.\, \mathsf{JoinHandle}^{\mathcal{N}}(\ell, \Phi)\}_{\top}$

FINISH-SPEC
$\{\mathsf{FinishHandle}^{\mathcal{N}}(\ell, \Phi) * \Phi(v)\}$ **finish**$([\ell, v])$ in π $\{\_.\, \mathsf{True}\}_{\top}$

JOIN-SPEC
$\{\mathsf{JoinHandle}^{\mathcal{N}}(\ell, \Phi)\}$ **join**$([\ell])$ in π $\{v.\, \Phi(v)\}_{\top}$

FIGURE 12.6: A Spawn-and-Join library

12.3 Spawn and Join

We now look at the verification of a spawn-and-join library, which allows us to spawn an arbitrary computation on a child thread, and wait for the child thread to receive the computation result. The implementation and specification are given in Figure 12.6.

The library provides 3 functions: **spawn**, **finish**, and **join**. **finish** and **join** are meant to be used—and indeed are implemented exactly—as message passing: once the child thread is done with the computation, it calls **finish**$([\ell, v])$, which writes the computation result v non-atomically to $\ell + 1$, and signals the completion with a release write of 1 to ℓ. A waiting thread calls **join**$([\ell])$, which waits with a repeat loop reading ℓ, and once the loop finishes it reads the result from $\ell + 1$, and cleans up the memory block of ℓ. The child thread can be spawned with **spawn**$([f])$, where f is the computation that is to be executed in parallel. **spawn** simply allocates the block of size 2. The allocated block with the base location ℓ is initialized with 0 for ℓ—we do not really need to initialize $\ell + 1$ (this applies to **mp** too). **spawn** then forks a new thread with the computation f, which takes in the location ℓ to report the result. It is assumed that f will call **finish**$([\ell, v])$ at the end. If this is not the case, then f does not need to take ℓ as an argument but we assume f returns the computation result, and we can then change **spawn**'s line 3 to **fork** $\{$ **finish**$([\ell, f()])$ $\}$.

The specifications of the library functions assume a predicate $\Phi : \mathit{Val} \to \mathsf{vProp}$, where the computation is expected to produce not just some value v, but also the resource $\Phi(v)$. The specifications assume a namespace $\mathcal{N}$ that will be used to allocate the library invariant. The specifications then involve two assertions: the finish handle $\mathsf{FinishHandle}^{\mathcal{N}}(\ell, \Phi)$ and the join handle $\mathsf{JoinHandle}^{\mathcal{N}}(\ell, \Phi)$. Intuitively, both handles will be generated by **spawn**, and then the finish handle will be given to the computation f, which eventually must call **finish**, while the join handle will be given to the caller of **join**. As such, SPAWN-SPEC says that **spawn** (1) assumes for the computation f a weakest pre-condition that will

consume the finish handle that **spawn** generates, and (2) returns the join handle in the post-condition. FINISH-SPEC says that once f produces the result v and finally calls **finish**, it consumes the finish handle and release the resource $\Phi(v)$. JOIN-SPEC says that **join** consumes the join handle generated by **spawn** and once it is done the resulting resource $\Phi(v)$ is returned.

Note that we could have stated the pre-condition of SPAWN-SPEC with a Hoare triple for f, *i.e.*,

$$\forall \ell, \rho. \left\{\mathsf{FinishHandle}^{\mathcal{N}}(\ell, \Phi)\right\} f([\ell]) \text{ in } \rho \left\{\_.\ \mathsf{True}\right\}_{\top}$$

However, stating as it in in SPAWN-SPEC, we allow the client to use extra resources to verify their f. Recall that a Hoare triple is defined with a persistent modality ($\Box$), so if we use a Hoare triple the client can only use $\mathsf{FinishHandle}^{\mathcal{N}}(\ell, \Phi)$ for the proof of f.

Definition 12.4 (Invariant for Spawn-and-Join)**.** The invariant of Spawn-and-Join is almost the same as that for **mp_reclaim** (Definition 12.3), except that the message to be sent is now some value v together with the resource $\Phi(v)$.

$$\begin{aligned}
&\mathsf{spawnJoinI}(\ell, \gamma_l, \gamma) : \mathsf{vProp} ::= \\
&\quad \exists h, b, t_0, V_0.\ \ell \mapsto^{\gamma_\ell}_{\mathtt{sw}} h * \mathbf{let}\ h_0 := [t_0 \leftarrow (0, V_0)]\ \mathbf{in} \\
&\quad \mathbf{if}\ b = \mathbf{false}\ \mathbf{then}\ h = h_0\ \mathbf{else} \\
&\quad \exists t_1 > t_0, V_1.\ h = [t_0 \leftarrow (0, V_0)][t_1 \leftarrow (1, V_1)] \\
&\qquad * \mathbb{@}_{V_1}(\heartsuit^{\gamma}_{1/2} * \exists v.\ \ell + 1 \mapsto v * \Phi(v))
\end{aligned}$$

Definition 12.5 (Model of Handles)**.**

$$\begin{aligned}
\mathsf{FinishHandle}^{\mathcal{N}}(\ell, \Phi) ::= \exists \gamma_\ell, \gamma, t, V.\ \ell + 1 \mapsto \not{\mathbb{Q}} * \ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sw}} [t \leftarrow (0, V)] \\
* \heartsuit^{\gamma}_{1/2} * \boxed{\mathsf{spawnJoinI}(\ell, \gamma_l, \gamma)}^{\gamma, \mathcal{N}} \\
\mathsf{JoinHandle}^{\mathcal{N}}(\ell, \Phi) ::= \exists \gamma_\ell, \gamma, t, V.\ \ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sn}} [t \leftarrow (0, V)] * \dagger^2 \ell \\
* \heartsuit^{\gamma}_{1/2} * \boxed{\mathsf{spawnJoinI}(\ell, \gamma_l, \gamma)}^{\gamma, \mathcal{N}}
\end{aligned}$$

The model for the finish and join handles mirror exactly what we have seen in the proof of **mp_reclaim** (Figure 12.4). In particular, the finish handle $\mathsf{FinishHandle}^{\mathcal{N}}(\ell, \Phi)$ carries the resources needed to send the message (but without $\Phi(v)$), mirroring the resources owned by the thread ρ of **mp_reclaim**: (1) the non-atomic points-to of the location $\ell + 1$ that will be used to store the result, (2) the single-writer ownership of the flag ℓ, and (3) the half token $\heartsuit^{\gamma}_{1/2}$ and the knowledge of the cancelable invariant $\boxed{\mathsf{spawnJoinI}(\ell, \gamma_l, \gamma)}^{\gamma, \mathcal{N}}$. The join handle $\mathsf{JoinHandle}^{\mathcal{N}}(\ell, \Phi)$ carries the resources needed to receive the message and to clean up the memory locations, mirroring the resources owned by the thread π of **mp_reclaim** from line π4: (1) the seen-history observation of the flag ℓ, (2) the block ownership of ℓ, and (3) the other half token $\heartsuit^{\gamma}_{1/2}$ and the knowledge of the same cancelable invariant.

The proofs of FINISH-SPEC and JOIN-SPEC then follow almost the same proof of **mp_reclaim** in Figure 12.4. The main differences is the extra

Context: let $P := \left(\forall \ell, \rho.\, \mathsf{FinishHandle}^{\mathcal{N}}(\ell, \Phi) \mathrel{-\!\!*} \mathsf{wp}\, f([\ell])\, \mathsf{in}\, \rho\, \{\_.\, \mathsf{True}\}[\top]\right)$

$\{P\}$

1: **let** $\ell := \mathbf{alloc}(2)$ **in**

2: $\ell :=_{\mathsf{na}} 0$; $\left\{\ell \mapsto 0 * \ell + 1 \mapsto \not{\mathfrak{Z}} * \dagger^2 \ell\right\}$ // NA-ALLOC and NA-WRITE

$\left\{P * \ell + 1 \mapsto \not{\mathfrak{Z}} * \dagger^2 \ell * \exists \gamma_\ell, t_0, V_0.\, \sqsupseteq V_0 * \ell \mapsto^{\gamma_\ell}_{\mathtt{sw}} [t_0 \leftarrow (0, V_0)] * \ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sw}} [t_0 \leftarrow (0, V_0)]\right\}$ // NA-AT-SW

$\left\{P * \ell + 1 \mapsto \not{\mathfrak{Z}} * \dagger^2 \ell * \ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sw}} [t_0 \leftarrow (0, V_0)] * \heartsuit^{\gamma}_1 * \boxed{\mathsf{spawnJoinI}(\ell, \gamma_\ell, \gamma)}^{\gamma, \mathcal{N}}\right\}$ // CINV-ALLOC-OPEN

Context: $\boxed{\mathsf{spawnJoinI}(\ell, \gamma_\ell, \gamma)}^{\gamma, \mathcal{N}}$

$\left\{P * \ell + 1 \mapsto \not{\mathfrak{Z}} * \ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sw}} [t_0 \leftarrow (0, V_0)] * \heartsuit^{\gamma}_{1/2} * \ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sn}} [t_0 \leftarrow (0, V_0)] * \dagger^2 \ell * \heartsuit^{\gamma}_{1/2}\right\}$

// AT-SW-SY and AT-SY-SN and CINV-TOK-FRAC

$\left\{P * \mathsf{FinishHandle}^{\mathcal{N}}(\ell, \Phi) * \mathsf{JoinHandle}^{\mathcal{N}}(\ell, \Phi)\right\}$

3: **fork**$\{\ldots\}$ // HOARE-FORK

Child thread:

$\left\{P * \mathsf{FinishHandle}^{\mathcal{N}}(\ell, \Phi)\right\}$

$f([\ell])\, \mathsf{in}\, \rho$

$\{\_.\, \mathsf{True}\}$ // By applying P

$\left\{\mathsf{JoinHandle}^{\mathcal{N}}(\ell, \Phi)\right\}$

4: ℓ $\left\{\ell.\, \mathsf{JoinHandle}^{\mathcal{N}}(\ell, \Phi)\right\}$

FIGURE 12.7: Hoare proof outlines for SPAWN-SPEC

releasing and acquiring of the resulting resource $\Phi(v)$. In the proof of FINISH-SPEC, corresponding to line $\rho 2$ of **mp_reclaim**, when invoking AT-WRITE-SW-REL-VJ we also release $\Phi(v)$ to the view V_1, *i.e.*, we get $@_{V_1}\Phi(v)$ in the post-condition, which we then put into the invariant. In the proof of JOIN-SPEC, at the end of the loop in line $\pi 4$ of **mp_reclaim**, we acquire also $@_{V_1}\Phi(v)$, which again can be used with VA-ELIM to get $\Phi(v)$.

The proof of SPAWN-SPEC also follows the proof of **mp_reclaim**. We give its Hoare proof outlines in Figure 12.7.

12.4 A Release-Acquire Treiber Stack

We now look at the verification of a linked-list based Treiber stack implementation using release-acquire and relaxed accesses, given in Figure 12.8. The interface includes 5 functions:

1. **new_stack**$([])$ allocates and returns a new, empty stack handle s;
2. **push**$([s, v])$ pushes a new element v into the stack s;
3. **pop**$([s])$ pops and returns an element in s, unless s is empty then it returns 0;
4. **try_push**$([s, v])$ is a *try* version of **push**, which returns true if it have successfully pushed to the stack s, or returns false if it fails to do so. **try_push** can fail due to contention by concurrent pushes/pops. Unlike **try_push**, **push** would try again until it succeeds.
5. **try_pop**$([s, v])$ is a *try* version of **pop**. It returns an element from the stack s, or returns empty (0), or returns -1 in case it fails due to

```
new_stack ::=
  λ[].
  1: let s := alloc(1) in
  2: s :=na 0;
  3: s

try_push_swap ::=
  λ[s, n].
  1: let s_h := *rlx s in
  2: n :=na s_h;
  3: CAS^rel(s, s_h, n)

try_push ::=
  λ[s, v].
  1: let n := alloc(2) in
  2: (n + 1) :=na v; // write elem
  3: if try_push_swap([s, n])
  4:  then true
  5:  else free(n, 2); false

push ::=
  λ[s, v].
  1: let n := alloc(2) in
  2: (n + 1) :=na v; // write elem
  3: repeat (try_push_swap([s, n]))

try_pop ::=
  λ[s].
  1: let s_h := *acq s in
  2: if s_h == 0 // null
  3:  then 0 // EMPTY
  4:  else
  5:   let n := *na s_h
  6:   if CAS^acq(s, s_h, n) // swap
  7:    then *na(s_h + 1) // read elem
  8:    else −1 // FAIL
pop ::=
  rec pop([s]) :=
  1: let v := try_pop([s]) in
  2: if v == −1
  3:  then pop([s]) else v
```

FIGURE 12.8: A simple release-acquire implementation for Treiber stacks

contention by concurrent pushes/pops. Unlike `try_pop`, `pop` would try again until it succeeds or until it finds that the stack is empty.

12.4.1 *A Release-Acquire Implementation of the Treiber Stack*

The implementation employs a list-linked based representation of the stack, which only requires *non-atomic* (**na**) accesses, because once a node is initialized it is never changed. The only atomic location is the location for the head of the stack. We note that the implementation does not take care of resource reclamation, and nor do we have an interface for that. The implementation includes an extra internal function `try_push_swap`.

- `new_stack`([]) allocates a location s that will store the pointer to the *head* node, *i.e.*, the top of the stack. s is initialized to null, *i.e.*, the value 0 (line 2).

- `try_push`($[s, v]$) starts with allocating a *node* n of size 2 to store the to-be-pushed element v. The value v is stored at the location $n + 1$ (line 2), while the location n will be used to store the pointer to the *next* node in the linked list of the stack. `try_push` then calls `try_push_swap`($[s, n]$) to try to swap n in as the new head (line 3). The function returns **true** if the try succeeds. Otherwise, it cleans up the node n and returns **false** (line 5).

- `try_push_swap`($[s, n]$) tries to read the pointer s_h to the (potentially current) head of the stack s, and then stores s_h in the *next* "field" of the new node n (which is n) in line 2. Then in 3 it uses a release compare-and-swap (**CAS**) instruction to try to swap n for s_h in the location s. The function returns the value returned by the **CAS**. If

the **CAS** succeeds, it returns **true**, and the stack pointer s points-to the new head node n, which in turn points-to the old head node s_h.

If the **CAS** fails, it returns **false** and nothing changes but the node n's next field. The **CAS** may fail if s_h was not really the current head of s. This could be because the relaxed in line 1 may read a stale value, or because there are concurrent push or pop operations that have updated the head of the stack while the function was running between lines 1 and 3.

Note that the **CAS** is a release (**rel**) **CAS**, *i.e.*, it uses only the release write access mode for the successful write, and it only uses the relaxed access mode for the reads in both success and failure cases. The release write mode allows a successful push to release anything that happens before it to the matching successful pop.

- **push**$([s, v])$ is similar to **try_push**: it allocates and initializes a new node n, but instead of calling **try_push_swap** only once, it will keep calling **try_push_swap** until it succeeds.

- **try_pop**$([s])$ tries to read the top (first) node of the stack—which the head pointer points to—and the second node, and swaps the head pointer to point to the second node. More specifically, in line 1, it reads the potentially current head of the stack into s_h. If s_h is null (0), it returns empty (0) immediately, in line 3. If s_h is not null, then in line 5 the function reads the pointer to s_h's next node, stored in the location s_h, into n. The function then uses an acquire **CAS** to swap n for s_h in the location s (line 6). If the **CAS** succeeds, then n will be the new top of the stack, and the function can read and return the element value of the old top, from the location $s_h + 1$, in line 7.

  If the **CAS** fails, then **try_pop** returns -1 to signal failure due to possible contention by concurrent operations. The **CAS** is an acquire (**acq**) **CAS**, *i.e.*, it uses the acquire mode for the successful read, but uses the relaxed mode for the successful write and the failure read. This is because a pop does not try to release anything. Instead, it acquires what the matching push that it reads from was releasing. In other words, the happens-before hb relation is only established between matching pairs of a successful push and a successful pop.

- **pop**$([s])$ simply calls **try_pop**. It only returns if **try_pop** returns an actual element or the empty value 0. If **try_pop** fails due to contention, **pop** tries again.

We note again the all accesses to the link-list's nodes are non-atomic. Furthermore, only the successful write of a push operation uses the release access mode, and in a pop operation, only the read of the head in 1 and the successful read by the **CAS** in 6 use the acquire access mode. Other memory accesses use relaxed access mode.

STACK-BAG-NEW
$\{\mathsf{True}\}\ \texttt{new\_stack}([])\ \mathsf{in}\ \pi\ \{s.\ \mathsf{isStack}^{\mathcal{N}}(s, \Phi)\}_{\top}$

STACK-BAG-TRY-PUSH
$\{\mathsf{isStack}^{\mathcal{N}}(s, \Phi) * \Phi(v)\}\ \texttt{try\_push}([s, v])\ \mathsf{in}\ \pi\ \{b.\ (b = \texttt{false} * \Phi(v)) \lor b = \texttt{true}\}_{\top}$

STACK-BAG-TRY-POP
$\{\mathsf{isStack}^{\mathcal{N}}(s, \Phi)\}\ \texttt{try\_pop}([s])\ \mathsf{in}\ \pi\ \{v.\ v = -1 \lor v = 0 \lor \Phi(v)\}_{\top}$

STACK-BAG-PUSH
$\{\mathsf{isStack}^{\mathcal{N}}(s, \Phi) * \Phi(v)\}\ \texttt{push}([s, v])\ \mathsf{in}\ \pi\ \{\_.\ \mathsf{True}\}_{\top}$

STACK-BAG-POP
$\{\mathsf{isStack}^{\mathcal{N}}(s, \Phi)\}\ \texttt{pop}([s])\ \mathsf{in}\ \pi\ \{v.\ v = 0 \lor \Phi(v)\}_{\top}$

STACK-BAG-TRY-PUSH-SWAP
$\{\mathsf{isStack}^{\mathcal{N}}(s, \Phi) * \Phi(v) * n \mapsto \_ * (n+1) \mapsto v * \dagger^2 n\}$

$\texttt{try\_push\_swap}([s, v])\ \mathsf{in}\ \pi$

$\{b.\ (b = \texttt{false} * \Phi(v) * n \mapsto \_ * (n+1) \mapsto v * \dagger^2 n) \lor b = \texttt{true}\}_{\top}$

FIGURE 12.9: Bag or per-element specifications for Treiber stacks

12.4.2 *Bag or Per-element Specifications for Stacks*

We will verify the implementation in Figure 12.8 against the so-called *bag* or *per-element* specifications, given in Figure 12.9. These specifications are rather weak: they only establish a connection between matching pairs of successful push and pop operations. We will verify the implementation against stronger specifications that establish stack properties, *e.g.*, last-in-first-out (LIFO), in Part IV. Nevertheless, the bag specifications are sufficient as a good demonstration for iRC11—they were also used in GPS/iGPS.[6]

The specifications involve a persistent assertion $\mathsf{isStack}^{\mathcal{N}}(s, \Phi)$ which says that the location s is a stack tied to a predicate $\Phi : \mathit{Val} \to \mathsf{vProp}$. STACK-BAG-NEW says that $\texttt{new\_stack}([])$ returns a new stack s that satisfies $\mathsf{isStack}^{\mathcal{N}}(s, \Phi)$ for the user-chosen Φ and $\mathcal{N}$. The namespace $\mathcal{N}$ is needed to store the underlying invariant for the stack s. The predicate $\Phi(v)$ dictates the per-element resource that a push operation of v will release. As can be seen in STACK-BAG-TRY-PUSH and STACK-BAG-PUSH, a successful push of v to the stack s will consume $\Phi(v)$, while a failed `try_push` will not. On the other side, STACK-BAG-TRY-POP and STACK-BAG-POP says that a successful non-empty pop of some element v from the stack s will acquire the corresponding resource $\Phi(v)$ that has been released by the push of v. As such, the specifications at least recognize the synchronization between matching pairs of push and pop operations, by which a user-chosen resource $\Phi(v)$ can flow from one thread to another. Under the hood, this synchronization is established by the release-acquire synchronization through the stack location s (between line 3 of `try_push_swap` and line 1 of `try_pop`).

We also give the specification STACK-BAG-TRY-PUSH-SWAP for the internal function `try_push_swap`. The function assumes the non-atomic ownership of the locations n and $n+1$, and requires that $n+1$ has been set to the to-be-pushed value v. Additionally it also assumes the

[6] Turon et al., "GPS: navigating weak memory with ghosts, protocols, and separation" [TVD14]; Kaiser et al., "Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris" [Kai+17].

per-element resource $\Phi(v)$. All of these resources will be consumed if the function succeeds.

12.4.3 *Verification of the Treiber Stack against the Bag Specifications*

From the specification STACK-BAG-TRY-PUSH-SWAP for **try_push_swap**, we can easily verify **try_push** and **push** against STACK-BAG-TRY-PUSH and STACK-BAG-PUSH. Furthermore, we also can easily verify **pop** against STACK-BAG-POP assuming STACK-BAG-TRY-POP for **try_pop**. We therefore present the verifications of **new_stack**, **try_push_swap**, and **try_pop** below. As usual, we start with defining the invariant for the stack implementation.

Definition 12.6 (Invariant for Per-element Treiber Stack)**.**

$$\mathsf{null0}(v) ::= (v = \mathsf{Some}(\ell))\ ?\ \ell : 0$$

$$\mathsf{AllNodes}(vs) ::= \mathop{\Large *}_{(\mathsf{Some}(n),V)\in vs} \exists q, v.\, @_V(n \overset{q}{\mapsto} \mathsf{null0}(v))$$

$$\mathsf{InNodes}(S) ::= \mathop{\Large *}_{n=S(i)} \exists v.\, n+1 \mapsto v * \Phi(v) * \dagger^2 n * \exists q.\, n \overset{q}{\mapsto} \mathsf{null0}(S(i+1))$$

$$\begin{aligned}\mathsf{TreiberBI}(s,\gamma) ::= {}& \exists V_s, t_0, V_h, vs, S.\\ & \mathbf{let}\, v_h : \mathit{Loc}^? := \mathtt{hd}(S); vs' : \overline{(\mathit{Loc}^?, \mathit{View})} := vs +\!\!+ [(v_h, V_h)]\ \mathbf{in}\\ & \mathbf{let}\, h := [t_0 + i \leftarrow (\mathsf{null0}(v), V) \mid vs'(i) = (v, V)]\ \mathbf{in}\\ & @_{V_s}(s \mapsto^{\gamma}_{\mathtt{con}} h) * @_{V_h}\mathsf{InNodes}(S) * \mathsf{AllNodes}(vs') * \dagger^1 s\end{aligned}$$

$$\mathsf{isStack}^{\mathcal{N}}(s) ::= \exists \gamma, h.\, s \sqsupseteq^{\gamma}_{\mathtt{sn}} h * \boxed{\mathsf{TreiberBI}(s,\gamma)}^{\mathcal{N}}$$

The invariant content $\mathsf{TreiberBI}(s,\gamma)$ for some location s an its atomic period identifier γ (used by its atomic points-to) is objective, as we will put it in an objective invariant (§11.1) because we do not care to reclaim the stack. Nevertheless, we leave the block ownership ($\mathbf{free}(1,s)$ for the stack pointer and $\dagger^2 n$ for each node) in the definition as a reminder that we may want to extend the invariant to support reclamation of the stack. $\mathsf{isStack}^{\mathcal{N}}(s,\Phi)$ simply asserts the existence of the objective invariant for $\mathsf{TreiberBI}(s,\gamma)$ in $\mathcal{N}$, and a seen-history observation for s, needed to perform atomic operations on s.

The definitions rely on a null0 function that turns an optional location ($v \in \mathit{Loc}^?$) into a nullable location value, where null is the value 0. The invariant content $\mathsf{TreiberBI}(s,\gamma)$ is composed of 3 parts: the atomic points-to of s, the ownership of nodes (in S) that are currently in the stack, and the ownership of all nodes (in vs') that have ever been in the stack.

- The atomic points-to ownership $s \mapsto^{\gamma}_{\mathtt{con}} h$ of the stack pointer is put in the concurrent mode (`con`) and at some view V_s (to make it objective). The history h is a contiguous block of writes because we only use **CAS**'s on s.[7] This contiguous block of writes starts at the timestamp t_0, and contains the writes from the list

[7] Note that we do not use the CAS-only mode (`cas`) atomic points-to yet—that mode was developed mainly for GPS protocols in Part III. Nevertheless, the contiguous block-of-writes abstraction can also be a useful feature for CAS-only atomic points-to, but it has not been incorporated into atomic points-to.

$vs' \in \overline{(\mathit{Loc}^?, \mathit{View})}$ of pairs of optional locations and views. That is, the history of s only contains nullable location values.

The top write of the block, which is the latest write to s and the pointer to the current top node of the stack, is the message $(t_0 + |vs|, \mathsf{null0}(v_h), V_h)$,[8] where vs is the list of non-current writes to s. v_h is the head ($\mathtt{hd}$) of the stack's *abstract state* $S \in \overline{\mathit{Loc}}$, which is a list of pointers to nodes in the stack. We model the abstract stack as a list whose top element is at index 0 and bottom element is at index $(|S| - 1)$. If the abstract stack is empty, *i.e.*, $S = []$, then $\mathtt{hd}(S) = \mathsf{None}$, and the latest write to s has the value $\mathsf{null0}(\mathsf{None}) = 0$, *i.e.*, the null value.

[8] Indices i to a list start at 0.

- $\mathsf{InNodes}(S)$ contains ownership of all nodes that are currently in the stack. It constructs a singly-linked list starting from the top node of the abstract stack S, which is at index 0, and ending at the bottom node at index $(|S| - 1)$. The ownership of each node is grouped together using the big separating conjunction $\ast$.

  For each node $S(i) = n \in \mathit{Loc}$ that is currently in the stack, we have: (1) the full non-atomic points-to ownership $n + 1 \mapsto v$ of the *data* field with a pushed value v, and (2) the corresponding released resource $\Phi(v)$ of the push, and (3) a fractional non-atomic points-ownership $n \overset{q}{\mapsto} \mathsf{null0}(S(i+1))$ of n's *next* field that points-to the next node in the stack. If $(i + 1)$ is out-of-bound, then $S(i + 1)$ is None and the next field of the bottom node will store the null value (0).

- $\mathsf{AllNodes}(vs')$ contains fractional ownership of next fields of all nodes (vs') that have ever been in the stack. For every write to s with a non-null location n (a node) and message view V, it owns a fraction of the non-atomic points-to $n \overset{q}{\mapsto} \mathsf{null0}(v))$ for n's next field, which has some nullable location value v. The fractional points-to is put at the view V of the write message to the stack location s. $\mathsf{AllNodes}(vs')$ is needed because a pop operation can read the next field of *any* node that has ever been in the stack (line 5 of **`try_pop`**).

Remark 12.7 (On Reclaiming Stack Resources)**.** Looking at the definition of TreiberBI, we can see that the main problem with reclaiming the stack's resources is in the non-atomic points-to ownership of the nodes' next fields: they are split into fractions that are not carefully tracked (existentially quantified), and there are overlapping ownership between $\mathsf{InNodes}(S)$ and $\mathsf{AllNodes}(vs')$. $\mathsf{AllNodes}(vs')$ is needed because of the non-atomic read of a node's next field in **`try_pop`**'s line 5. More specifically, we need to acquire some fraction q of n's non-atomic in line 1, so that at line 5 we can read n. To enable reclamation, we either have to recollect this fraction q, or avoid giving out a fraction at all. With the latter choice, we keep the full ownership of n in the invariant, and turn the non-atomic read in 5 into a relaxed read of n, which now can open the invariant to access n. Naturally, we have to avoid the overlaps between $\mathsf{InNodes}(S)$ and $\mathsf{AllNodes}(vs')$, by carefully split the nodes into

those currently in the stack, and those that have been but are no longer in the stack. Once all of that is set up, we use a cancelable invariant (§11.2) to be able to reclaim all resources that have ever been put into the stack.

We now look at the proof sketches of the most 3 important functions.

Proof sketch of **new_stack**. The proof is easy. We use NA-ALLOC and NA-WRITE (§9.1) respectively for allocation and initialization of s. Then we use NA-AT-SW (Figure 10.2) to turn the non-atomic points-to to an atomic points-to in single-writer mode (`sw`) with some atomic period identifier γ, and AT-SW-CON to switch the mode to concurrent (`con`). We then use VA-INTRO to put the atomic points-to of s at some view V_s. From the allocated resources we can easily construct TreiberBI(s, γ), because the history of s is a singleton with a null location value. We use OINV-ALLOC-OBJ (§11.1) to allocate the invariant, and then we are done.

$\{\mathsf{True}\}$
1: **let** $s := \mathbf{alloc}(1)$ **in**
$\{s.\ \dagger^1\ s * s \mapsto \not{\textcircled{2}}\}$ // NA-ALLOC
2: $s :=_{\mathbf{na}} 0$; $\{\dagger^1\ s * s \mapsto 0\}$ // NA-WRITE
$\{\dagger^1\ s * \exists\gamma, t_0, V_h.\ s \mapsto^{\gamma}_{\mathtt{sw}} [t_0 \leftarrow (0, V_h)] * s \sqsupseteq^{\gamma}_{\mathtt{sw}} [t_0 \leftarrow (0, V_h)]\}$
$\{\dagger^1\ s * s \sqsupseteq^{\gamma}_{\mathtt{sn}} [t_0 \leftarrow (0, V_h)] * \exists V_s.\ @_{V_s}(s \mapsto^{\gamma}_{\mathtt{con}} [t_0 \leftarrow (0, V_h)])\}$
// NA-AT-SW and AT-SW-CON and AT-SW-SY and AT-SY-SN and VA-INTRO
$\{s \sqsupseteq^{\gamma}_{\mathtt{sn}} [t_0 \leftarrow (0, V_h)] * \mathsf{TreiberBI}(s, \gamma)\}$
3: $s\ \{s.\ \mathsf{isStack}^{\mathcal{N}}(s, \Phi)\}$ // OINV-ALLOC-OBJ

□

Proof sketch of **try_push_swap**. The proof is also easy. We open the invariant twice using OINV-ACC-OBJ, in lines 1 and 3 to get access to the atomic points-to of s to read or **CAS** on it. In line 1 we do not change the invariant. In line 3, if the **CAS** succeeds, then we extend the abstract state S with our new node n into $[n] +\!\!+ S$ and put all of the resources in the pre-condition to that node in the invariant. The proof outlines are given in Figure 12.10.

The most important point is that we can easily establish deterministic pointer comparison (the conditions involving P_{cmp}) when performing the **CAS**, because the location s only stores nullable locations, whose points-to ownership (if non null) are kept inside the invariant TreiberBI and thus we know that they are all alive. □

Proof sketch of **try_pop**. The proof is not so complicated. We again open the invariant twice using OINV-ACC-OBJ, in lines 1 and 6 to get access to the atomic points-to of s to read or **CAS** on it. In line 1, we do not change the invariant, but if we read s_h to be non-null, we also acquire from AllNodes(vs') some fraction q of s_h's non-atomic points-to for its next field, *i.e.*, $s_h \overset{q}{\mapsto} \mathsf{null0}(n)$ for some n. This fraction will be used to read s_h's next field in line 4.[9]

In line 6, if the **CAS** succeeds, we know that s_h is indeed the top of the current abstraction S, *i.e.*, $S = [s_h] +\!\!+ S'$ for some S'. We then

[9] And it is also the source of leaking ownership of nodes in this proof.

$\left\{\mathsf{isStack}^{\mathcal{N}}(s,\Phi) * \Phi(v) * n \mapsto \_ * (n+1) \mapsto v * \dagger^2 n\right\}$

Context: $\boxed{\mathsf{TreiberBI}(s,\gamma)}^{\mathcal{N}}$

$\left\{s \sqsupseteq^{\gamma}_{\mathsf{sn}} h_0 * \Phi(v) * n \mapsto \_ * (n+1) \mapsto v * \dagger^2 n\right\}_{\top}$

OINV-ACC-OBJ

$\left\{s \sqsupseteq^{\gamma}_{\mathsf{sn}} h_0 * \sqsupseteq\varnothing * \triangleright \mathsf{TreiberBI}(s,\Phi)\right\}_{\top\setminus\mathcal{N}}$ // VS-BOT

1: **let** $s_h :=$ *[rlx] s **in**

$\{s_h.\, \exists h_0', t_h.\, h_0'(t_h) = (s_h, \_) * s \sqsupseteq^{\gamma}_{\mathsf{sn}} h_0' * \mathsf{TreiberBI}(s,\Phi)\}_{\top\setminus\mathcal{N}}$ // AT-READ-SN

$\left\{s \sqsupseteq^{\gamma}_{\mathsf{sn}} h_0' * \Phi(v) * n \mapsto \_ * (n+1) \mapsto v * \dagger^2 n\right\}_{\top}$

2: $n :=_{\mathsf{na}} s_h$; $\left\{\Phi(v) * n \mapsto s_h * (n+1) \mapsto v * \dagger^2 n\right\}_{\top}$ // NA-WRITE

$\left\{s \sqsupseteq^{\gamma}_{\mathsf{sn}} h_0' * \exists V_0.\, \sqsupseteq V_0 * @_{V_0}(\Phi(v) * n \mapsto s_h * (n+1) \mapsto v) * \dagger^2 n * \triangleright \mathsf{TreiberBI}(s,\gamma)\right\}_{\top\setminus\mathcal{N}}$ // VA-INTRO

$\left\{\begin{array}{l} s \sqsupseteq^{\gamma}_{\mathsf{sn}} h_0' * \sqsupseteq V_0 * @_{V_0}(\Phi(v) * n \mapsto s_h * (n+1) \mapsto v) * \dagger^2 n * \\ @_{V_s}(s \mapsto^{\gamma}_{\mathsf{con}} h) * @_{V_h}\mathsf{InNodes}(S) * \mathsf{AllNodes}(vs') \end{array}\right\}$ // Unfolding TreiberBI

We know s_h is a value written to h at t_h, because $h_0' \subseteq h$.

3: $\mathbf{CAS}^{\mathbf{rel}}(s, s_h, n)$

OINV-ACC-OBJ

$\left\{\begin{array}{l} b.\, \exists h', V' \sqsupseteq V_0, v'.\, \sqsupseteq V' * @_{V_s \sqcup V'}(s \mapsto^{\gamma}_{\mathsf{con}} h') * \\ @_{V'}(\Phi(v) * n \mapsto s_h * (n+1) \mapsto v) * \dagger^2 n * @_{V_h}\mathsf{InNodes}(S) * \mathsf{AllNodes}(vs') * \dagger^1 s * \\ ((b = \mathbf{false} * s_h \neq v' * h' = h) \vee (b = \mathbf{true} * s_h = v' * \exists V_w \sqsupseteq V' \sqcup V_h.\, h' = h[t_h + 1 \leftarrow (n, V_w)])) \end{array}\right\}_{\top\setminus\mathcal{N}}$

// AT-CAS-SN-GEN Note that any location values readable from s are alive because we store their fractional non-atomic points-to ownership in $\mathsf{AllNodes}(vs')$.

$\left\{\begin{array}{l} \quad (b = \mathbf{false} * \Phi(v) * n \mapsto \_ * (n+1) \mapsto v * \dagger^2 n * \\ \qquad @_{V_s \sqcup V'}(s \mapsto^{\gamma}_{\mathsf{con}} h) * @_{V_h}\mathsf{InNodes}(S) * \mathsf{AllNodes}(vs') * \dagger^1 s) \\ \vee\ (b = \mathbf{true} * @_{V_s \sqcup V'}(s \mapsto^{\gamma}_{\mathsf{con}} h') * @_{V_w}\mathsf{InNodes}([n] +\!\!+ S) * \mathsf{AllNodes}(vs' +\!\!+ [(\mathsf{Some}(n), V_w)]) * \dagger^1 s) \end{array}\right\}$

In the failure case ($b = \mathbf{false}$), we use VA-ELIM to get back the original resources without the view-at modality.

In the successful case ($b = \mathbf{true}$), the resources are put inside InNodes and AllNodes of the invariant.

$\left\{\begin{array}{l} \quad (b = \mathbf{false} * \Phi(v) * n \mapsto \_ * (n+1) \mapsto v * \dagger^2 n * \mathsf{TreiberBI}(s,\gamma)) \\ \vee\ (b = \mathbf{true} * \mathsf{TreiberBI}(s,\gamma)) \end{array}\right\}_{\top\setminus\mathcal{N}}$

$\left\{b.\, (b = \mathbf{false} * \Phi(v) * n \mapsto \_ * (n+1) \mapsto v * \dagger^2 n) \vee b = \mathbf{true}\right\}_{\top}$

FIGURE 12.10: Hoare proof outlines for `try_push_swap`

can acquire the resources for s_h in $\mathsf{InNodes}(S)$, and leave the rest as $\mathsf{InNodes}(S')$ for the new abstract state S'.

We give the Hoare proof outlines in Figure 12.11. We note again that we always have deterministic pointer comparison with **CAS**'s, because ownership of all reachable locations in the stack are stored in the invariant TreiberBI, so we know they are all alive. □

Remark 12.8 (On the Use of Acquire **CAS** in `try_pop`)**.** We note that we do not really need the **CAS** of `try_pop` (line 6) to use the acquire access mode. In fact, the acquire access mode used in the read of the stack pointer in line 1 alone should be sufficient, and such implementation should also be verifiable in iRC11. Intuitively, by reading acquire in `try_pop`'s line 1, we have been synchronized with the write of n to s, whose view is V_n, *i.e.*, we acquire $\sqsupseteq V_n$. The view V_n should be bigger

$\left\{\mathsf{isStack}^{\mathcal{N}}(s,\Phi)\right\}_{\top}$

Context: $\boxed{\mathsf{TreiberBI}(s,\gamma)}^{\mathcal{N}}$

$\{s \sqsupseteq^{\gamma}_{\mathtt{sn}} h_0\}_{\top}$

OINV-ACC-OBJ

$\{s \sqsupseteq^{\gamma}_{\mathtt{sn}} h_0 * \sqsupseteq\varnothing * \triangleright\,\mathsf{TreiberBI}(s,\Phi)\}_{\top\setminus\mathcal{N}}$ // VS-BOT

1: **let** $s_h := {}^{*\mathsf{acq}}s$ **in**

$\{s_h.\, \exists h_0', t_h, V_0.\, h_0'(t_h) = (s_h, V_0) * \sqsupseteq V_0 * s \sqsupseteq^{\gamma}_{\mathtt{sn}} h_0' * \mathsf{TreiberBI}(s,\Phi) * (s_h = 0 \lor s_h \neq 0 * \exists q, n.\, @_{V_0} s_h \overset{q}{\mapsto} n)\}_{\top\setminus\mathcal{N}}$

// AT-READ-SN-ACQ

Context: $h_0'(t_h) = (s_h, V_0) * \sqsupseteq V_0 * s \sqsupseteq^{\gamma}_{\mathtt{sn}} h_0'$

$\{\mathsf{TreiberBI}(s,\Phi) * (s_h = 0 \lor s_h \neq 0 * s_h \overset{q}{\mapsto} n)\}_{\top\setminus\mathcal{N}}$ // VA-ELIM

$\{s \sqsupseteq^{\gamma}_{\mathtt{sn}} h_0' * (s_h = 0 \lor s_h \neq 0 * s_h \overset{q}{\mapsto} n)\}_{\top}$

2: **if** $s_h == 0$ // null

Empty

$\{s_h = 0\}$

3: **then** 0 $\{v.\, v = 0\}_{\top}$

4: **else**

$\{s_h \overset{q}{\mapsto} n\}$

5: **let** $n := {}^{*\mathsf{na}}s_h$ $\{s_h \overset{q}{\mapsto} n\}_{\top}$ // NA-READ

Non-empty

OINV-ACC-OBJ

$\{s_h \overset{q}{\mapsto} n * \triangleright\,\mathsf{TreiberBI}(s,\gamma)\}_{\top\setminus\mathcal{N}}$

$\left\{s_h \overset{q}{\mapsto} n * \sqsupseteq V_0 * s \sqsupseteq^{\gamma}_{\mathtt{sn}} h_0' * s \mapsto^{\gamma}_{\mathtt{con}} h * @_{V_h}\mathsf{InNodes}(S) * \mathsf{AllNodes}(vs') * \dagger^1 s\right\}_{\top\setminus\mathcal{N}}$

// Unfolding TreiberBI

We know s_h is a value written to h at t_h, because $h_0' \subseteq h$. specifically, $h(t_h) = (s_h, V_0)$

6: **if** $\mathsf{CAS}^{\mathsf{acq}}(s, s_h, n)$

$$\left\{\begin{array}{l} b.\, s_h \overset{q}{\mapsto} n * \exists h', V' \sqsupseteq V_0, v'.\, \sqsupseteq V' * @_{V_s \sqcup V'}(s \mapsto^{\gamma}_{\mathtt{con}} h') * @_{V_h}\mathsf{InNodes}(S) * \mathsf{AllNodes}(vs') * \dagger^1 s * \\ \quad (b = \mathbf{false} * s_h \neq v' * h' = h) \\ \lor\, (b = \mathbf{true} * s_h = v' * V_0 = V_h \sqsubseteq V' * t_h = \mathtt{max}(\mathtt{dom}(h)) * \exists V_w \sqsupseteq V_h.\, h' = h[t_h + 1 \leftarrow (n, V_w)]) \end{array}\right\}_{\top\setminus\mathcal{N}}$$

// AT-CAS-SN-GEN Noe that any location values readable from s are alive because we store their fractional non-atomic points-to ownership in $\mathsf{AllNodes}(vs')$.

$$\left\{\begin{array}{l} b.\, s_h \overset{q}{\mapsto} n * \quad (b = \mathbf{false} * \mathsf{TreiberBI}(s,\gamma)) \\ \qquad\qquad \lor\, (b = \mathbf{true} * S = [(s_h, V_h)] +\!\!+ S' * @_{V_s \sqcup V'}(s \mapsto^{\gamma}_{\mathtt{con}} h') * \\ \qquad\qquad\quad @_{V_w}\mathsf{InNodes}(S') * \mathsf{AllNodes}(vs' +\!\!+ [(n, V_w)]) * \dagger^1 s * \\ \qquad\qquad\quad @_{V_h}(\exists v, q'.\, \Phi(v) * s_h + 1 \mapsto v * \dagger^2 s_h * s_h \overset{q'}{\mapsto} n)) \end{array}\right\}_{\top\setminus\mathcal{N}}$$

$\{b.\, s_h \overset{q}{\mapsto} n * \mathsf{TreiberBI}(s,\gamma) * (b = \mathbf{false} \lor (b = \mathbf{true} * \exists v, q'.\, \Phi(v) * s_h + 1 \mapsto v * \dagger^2 s_h * s_h \overset{q'}{\mapsto} n))\}_{\top\setminus\mathcal{N}}$

// VA-ELIM since we have $\sqsupseteq V'$ and $V_h \sqsubseteq V'$

7: **then** ${}^{*\mathsf{na}}(s_h + 1)$ $\left\{v.\, \Phi(v) * s_h + 1 \mapsto v * \dagger^2 s_h * s_h \overset{q+q'}{\mapsto} n\right\}_{\top}$

8: **else** -1 $\{v.\, v = -1 * s_h \overset{q}{\mapsto} n\}_{\top}$

$\{v.\, v = -1 \lor \Phi(v)\}_{\top}$ // Dropping points-to and block ownership of s_h

$\{v.\, v = -1 \lor v = 0 \lor \Phi(v)\}_{\top}$

FIGURE 12.11: Hoare proof outlines for `try_pop`

than the view at which the node n was pushed to the stack, so the seen-view observation $\sqsupseteq V_n$ should allow us to access the resources released together with the push of n, *i.e.*, $(\exists v.\, n+1 \mapsto v * \Phi(v) * \ldots)$ in $\mathsf{InNodes}(S)$.

Unfortunately, the twist is that n can be written multiple times to s, each time with an increasingly bigger view V_n. This can happen when n keeps coming back as the top of the stack, because some other nodes are pushed on top of n and then popped. Each time n comes back as the top of the stack, it is written to s with a bigger view V_n, because we only **CAS** on s. As such, when we read n from s with the acquire mode in line 1, we may have read the i-th write of n to s and acquire some observation of V_n^i, *i.e.*, $\sqsupseteq V_n^i$. Meanwhile, the resources $(\exists v.\, n+1 \mapsto v * \Phi(v) * \ldots)$ we want to acquire (when the **CAS** on s line 6 succeeds) hold at the view V_n^0 of the very *first* write of n to s—the release write of the push of n to s. That is, we should have $@_{V_n^0}(\exists v.\, n+1 \mapsto v * \Phi(v) * \ldots)$ in the invariant. So if we want to acquire those resources with a fully relaxed **CAS** in line 6, we need to track the fact that $V_n^0 \sqsubseteq V_n^i$ for *all* i. With that, we can use $\sqsupseteq V_n^i$ we acquired in line 1 to eliminate the view-at modality and acquire $(\exists v.\, n+1 \mapsto v * \Phi(v) * \ldots)$. In short, to support the fully relaxed **CAS** in line 6 of **try_pop**, we need to adjust our invariant so that (1) for each node n, we keep the InNodes resources at the view V_n^0 of the first write of n to s (the write of the push of n to s), and (2) we know that any later write of n to s has a view V_n^i that is bigger than V_n^0.

The invariant $\mathsf{TreiberBI}(s, \gamma)$ (Definition 12.6) on the other hand is rather simple: we keep all InNodes resources at the view V_h of the latest write to s. As such, we need an acquire **CAS** to synchronize with that view V_h in other to acquire the resources of the top node in InNodes. To keep the proof simple, we have decided to present this invariant for the **try_pop** version with the acquire **CAS** (Figure 12.8).

CHAPTER SUMMARY. In this chapter we have demonstrated multiple features of iRC11, using several examples. We show how to switch from non-atomic to atomic points-to to share locations, how to build concurrent protocols using atomic and non-atomic points-to, how to perform atomic accesses with resources under view-explicit modalities, how to cancel invariants and eliminate the view-explicit modalities to regain shared resources, and how to convert atomic points-to back to non-atomic ones to deallocate or reuse them.

We have also seen that working with explicit views and histories are quite cumbersome, and in Part III we will have how to abstract them further with GPS protocols, and we will see more complex verifications of concurrent Rust libraries against their assigned Rust types in $\mathsf{RB}_{\mathtt{rlx}}$. Nevertheless, the GPS abstraction in Part III is not so convenient to work with when we have multiple atomic locations, and hiding views indeed weakens the logic. Furthermore, GPS protocols would not be sufficient to verify the **try_pop** version that uses a fully relaxed **CAS** (see Remark 12.8). In Part IV we will need explicit views and general invariants that can store and relate multiple atomic locations, in order to verify libraries against stronger logically-atomic specifications of Compass.

13

Related Work

He et al.[1] present GPS+, a extension to GPS that supports relaxed accesses and release/acquire fences. The logic, however, lacks support for relaxed RMW operations *e.g.*, **CAS**/**FAA** operations. Additionally, their logic is based on the axiomatic semantics of C11 which cannot be used together with Iris and thus would not allow us to extend the existing RustBelt development. GPS+ also does not support mechanized verification of programs.

[1] He et al., "GPS+: Reasoning About Fences and Relaxed Atomics" [He+18].

iGPS[2] supports a mechanism called *fractional protocols*, which is closely related to cancelable invariants. However, iGPS's fractional protocols are not as powerful as iRC11's cancelable invariants in that they cannot reclaim the resources governed by the protocol at the thread's local view. This is because the protocol tokens they use are modeled with unsynchronized ghost state. By using synchronized ghost state, we can support full reclamation of all resources governed by either iRC11 cancelable invariants or borrow propositions.

[2] Kaiser et al., "Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris" [Kai+17].

Tassarotti et al.[3] use GPS to verify an implementation of the Read-Copy-Update (RCU) technique. With GPS, they are able verify the reclamation of clients' non-atomic locations, but not RCU's internal atomic locations because they are governed by GPS's *non*-cancelable protocols. Kaiser et al. fixed this problem by re-verifying RCU in iGPS with their fractional protocols. However, as mentioned above, fractional protocols are not a general solution.

[3] Tassarotti et al., "Verifying read-copy-update in a logic for weak memory" [TDV15].

FSL and FSL++[4] are the first RMC logics to support fences. iRC11 directly inherits the idea of fence modalities from them, but provide a model for the modalities on top of the operational semantics of ORC11,[5] whereas FSL and FSL++ are proven directly on top of the axiomatic semantics of (R)C11. iRC11 also provides better support for mechanized verification of programs, thanks to the Iris proofmode.

[4] Doko and Vafeiadis, "A Program Logic for C11 Memory Fences" [DV16]; Doko and Vafeiadis, "Tackling Real-Life Relaxed Concurrency with FSL++" [DV17].

[5] see Section 8.3

Gotsman et al.[6] provide an SC-based logic where ownership of a location can be turned into a lock with fractional permissions for shared accesses, and later when the full permission of the lock is collected, the ownership of the location can be reclaimed. Hobor et al.[7] provide a similar mechanism that additionally allows attaching "invariant resources" to locks. Our cancelable invariants are more general than these mechanisms in that our cancelable invariants are not specifically tied to locks and are proven sound with respect to a much weaker memory model.

[6] Gotsman et al., "Local Reasoning for Storable Locks and Threads" [Got+07].

[7] Hobor et al., "Oracle Semantics for Concurrent Separation Logic" [HAN08].

Svendsen et al.[8] introduce the first program logic for the promising

[8] Svendsen et al., "A Separation Logic for a Promising Semantics" [Sve+18].

semantics. Their logic is based on RSL and supports relaxed accesses but not fences. Moreover, unlike FSL, it disallows the transfer of ownership through relaxed accesses, among other reasoning principles that have proven useful in $\mathsf{RB}_{\texttt{rlx}}$. Extending iRC11 to account for promises is a very interesting avenue for future work.

It is worth re-iterating that ORC11 and iRC11 currently do not support SC accesses and SC fences. In fact, we are not aware of any existing RMC separation logic that does.[9] Adding support for SC would enable us to verify some interesting and challenging fine-grained concurrent algorithms, such as the work-stealing queue,[10] as well as epoch-based resource reclamation schemes such as that implemented by Rust's `crossbeam` library.[11]

Very recently, Hammond et al.[12] introduces the first CSL for the ARMv8 "user" concurrency memory model,[13] named AxSL. AxSL provides rules similar to those of RSL, where resources can be transferred among threads following a per-location protocol Φ: a read of value v from the location ℓ should acquire the resource $\Phi(\ell, v)$, and a write of v to ℓ should release the resource $\Phi(\ell, v)$. However, unlike RSL which does not support ghost state as a resource and only allows resource transfer over release-acquire synchronization, AxSL is built in Iris (so resources can include arbitrary ghost state) and allows resource transfer over weaker accesses, such as relaxed ones, although not arbitrary. To be sound with respect to the ARMv8 RMC memory model, AxSL does not allow resources to flow along $\mathsf{po} \cup \mathsf{rf}$ which can be cyclic. Recall that all RSL, FSL, FSL++, GPS, iGPS, iRC11 assume $\mathsf{po} \cup \mathsf{rf}$ to be acyclic, and, as a result, resources in these logic can flow implicitly along $\mathsf{po} \cup \mathsf{rf}$, and we can get simple reasoning principles following program syntax. On the other hand, in the ARMv8 memory model, intra-thread concurrency can happen, as independent instructions can be executed and speculated concurrently. Consequently, while AxSL's reasoning principles still follow program syntax, the logic maintains that resources can only flow along dependencies that prevent intra-thread concurrent execution of instructions. Concretely, the logic supports explicit reasoning about memory events generated by the program and dependencies among them (edges among events in candidate execution graphs), and only allows resources to flow along the memory model's *ordered-before* relation ob (which includes the intra-thread locally-ordered-before relation lob and the inter-thread observed-by relation obs). The restriction is enforced by tying resources to events, *e.g.*, $a \rightsquigarrow \Phi(\ell, v)$, and by allowing an event b to acquire $\Phi(\ell, v)$ only if a is ordered before b. This construction is similar to FSL modalities, or our view-explicit modalities (§8.5), but at a finer-grained level of events.

Another impressive achievement of AxSL is that it is proven by instantiating Iris with a simple operationalisation of the axiomatic semantics of [Pul+18]. The operationalised semantics is very close to the original axiomatic one. We have conjectured that we can do the same with OGS (§3.6), but we have not put in the effort to do so. The benefit of this approach is that we can cut down the trusted code base (TCB), and avoid doing a correspondence proof as in §3.6. On the other hand, the model of predicates, such as that of the view-explicit ones or of weakest

[9] The abstract of the RSL paper ([VN13]) claims that it supports reasoning about SC accesses, but according to Vafeiadis [personal communication], this is a mistake, and indeed the body of the paper does not.

[10] Chase and Lev, "Dynamic circular work-stealing deque" [CL05].

[11] *Crossbeam: Tools for concurrent programming in Rust* [Www].

[12] Hammond et al., "An Axiomatic Basis for Computer Programming on the Relaxed Arm-A Architecture: The AxSL Logic" [Ham+24].

[13] Pulte et al., "Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8" [Pul+18].

preconditions, would become more complex. In fact, the authors of AxSL report a non-standard and challenging model and adequacy proof for their logic, which expectedly should be more complex than those done by Vafeiadis et al. for RSL and FSL.

Part III

RUSTBELT MEETS RELAXED MEMORY

14

Challenge: RustBelt and Relaxed Memory

The Rust programming languages[1] strike a delicate balance between safety and control using a *substructural* type system, in which types not only classify data but also represent *ownership* of resources, such as the right to read, write, or deallocate a piece of memory. By tracking ownership in the types, Rust is able to prohibit dangerous combinations of mutation and aliasing, a well-known source of programming pitfalls and security vulnerabilities in both C/C++ and Java. And yet, the type system is expressive enough to type-check many common systems programming idioms. Nonetheless, certain kinds of functionality (*e.g.*, some pointer-based data structures, synchronization abstractions, garbage collection mechanisms) cannot be implemented within the strictures of Rust's type system. Rust provides these abstractions instead via *libraries* whose implementations internally utilize *unsafe features* (*e.g.*, unchecked type casts, array accesses without bounds checks, or accesses of "raw" pointers whose aliasing is untracked by the type system). These libraries are *claimed* to be safe extensions to Rust because they encapsulate their uses of unsafe features in "safe APIs". However, given that the set of such extensions is far from fixed—new and surprising "safe APIs" are being developed all the time—there is a pressing need to understand what property an internally-unsafe library ought to satisfy to be deemed a safe extension to Rust.

[1] Klabnik and Nichols, *The Rust Programming Language* [KN18].

To formalize Rust's "extensible" notion of safety, RustBelt[2] follows prior work on Foundational Proof-Carrying Code[3] by employing a *semantic soundness* proof. First, it defines a *semantic model* of Rust types: a mapping from types `T` to logical predicates on terms $\Phi(e)$, which asserts what it means for the term e to *behave* safely at type `T` (even if internally e uses unsafe features). Then, the RustBelt proof breaks into two main parts:

[2] Jung et al., "RustBelt: Securing the Foundations of the Rust Programming Language" [Jun+18a].

[3] Ahmed et al., "Semantic foundations for typed assembly languages" [Ahm+10].

1. *Safety of libraries that use unsafe features:* For any library that makes use of unsafe features, the implementation of the library is proven to satisfy the semantic model of its API, thus establishing that it is safe for clients to make use of the library. RustBelt proved safety for a number of widely-used Rust libraries, including `Arc`, `Rc`, `Cell`, `RefCell`, `Mutex`, and `RwLock`.

2. *Safety of the λ_{Rust} type system:* The syntactic typing rules of λ_{Rust}

are proven to respect the semantic model, thus establishing that code written in the "safe" fragment of Rust is in fact observably safe—*i.e.*, its behavior is well-defined.

Put together, these imply that if a program $\mathcal{P}$ is well-typed, and its only uses of unsafe features appear within the libraries that have been verified safe (in part 1), then $\mathcal{P}$ is observably safe.

In carrying out their semantic soundness proof for RustBelt, Jung et al. relied on the higher-order concurrent separation logic framework Iris.[4] Separation logic is a good fit for modeling Rust because it is designed around the same notion of *ownership* as Rust's type system, and thus provides built-in support for ownership-based reasoning. One benefit of using Iris is that it was designed to support the derivation of new separation logics with domain-specific reasoning principles. Jung et al. exploited this facility to derive a new logic called the *lifetime logic*, which they used extensively in their proofs in order to reason about Rust's "lifetimes" and "borrowing" mechanisms at a higher level of abstraction.[5] A second benefit of using Iris is that it comes with tactical support for developing *machine-checked* proofs interactively in Coq;[6] this support made it possible for RustBelt to be fully mechanized in Coq.

[4] Jung et al., "Iris from the ground up: A modular foundation for higher-order concurrent separation logic" [Jun+18b].

[5] Klabnik and Nichols, *The Rust Programming Language* [KN18], §4.2, §10.3.

[6] Krebbers et al., "Interactive Proofs in Higher-Order Concurrent Separation Logic" [KTB17]; Krebbers et al., "MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic" [Kre+18].

RustBelt for Relaxed Memory. In the original RustBelt work, Iris was instantiated with a sequentially consistent (SC) semantics for λ_{Rust}. This SC instantiation of Iris (call it "Iris-SC") provides a variety of proof rules that are valid only under SC semantics and not under relaxed-memory semantics. To adapt RustBelt to relaxed memory, we would like to "port" RustBelt so that it is built on top of iRC11, which is sound for the λ_{Rust} + ORC11 semantics, rather than Iris-SC. Following the structure of RustBelt, this porting effort breaks down into two major tasks:

Task 1: Re-prove the safety of the Rust libraries considered by RustBelt, this time verifying their real, relaxed-memory implementations in iRC11.

Task 2: Re-prove the safety of the λ_{Rust} type system, this time relying only on proof rules that are sound in iRC11.

Key Challenge. As it turns out, both of these tasks require us to overcome a technical challenge that is relevant not just to Rust but to relaxed-memory verification in general: namely, that **existing previous work on separation logic does not provide an adequate foundation for reasoning about resource reclamation under relaxed memory**. We will first explain this challenge in the context of Task 1, before briefly describing how it also informs Task 2.

14.1 Task 1: Re-prove the Safety of Rust Libraries under RMC

One of the main motivations for using a "systems programming" language like Rust or C/C++ (as opposed to a garbage-collected language like Java) is to have more precise control over limited resources such as memory. In particular, the Rust programmer can be assured that when an

SC-CInv-acc
$$\frac{\mathcal{N} \subseteq \mathcal{E}}{\boxed{I}^{\gamma,\mathcal{N}} * \heartsuit^{\gamma}_{q} \vdash {}^{\mathcal{E}}\Rightarrow^{\mathcal{E}\setminus\mathcal{N}} \left(\triangleright I * \left(\triangleright I \;{}^{\mathcal{E}\setminus\mathcal{N}}\!\!\Rightarrow\!\!\ast^{\mathcal{E}}\; \heartsuit^{\gamma}_{q}\right)\right)}$$

SC-CInv-tok
$$\heartsuit^{\gamma}_{q+q'} \dashv\vdash \heartsuit^{\gamma}_{q} * \heartsuit^{\gamma}_{q'}$$

SC-CInv-cancel
$$\boxed{I}^{\gamma,\mathcal{N}} * \heartsuit^{\gamma}_{1} \vdash \Rightarrow_{\mathcal{E}} \triangleright I$$

FIGURE 14.1: Key rules for cancelable invariants in Iris-SC

object goes out of scope, the destructor (`drop` method) associated with its type will be invoked and any resources it owns will be reclaimed. Yet the safety of destructors is often quite subtle because objects can contain references to resources that are shared with other objects. For example, objects of type `Arc<T>` are simply aliases to a shared **`struct`** containing an object of type `T` along with a *reference counter*, which keeps track of the current number of active aliases to the object. Consequently, the destructor for `Arc<T>` cannot simply reclaim the shared **`struct`** that it points to: rather, it decrements the shared reference counter, and only if it observes that it was the last remaining alias can it safely reclaim the memory for the reference counter and invoke the destructor for the object of type `T`.

RustBelt showed how to put this subtle kind of resource reclamation on a sound formal footing using Iris-SC's mechanism of *cancelable invariants* (Figure 14.1), a generalization of Gotsman et al.[7] and Hobor et al.[8]'s "storable locks". A cancelable invariant $\boxed{I}^{\gamma,\mathcal{N}}$ is an invariant governing a shared resource I which is only "active" for a certain period of time, after which point it is "cancelled". To access the shared resource during an atomic step of computation (SC-CInv-acc), a thread must prove that the invariant is still active by exhibiting ownership of a fractional *invariant token* $\heartsuit^{\gamma}_{q}$, where q is a fraction in (0,1]. If a thread π can assert ownership of $\heartsuit^{\gamma}_{1}$ (*i.e.*, the "full" γ token), it knows that no other thread can assert that the invariant is active; thus it is safe for π to cancel the invariant and reclaim full ownership of I (SC-CInv-cancel), after which it can free the memory governed by I if it wants to. In RustBelt, cancelable invariants played a crucial role in verifying the safety of destructors such as `Arc`'s.

[7] Gotsman et al., "Local Reasoning for Storable Locks and Threads" [Got+07].

[8] Hobor et al., "Oracle Semantics for Concurrent Separation Logic" [HAN08].

However, adapting cancelable invariants to the relaxed-memory setting turns out to be quite tricky—tricky enough that no existing relaxed-memory separation logic supports them.[9] The main problem arises in how to model the cancelable invariant *tokens*. Under SC, one can simply model invariant tokens as a form of ghost state. But in existing relaxed-memory separation logics, ghost state is *view-agnostic*, meaning that ownership of it can be transferred between threads without the need for any physical synchronization. On the one hand (see §19), view-agnostic ghost state is indispensable for representing *globally consistent* state, such as (in the case of `Arc`) the number of `Arc` aliases currently in existence. On the other hand, if invariant tokens are modeled naively as view-agnostic ghost state, the logic of cancelable invariants becomes unsound! In particular, the access rule SC-CInv-acc is not sound in relaxed memory.

[9] iGPS supports a related notion of "fractional protocol", but it is not nearly as powerful as cancelable invariants and is thus not general enough to account for resource reclamation in Rust.

Our solution is to instead model invariant tokens using a novel notion of *synchronized ghost state*: ghost state that implicitly tracks the

subjective view of the thread that owns it, and that therefore can only be transferred between threads using physical synchronization. **Using synchronized ghost state, iRC11 offers the first general account of resource reclamation in relaxed-memory separation logic.** The resulting interface and model of iRC11 cancelable invariants have been provided in §11.2. In the subsequent chapters in this part, we will demonstrate its effectiveness on a number of real Rust libraries.

14.2 Task 2: Re-prove the Safety of the λ_{Rust} Type System under RMC

In contrast to RustBelt's proofs of safety for libraries, its proof of safety for the λ_{Rust} type system did not rely directly on cancelable invariants or any other SC-specific features of Iris-SC. Rather, as mentioned above, the safety proof for the type system made essential use of a Rust-oriented logic called the *lifetime logic*, which was a domain-specific logic derived within Iris-SC. Thus, if we are able to show that the lifetime logic remains sound under relaxed memory—by instead deriving its soundness in iRC11—then $\text{RB}_{\texttt{rlx}}$ can inherit RustBelt's safety proof for the λ_{Rust} type system without modification!

Synchronized ghost state is the key to making this modular porting strategy possible. Specifically, the lifetime logic is centered around a mechanism called *borrow propositions*, describing resources that are borrowed for the duration of a Rust "lifetime" and that can be reclaimed once the lifetime is over. Borrow propositions are similar in many ways to cancelable invariants, but also more flexible and more complex in terms of the protocols they support for sharing and reclamation of resources. Just as synchronized ghost state enables us to adapt cancelable invariants to relaxed memory, it plays an analogously central role in adapting borrow propositions to relaxed memory as well.

14.3 Contributions of RustBelt Relaxed

RustBelt Relaxed, or $\text{RB}_{\texttt{rlx}}$ for short, is an adaptation of RustBelt to ORC11 and, like its predecessor, is fully mechanized in Coq. We use iRC11 both to re-verify the standard libraries that internally use unsafe features, and to re-prove the soundness of RustBelt's lifetime logic. The safety proof of λ_{Rust}'s type system, by virtue of being built atop the lifetime logic, did not need to be changed at all.

$\text{RB}_{\texttt{rlx}}$ has ported all verifications done in RustBelt, including the following concurrency libraries: `thread::spawn`, `rayon::join`,[10] `Mutex`, `RwLock`, and `Arc`. For the most part, we were able to verify Rust's uses of relaxed-memory operations in these concurrent libraries as is. Only in the implementation of `Arc` did we need to strengthen the consistency level of two memory reads (from relaxed to acquire) in order to make our verification go through. And in one of these cases, our attempt to verify the original (more relaxed) access led us to expose it as the source of a previously undetected data race in the library. Our fix for this race has

[10] whose verification we have seen in §12.3.

since been merged into the Rust codebase.[11] Meanwhile, the verifications of sequential libraries—`Rc`, `Cell`, `RefCell`—remain largely unchanged from RustBelt.

The structure of the remaining chapters in this part is as follows. Please also refer to Figure 1.1 for their dependency graph. Chapter 15 briefly reviews the *lifetime logic*, the core abstraction needed for the original RustBelt's soundness proof the Rust's type system, and Chapter 16 discusses, at a high level, the changes to the interfaces as well as the model of the lifetime logic, so as to be sound on top of iRC11. Chapter 17 discusses how to construct *GPS single-location protocols*[12] using atomic points-to (§10), and how to combine them with cancelable invariants (§11.2) to build *cancelable* GPS protocols. Chapter 18 presents how to combine cancelable GPS protocols with the lifetime logic to verify safety of Rust's `RwLock` library. Chapter 19 presents how to use iRC11 cancelable GPS protocols to verify safety of Rust's `Arc` library. It also discusses a bit of history of how the bug in `Arc` was found.

[11] Jourdan, *Insufficient synchronization in Arc::get_mut* [Jou18].

[12] Turon et al., "GPS: navigating weak memory with ghosts, protocols, and separation" [TVD14]; Kaiser et al., "Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris" [Kai+17].

15

The Lifetime Logic of SC RustBelt

Working with separation logics, one may have become used to being able to transfer ownership of resources from one piece of a program (*e.g.*, one thread) to another. In Chapter 12, we have seen this in action, in relaxed memory even, where ownership of resources are transferred through synchronization between threads, either by message-passing, or by joining, or by matching push and pop operations of a stack. These are all examples of what we call the traditional *direct* style of ownership transfer, where it is clear what is being transferred when.

Although direct ownership transfer is fairly simple, it is unfortunately not sufficient to explain a key feature of the Rust language, namely its *borrowing* mechanism. In this chapter, we review what borrowing is, and why direct ownership transfer does not easily account for it (§15.1). We will then review in §15.2 how RustBelt's original lifetime logic (in SC) comes to the rescue. In Chapter 16, we will discuss how to port the lifetime logic to relaxed memory.

15.1 Borrowing in Rust

The central tenet of Rust is that the most insidious source of safety vulnerabilities in systems programming is the unrestricted combination of mutation and aliasing—when one part of a program mutates some state in such a way that it corrupts the view of other parts of the program that have aliases to (*i.e.*, references to) that state. Consequently, Rust's type system enforces the discipline of *aliasing XOR mutability* (AXM, for short): a value of type `T` may *either* have multiple aliases (called *shared references*), of type `&T`, *or* it may be mutated via a unique, *mutable reference*, of type `&mut T`, but it may not be both aliased and mutable at the same time.

To create a mutable reference to an object `o : T` in Rust, one *borrows* `o` for the duration of some *lifetime* `'a`, with the result being a reference value `r` of type `&'a mut T`. Borrowing causes the ownership of `o` to be *split in time*: while the lifetime `'a` is alive, the borrower controls the object and can use `r` to mutate it; but once `'a` is dead, the original owner of `o` can reclaim ownership of it. The reclamation that occurs once the lifetime `'a` is over is essentially a form of ownership transfer from the borrower to the original owner of `o`. And the natural question that arises when proving the safety of Rust is: how do we know that this reclamation

is sound?

One might think that there is an obvious way of modeling this reclamation using direct ownership transfer: when the lifetime `'a` is over, the borrower just needs to hand ownership of the borrowed reference back to the original owner. Unfortunately, it is not always that straightforward.

Example 15.1 (Indirect Reclamation of Resources)**.** Consider the following example taken from the RustBelt paper.[1] The example makes use of the function `index_mut` from the `Vec` library.

[1] Jung et al., "RustBelt: Securing the Foundations of the Rust Programming Language" [Jun+18a].

```
1 let mut v = vec![21, 57];
2 { let mut head = v.index_mut(0);
3   // head : &'a mut i32
4   // Cannot access v: v.push(42) rejected
5   *head = 23; }                          Lifetime 'a
6 v.push(42);
```

In this example, we start with a vector `r` of two elements `21` and `57`. Then, in lines 2-5, using `index_mut`, we get a mutable deep pointer `head` into the head element of `r` and update its contents to `23`. Such a deep aliasing into the vector's internal is dangerous, because if the pointer outlives the internal storage, the pointer will be a dangling one and dereferencing it is undefined behavior. This is why Rust will reject `v.push(42)` in line 4, as `push` may reallocate the internal storage.

But what does this have to do with indirect ownership transfer? What is happening is that the code block in lines 2-5 *borrows* the whole vector `r` from its parent block. Borrowing the whole vector is due to `index_mut`, whose type is

```
fn(&'a mut Vec<i32>, usize) -> &'a mut i32
```

This function takes a mutable reference `r` to an integer vector, along with an index `n`, and returns an interior mutable reference `e` to the `n`-th element of the vector. Crucially, thanks to Rust's substructural type system, the caller of this function *gives up* ownership of the argument `r` in exchange for the result `e`. The surrendering of `r` is quite important here because otherwise `r` could be used to subsequently mutate the object in a way that would invalidate the interior pointer `e`. On the other hand, the lifetime `'a` ensures that `e` can only be accessed during the lifetime at which the original `r` was accessible.

In our example, `index_mut` borrows the whole vector `v` and returns a single mutable reference into the first index as `head`, restricted to the duration of the lifetime `'a`. Firstly, the inner block does not need to care about transferring the vector `v` back to the parent block once it is done. The inner block simply knows that it will finish using `v` before the lifetime `'a` ends—the moment after which the vector will be given back to the parent block indirectly.

Secondly, the inner block effectively declares that it only cares about one index in the vector and only wants to borrow that single index. But the life of the index is tied to the life of the whole vector, therefore, to prevent dangerous aliasing, it is necessary to borrow the whole vector with the same lifetime `'a`. In other words, even when the borrower

forgets about the borrow `&'a mut Vec<i32>`, it is still indirectly tied to the borrow `&'a mut i32`, which, in our case, is the `head` pointer. When the lifetime `'a` ends, the ownership of not just the `head` index but the whole vector will be indirectly transferred back to the parent block. Regaining ownership of the whole vector, the parent block can `push` again in line 6.

In summary, in addition to direct ownership transfer, Rust also employs an indirect transfer scheme which (1) uses lifetimes to indirectly specify *when* the return transfer happens: all borrowed resources will be returned at the end of the lifetime which does not need to be directly agreed on up front; and (2) uses borrows and their relations to indirectly specify *what* needs to be returned without bookkeeping every bit of borrowed resources. Translating this indirect transfer scheme to separation logics is no easy task, but have been accomplished by Jung et al. [Jun+18a] with the *lifetime logic*.

15.2 The Lifetime Logic Primer, in SC

At a high level, the idea of the lifetime logic is to formalize the intuition mentioned above: borrowing an object `o` for a lifetime `'a` splits ownership of `o` *in time*, between a "borrow" assertion, which the borrower can use to access `o` while `'a` is alive, and an "inheritance" assertion, which the original owner can use to reclaim ownership of `o` once `'a` is dead. Although "splitting ownership in time" is not a standard notion in separation logic, the Iris framework is designed to enable one to derive such non-standard notions of separation and embed them in the separating conjunction connective, and that is precisely what Jung et al. did.

The lifetime logic introduces several *abstract predicates* representing a variety of capabilities and permissions related to lifetimes and borrowing, together with axioms (proven sound in Iris-SC) for manipulating them, as shown in Figure 15.1. Let us begin with an overview of the new predicates:

- The *full borrow* $\&^{\kappa}_{\mathbf{full}}\, P$ asserts *temporary* ownership of resource P, *while the lifetime* κ *is alive*. It provides a direct means of modeling the semantics of Rust's mutable reference types.
- The timeless *lifetime token* $[\kappa]_q$ serves as a witness that the lifetime κ is still alive. Here, q is a fraction in $(0, 1]$. If $q = 1$, we say that this is the *full token* for κ. The use of fractions allows one to share the knowledge that a lifetime is alive with multiple parties.
- The *killer permission* $\mathsf{Kill}(\kappa)$ is a unique permission needed to kill the lifetime κ.
- The timeless and persistent *dead token* $[\dagger\kappa]$ is used to witness the knowledge that lifetime κ is dead.
- The *inheritance* $\mathsf{Inh}(\kappa, P)$ asserts the right to reclaim the ownership of borrowed resource P once κ is dead.

$\mathsf{timeless}([\kappa]_q)$
$\mathsf{timeless}([\dagger\kappa])$
$\mathsf{persistent}([\dagger\kappa])$

LFTL-BEGIN
$\mathsf{True} \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \exists\kappa.\, [\kappa]_1 * \mathsf{Kill}(\kappa)$

LFTL-KILL
$[\kappa]_1 * \mathsf{Kill}(\kappa) \Rrightarrow^{\varnothing}_{\mathcal{N}_{\mathrm{lft}}} [\dagger\kappa]$

LFTL-TOK-FRAC
$[\kappa]_{q+q'} \dashv\vdash [\kappa]_q * [\kappa]_{q'}$

LFTL-TOK-NOT-DEAD
$[\kappa]_q * [\dagger\kappa] \vdash \mathsf{False}$

LFTL-FULL-BOR
$\triangleright P \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa}_{\mathbf{full}} P * \mathsf{Inh}(\kappa, P)$

LFTL-FULL-INH
$[\dagger\kappa] * \mathsf{Inh}(\kappa, P) \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \triangleright P$

LFTL-FULL-ACC
$\&^{\kappa}_{\mathbf{full}} P * [\kappa]_q \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \triangleright P * \mathsf{Ret}(\kappa, P, q)$

LFTL-FULL-RET
$\triangleright P * \mathsf{Ret}(\kappa, P, q) \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa}_{\mathbf{full}} P * [\kappa]_q$

LFTL-FULL-SEP
$\&^{\kappa}_{\mathbf{full}}(P * Q) \Lrightarrow_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa}_{\mathbf{full}} P * \&^{\kappa}_{\mathbf{full}} Q$

FIGURE 15.1: Selected rules of SC RustBelt's lifetime logic

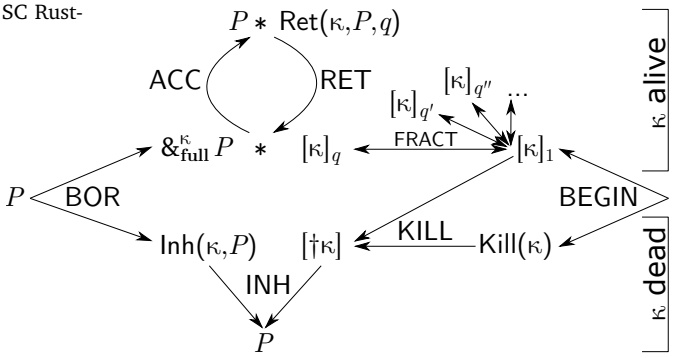


FIGURE 15.2: The life cycle of borrows and lifetimes

- The *return policy* $\mathsf{Ret}(\kappa, P, q)$ is used as part of the protocol for accessing the contents of a full borrow.

We briefly explain the rules in Figure 15.1 with the help of Figure 15.2, which depicts the life cycle of a lifetime and a full borrow. We start from the right of Figure 15.2, where we create a new lifetime using LFTL-BEGIN (BEGIN in Figure 15.2). This yields the full token $[\kappa]_1$ for a new lifetime κ, as well as the corresponding $\mathsf{Kill}(\kappa)$ permission. Lifetime tokens are *fractional* (LFTL-TOK-FRAC, FRACT), so that they can be split into (and joined back from) smaller pieces which enable multiple threads to simultaneously witness that κ is still alive.

Next, on the left of Figure 15.2, we see the "flagship" rule of the lifetime logic: given ownership of any assertion P, and any lifetime κ, we can use the *borrowing rule* LFTL-FULL-BOR (BOR in Figure 15.2) to create a borrow of P for κ. The rule splits ownership of P in time between two separately ownable assertions: (1) a full borrow $\&^{\kappa}_{\mathbf{full}} P$ that represents ownership of P while κ is alive; and (2) an inheritance $\mathsf{Inh}(\kappa, P)$ that can be used to reclaim P after κ dies. Intuitively, this rule directly models what happens when an object is borrowed in Rust, with the full borrow then being given to the borrower and the inheritance given to the object's original owner.

A thread owning *both* the full borrow $\&^{\kappa}_{\mathbf{full}} P$ *and* a token $[\kappa]_q$ (proving κ is alive) can trade them to obtain P using the *accessing rule* LFTL-FULL-ACC (ACC in Figure 15.2). As part of the trade, the thread is also given the return policy $\mathsf{Ret}(\kappa, P, q)$. Once the thread is done using P, it trades

P and $\mathsf{Ret}(\kappa, P, q)$ to get back $\&^{\kappa}_{\mathbf{full}} P$ and $[\kappa]_q$ (LftL-full-ret, RET in Figure 15.2).

Once all accesses to borrows at lifetime κ are done, we can recollect the full token $[\kappa]_1$ and use the killer permission $\mathsf{Kill}(\kappa)$ with LftL-kill (KILL) to end the lifetime. This yields the dead token $[\dagger\kappa]$. Since κ is now dead, the content P in $\&^{\kappa}_{\mathbf{full}} P$ cannot be accessed any more and can thus be reclaimed. Anyone owning $[\dagger\kappa]$ and the inheritance $\mathsf{Inh}(\kappa, P)$ can use LftL-full-inh (INH in Figure 15.2) to reclaim P. LftL-tok-not-dead says that a fraction $[\kappa]_q$ of the lifetime token indeed proves that the lifetime is alive: it is disjoint from the dead token $[\dagger\kappa]$.

Note that LftL-kill uses a wand step viewshift (Notation 7.8), *i.e.*, its user needs an actually step to discharge the later in the viewshift. As such, we use LftL-kill around a so-called ghost instruction **`endlft`** (Notation 4.3, Figure 4.2). In fact, λ_{Rust} code will be populated with ghost instructions **`newlft`** and **`endlft`** to mark the beginning and the end of some lifetime. One can image that these ghost instructions are inserted by Rust's type inference, or explicitly by programmers.

In short, borrows are tied to lifetimes' life cycles. After the lifetime ended, the inheritor (one who owns the inheritance) can reclaim the borrowed resources. Note that the inheritance does not need to be used immediately after the lifetime dies. The inheritor may only receive the dead token a long time after the lifetime dies, and even then it does not need to inherit immediately. This supports a part of Rust's indirect transfer: participants do not need to agree up front on when the return transfer happens.

Although not depicted in Figure 15.2, another crucial rule of the lifetime logic is LftL-full-sep, which lets one go back and forth between a borrow of $P * Q$ and separate borrows of P and Q. This rule is essential in verifying the soundness of Rust functions like `index_mut` (§15.1) that split a reference to an object into references to its sub-components.

Example 15.2 (MP in the Lifetime Logic)**.** Let us now quickly demonstrate how the lifetime logic can support a somewhat different verification of the MP example, albeit with the SC semantics, in Figure 15.3. Here, instead of transferring the location ℓ_x from thread 1 to thread 2 directly, we transfer a lifetime token, which thread 2 then uses to reclaim ownership of ℓ_x. The use of the lifetime logic here is clearly overkill since direct ownership transfer of ℓ_x already suffices, but it will nonetheless give the reader a concrete feel for the lifetime logic in action.

In Figure 15.3(a), we start by creating a lifetime κ (LftL-begin). Then, with the ownership of $\ell_x \mapsto 0$, we create a borrow $\&^{\kappa}_{\mathbf{full}}(\ell_x \mapsto \_)$ using LftL-full-bor. We assume a *send-receive* protocol SendRecv for ℓ_y that satisfies the rules SendRecv-Create, SC-Send, and SC-Recv (in the top of Figure 15.3). We instantiate this protocol with $[\kappa]_{1/2}$ as the content to be sent. Again, we could have instantiated the protocol with $\ell_x \mapsto \_$ and be done with it. Instead, here we want to demonstrate the use of borrows.

When spawning two threads, we give a half token $[\kappa]_{1/2}$, the borrow $\&^{\kappa}_{\mathbf{full}}(\ell_x \mapsto \_)$, and Send to the thread π, and give the other half of the κ token, the killer, the inheritance, and Recv to thread ρ.

SENDRECV-CREATE
$\ell_y \mapsto 0 \Rrightarrow \divideontimes_{\mathcal{E}} \mathsf{Send}_{\ell_y}(P) * \mathsf{Recv}_{\ell_y}(P)$

SC-SEND
$\{\mathsf{Send}_{\ell_y}(P) * P\}\, \ell_y :=_{\mathbf{sc}} 1\, \{\mathsf{True}\}$

SC-RECV
$\{\mathsf{Recv}_{\ell_y}(P)\}\, {}^{*\mathbf{sc}}\ell_y\, \{v.\, v = 0 \lor P\}$

$\{\ell_x \mapsto 0 * \ell_y \mapsto 0\}$ **newlft**; $\{[\kappa]_1 * \mathsf{Kill}(\kappa) * \ell_x \mapsto 0 * \ell_y \mapsto 0\}$ // LFTL-BEGIN
$\{[\kappa]_1 * \mathsf{Kill}(\kappa) * \&^{\kappa}_{\mathbf{full}}(\ell_x \mapsto \_) * \mathsf{Inh}(\kappa, \ell_x \mapsto \_) * \ell_y \mapsto 0\}$ // LFTL-FULL-BOR
$\left\{[\kappa]_1 * \mathsf{Kill}(\kappa) * \&^{\kappa}_{\mathbf{full}}(\ell_x \mapsto \_) * \mathsf{Inh}(\kappa, \ell_x \mapsto \_) * \mathsf{Send}_{\ell_y}([\kappa]_{1/2}) * \mathsf{Recv}_{\ell_y}([\kappa]_{1/2})\right\}$ // SENDRECV-CREATE

((a)) Proof of initialization.

$\left\{[\kappa]_{1/2} * \&^{\kappa}_{\mathbf{full}}(\ell_x \mapsto \_) * \mathsf{Send}_{\ell_y}([\kappa]_{1/2})\right\}$
$\left\{\ell_x \mapsto \_ * \mathsf{Ret}(\kappa, \ell_x \mapsto -, 1/2) * \mathsf{Send}_{\ell_y}([\kappa]_{1/2})\right\}$
// LFTL-FULL-ACC
$\ell_x := 42;$
$\left\{\ell_x \mapsto \_ * \mathsf{Ret}(\kappa, \ell_x \mapsto -, 1/2) * \mathsf{Send}_{\ell_y}([\kappa]_{1/2})\right\}$
// NA-WRITE
$\left\{[\kappa]_{1/2} * \mathsf{Send}_{\ell_y}([\kappa]_{1/2})\right\}$ // LFTL-FULL-ACC
$\ell_y :=_{\mathbf{sc}} 1;$
$\{\mathsf{True}\}$ // SC-SEND

((b)) Proof of thread π.

$\left\{[\kappa]_{1/2} * \mathsf{Kill}(\kappa) * \mathsf{Inh}(\kappa, \ell_x \mapsto \_) * \mathsf{Recv}_{\ell_y}([\kappa]_{1/2})\right\}$
if $({}^{*\mathbf{sc}}\ell_y \mathrel{!}= 0)$
then case
  $\left\{[\kappa]_{1/2} * \mathsf{Kill}(\kappa) * \mathsf{Inh}(\kappa, \ell_x \mapsto \_) * [\kappa]_{1/2}\right\}$
  // SC-RECV
  $\{[\kappa]_1 * \mathsf{Kill}(\kappa) * \mathsf{Inh}(\kappa, \ell_x \mapsto \_)\}$
  // LFTL-TOK-FRAC
  endlft; $\{[\dagger\kappa] * \mathsf{Inh}(\kappa, \ell_x \mapsto \_)\}$
  // LFTL-KILL
  $\{\ell_x \mapsto \_\}$ // LFTL-FULL-INH
  $\ell_x := 57;\ \{\ell_x \mapsto 57\}$ // NA-WRITE

((c)) Proof of thread ρ.

FIGURE 15.3: MP verified with the lifetime logic in Iris-SC.

In Figure 15.3(b), thread π trades the token and the borrow to access $\ell_x \mapsto \_$ with LFTL-FULL-ACC and writes to ℓ_x. After that, with LFTL-FULL-RET, thread π trades the return policy and $\ell_x \mapsto \_$ to get back the token and the borrow. Finally, thread π writes to ℓ_y and sends the token $[\kappa]_{1/2}$ to thread ρ.

In Figure 15.3(c), thread ρ uses Recv to get back the full token. Owning $\mathsf{Kill}(\kappa)$, it ends the lifetime and earns the dead token $[\dagger\kappa]$ (LFTL-KILL). Combining that with the inheritance, thread ρ reclaims the ownership of $\ell_x \mapsto \_$ (LFTL-FULL-INH) and can safely write (non-atomically) to ℓ_x. □

Remark 15.3 (Safety of Inheritance)**.** Let us note an important safety property of the lifetime logic: *the inheritance of a borrow can only be used after all accesses to the borrowed content have finished*. The key to ensuring this is that, during an access of the borrow $\&^{\kappa}_{\mathbf{full}} P$ via the accessing rule LFTL-FULL-ACC, the lifetime token $[\kappa]_q$ and the borrow assertion are "kept" by the return policy and are only returned in exchange for the borrowed content P. By withholding $[\kappa]_q$ and only returning it after the access finishes, the rule ensures that no party can have the full token $[\kappa]_1$ needed to kill κ while others are still accessing borrows associated with κ. Consequently, the inheritance can only be used *after* all accesses have finished.

This safety property is no difference from the safety property CANCEL-SAFE (Property 11.2) of cancelable invariants, both in Iris-SC (Figure 14.1) and in iRC11 (§11.2). Indeed, we can see that borrows and cancelable invariants share many similarities in their interfaces: a fractional token is needed to access some protected resources, and a full fraction of the

token is sufficient to reclaim those protected resources. In this aspect, we can in fact see borrows as a generalization of cancelable invariants, where the generalization is in the flexibility of reclamation, which is now tied to lifetimes. This allows multiple borrows to be managed by a single lifetime, and while the borrows become "canceled" when the lifetime is killed, their reclamation can be done much later than that.

However, when looking at the aspect of concurrent accesses, cancelable invariants are more accommodating than full borrows. Full borrows only allow *sequential* accesses, by the fact that the full borrow assertion $\&^{\kappa}_{\mathbf{full}} P$ is unique and cannot be shared. As such, if multiple threads want to access a full borrow, they need extra synchronization to pass on the ownership of the full borrow assertion. (In Example 15.2, only the thread π accesses the full borrow, while the thread ρ simply kills the lifetime and thus the borrow.) Meanwhile, cancelable invariants support multiple threads accessing shared resources atomically.

The SC lifetime logic does support other forms of borrows, including atomic borrows which do allow concurrent atomic accesses. Atomic borrows then are a true generalization of cancelable invariants. In Chapter 16, we will see that porting atomic borrows to RMC faces the same challenge as porting cancelable invariants to RMC—we need to maintain CANCEL-SAFE in the presence of accesses that do not establish synchronization. Fortunately, the solution is also the same, and we needed to only change the interface of the access rule for atomic borrows. Other kinds of borrows, including full borrows, maintain the same interfaces when being ported from SC to RMC. Naturally, we needed to update the model of the lifetime logic so as to be sound in λ_{Rust} + ORC11.

Note 15.4 (Rules with Viewshifts). Finally, let us note that the rules given in Figure 15.1 have been streamlined for better representation, with the additional concepts of the killer permission $\mathsf{Kill}(\kappa)$, the inheritance $\mathsf{Inh}(\kappa, P)$, and the return policy $\mathsf{Ret}(\kappa, P, q)$. In practice, these concepts are actually just wand viewshifts, and we simply have them bundled in the rules. We will see the rules in Figure 16.1 (§16.1).

Furthermore, note that the namespace $\mathcal{N}_{\mathrm{lft}}$ is a global, public namespace that is needed to establish the invariants for the internal model of the lifetime logic.

CHAPTER SUMMARY. In this chapter, we have reviewed borrows in Rust and how RustBelt accounts for them by developing the lifetime logic, which provides separation logic principles for borrows. We have seen the rules for managing lifetimes and full borrows. In the next chapter, we will see a more complete interface of the lifetime logic in RMC, and how to adapt the logic's model on top of iRC11.

16

Lifetime Logic Meets Relaxed Memory

In this chapter, we present a more complete interface of the lifetime logic *after* being ported from Iris-SC to iRC11. Fortunately, almost all proof rules of the lifetime logic are sound in RMC. The only change in the proof rules is in LftL-at-acc—the access rule for *atomic borrows*—which allows access to the borrowed resource only under the view-join modality. This is very much similar to how the access rule of cancelable invariants (CInv-acc, §11.2.1) needed to be changed in iRC11.

In this adaptation, the models for other borrows as well as the model of lifetime tokens are extended with instances of *synchronized ghost state* (Concept 11.5) to account for synchronization that always exists but needs to be witnessed explicitly under RMC. Despite these changes, the borrows enjoy the same proof rules as in SC. In particular, *the SC rules in Figure 15.1 for lifetimes and full borrows still hold in* iRC11.

We discuss more rules for lifetimes and full borrows in §16.1, and the interface of other borrow forms in §16.2. We simultaneously review these constructs (which are already developed in the original lifetime logic)[1] and describe the changes due to adaptation as they appear. Readers interested in more details of the origin lifetime logic are recommended to refer to the original paper and its technical appendix. In §16.3, we will discuss how to adapt the lifetime logic's model on top of iRC11.

[1] Jung et al., "RustBelt: Securing the Foundations of the Rust Programming Language" [Jun+18a].

16.1 More Rules for the Lifetime Logic

In addition to the rules in Figure 15.1, the relaxed lifetime logic built atop iRC11 also admits stronger rules, some of which are given in Figure 16.1. The rule LftL-begin-bd bundles LftL-begin and LftL-kill together, and similarly LftL-full-bor-bd bundles LftL-full-bor and LftL-full-inh, and LftL-full-acc-bd bundles LftL-full-acc and LftL-full-ret. The rest of Figure 16.1 presents stronger rules that are originally from SC lifetime logic, but now adapted to relaxed memory.

16.1.1 *Lifetime Tokens Track Observations*

LftL-tok-obj-split strengthens LftL-tok-frac by allowing splitting into two bits where one bit can be objective and thus can be put in invariant. This mirror the same rule CInv-tok-obj-split (§11.2.1) for iRC11 cancelable invariant tokens $\heartsuit^{\gamma}_{q}$. Intuitively, both types of tokens are now

synchronized ghost state, because they are witnesses not only for the liveness of a lifetime or a cancelable invariant, but also for what has happened during accesses[2] that the tokens have been used for. As such, the tokens are view-dependency, so sending them away (by *e.g.*, putting them in invariants) would require some synchronization. LftL-tok-obj-split helps in this regard: since the part $[\kappa]_q$ of $[\kappa]_{q+q'}$ is sufficient to bare witness for what $[\kappa]_{q+q'}$ itself has observed, the other part $[\kappa]_{q'}$ can start afresh with zero observation, *i.e.*, be placed under the objective modality (Definition 8.10), so that it can be sent to other threads without extra synchronization. The same situation applies for CInv-tok-obj-split and cancelable invariant tokens.

[2] of a borrow for a lifetime token, or of a cancelable invariant for an invariant token

The rule LftL-tok-not-dead-subj strengthens LftL-tok-not-dead: the fact $[\dagger\kappa]$ that the lifetime is dead is useful globally without the need of synchronization. That is, it is still sufficient if we have $[\dagger\kappa]$ under a subjective modality (§8.6). In fact, it is often the case that we only need $\langle \mathsf{subj} \rangle\, [\dagger\kappa]$ (see also *faking*, below). We would need $[\dagger\kappa]$ locally in inheritance to ensure the safety of inheritance (see LftL-full-bor-bd or LftL-full-inh, and Remark 15.3).

16.1.2 *Faking*

LftL-bor-fake witnesses the fact that, once a lifetime has ended, the borrows tied to it have also ended and the owners of the inheritances can freely reclaim the borrowed resources *without* owning the borrow assertions $\&^{\kappa}_{\mathbf{full}}\, P$. In other words, the borrow assertions $\&^{\kappa}_{\mathbf{full}}\, P$ become meaningless then. LftL-bor-fake thus allows us to create fake borrows if we know that the associated lifetime is dead, even only at the view the lifetime killer, without synchronization (*i.e.*, with only $\langle \mathsf{subj} \rangle\, [\dagger\kappa]$).

16.1.3 *Lifetime Inclusion*

The Rust's type system involves subtyping rules that rely on an inclusion between lifetime. Intuitively, a lifetime κ is included in a lifetime κ' if κ is *shorter* than κ'—in other words, κ' *outlives* κ. The subtyping rule with respect to lifetime (T-bor-lft in [Jun+18a, §3.3]) then allows a (type-level) borrow associated with κ to be a subtype of the borrow associated with the longer κ'.

In RustBelt, the (reflexive, transitive) lifetime inclusion relation $\sqsubseteq$ gives rise to a meet semi-lattice for lifetimes, where the meet composition ($\sqcap$) is commutative and associative. In other words, lifetimes κ form a *partial commutative monoid*. $\sqcap$ is also called the *intersection* of lifetimes. Intersection is useful to create fresh sub-lifetimes: the meet composition respects lifetime inclusion, following LftL-incl-inter and LftL-incl-glb. LftL-full-shorten says that a borrow with a lifetime κ' can be turned into a borrow with a shorter lifetime κ'. This lifetime-logic-level rule is the model for the type-level subtyping rule T-bor-lft.

LftL-tok-inter and LftL-dead-inter show the interactions between lifetime intersection and liveness. To know that the intersected lifetime $\kappa \sqcap \kappa'$ is alive, we need to know that both component lifetimes are alive. Reversely, the intersected lifetime is dead, then either κ or κ is dead. This

Bundled Rules.

LftL-begin-bd
$$\mathsf{True} \Rrightarrow_{\mathcal{E}} \exists \kappa.\, [\kappa]_1 * \Box([\kappa]_1 \mathrel{\text{=}\!\!\Rightarrow}^{\varnothing}_{\mathcal{N}_{\mathrm{lft}}} [\dagger\kappa])$$

LftL-full-bor-bd
$$\triangleright P \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa}_{\mathbf{full}} P * ([\dagger\kappa] \Rrightarrow\!\!\!\!\ast_{\mathcal{N}_{\mathrm{lft}}} \triangleright P)$$

LftL-full-acc-bd
$$\&^{\kappa}_{\mathbf{full}} P * [\kappa]_q \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \triangleright P * (\triangleright P \Rrightarrow\!\!\!\!\ast_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa}_{\mathbf{full}} P * [\kappa]_q)$$

Lifetime Liveness in Relaxed Memory.

LftL-tok-obj-split
$$[\kappa]_{q+q'} \vdash [\kappa]_q * \langle \mathsf{obj} \rangle\, [\kappa]_{q'}$$

LftL-tok-not-dead-subj
$$[\kappa]_q * \langle \mathsf{subj} \rangle\, [\dagger\kappa] \vdash \mathsf{False}$$

LftL-bor-fake
$$\langle \mathsf{subj} \rangle\, [\dagger\kappa] \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa}_{\mathbf{full}} P$$

Lifetime Inclusion.

$\mathsf{timeless}(\kappa \sqsubseteq \kappa')$
$\mathsf{persistent}(\kappa \sqsubseteq \kappa')$

LftL-incl-inter
$$\kappa \sqcap \kappa' \sqsubseteq \kappa$$

LftL-incl-glb
$$\frac{\kappa \sqsubseteq \kappa' \qquad \kappa \sqsubseteq \kappa''}{\kappa \sqsubseteq \kappa' \sqcap \kappa''}$$

LftL-full-shorten
$$\kappa' \sqsubseteq \kappa \vdash \&^{\kappa}_{\mathbf{full}} P \Rightarrow \&^{\kappa'}_{\mathbf{full}} P$$

LftL-tok-inter
$$[\kappa]_q * [\kappa']_{q'} \vdash \exists q''.\, [\kappa \sqcap \kappa']_{q''} * \left([\kappa \sqcap \kappa']_{q''} \mathrel{-\!\!*} [\kappa]_q * [\kappa']_q\right)$$

LftL-dead-inter
$$[\dagger\kappa \sqcap \kappa'] \dashv\vdash [\dagger\kappa] \lor [\dagger\kappa']$$

LftL-unit-static
$$\kappa \sqcap \varepsilon = \kappa$$

LftL-incl-static
$$\kappa \sqsubseteq \varepsilon$$

LftL-static-not-dead
$$[\dagger\varepsilon] \vdash \mathsf{False}$$

LftL-tok-static
$$\vdash [\varepsilon]_q$$

Reborrowing.

LftL-reborrow
$$\kappa' \sqsubseteq \kappa \vdash \&^{\kappa}_{\mathbf{full}} P \Rrightarrow\!\!\!\!\ast_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa'}_{\mathbf{full}} P * \left([\dagger\kappa'] \Rrightarrow\!\!\!\!\ast_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa}_{\mathbf{full}} P\right)$$

LftL-bor-unnest
$$\&^{\kappa'}_{\mathbf{full}}(\&^{\kappa}_{\mathbf{full}} P) \mathrel{\text{=}\!\!\Rightarrow}_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa \sqcap \kappa'}_{\mathbf{full}} P$$

Stronger Access Rules.

LftL-full-acc-strong
$$\&^{\kappa}_{\mathbf{full}} P * [\kappa]_q \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \exists \kappa'.\, \kappa \sqsubseteq \kappa' * \triangleright P * \left(\forall Q.\, \triangleright(\triangleright Q * \langle \mathsf{subj} \rangle\, [\dagger\kappa'] \Rrightarrow\!\!\!\!\ast_{\varnothing} \triangleright P) * \triangleright Q \Rrightarrow\!\!\!\!\ast_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa'}_{\mathbf{full}} Q * [\kappa]_q\right)$$

LftL-full-acc-atomic-strong
$$\&^{\kappa}_{\mathbf{full}} P \;\; {}^{\mathcal{N}_{\mathrm{lft}}}\!\Rrightarrow^{\varnothing} \;\; \vee \begin{cases} \exists P', \kappa'.\, \kappa \sqsubseteq \kappa' * \triangleright(\lceil P' \rceil \wedge P) * \left(\begin{array}{l} \forall Q.\, \triangleright(\triangleright Q * \langle \mathsf{subj} \rangle\, [\dagger\kappa'] \Rrightarrow\!\!\!\!\ast_{\varnothing} \triangleright P) * \triangleright(\lceil P' \rceil \wedge Q) \\ \qquad {}^{\varnothing}\!\Rrightarrow\!\!\!\!\ast^{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa'}_{\mathbf{full}} Q \end{array}\right) \\ \exists \kappa'.\, \kappa \sqsubseteq \kappa' * \langle \mathsf{subj} \rangle\, [\dagger\kappa'] * {}^{\varnothing}\!\Rrightarrow^{\mathcal{N}_{\mathrm{lft}}} \mathsf{True} \end{cases}$$

FIGURE 16.1: More selected rules for lifetimes and full borrows, ported to λ_{Rust} + ORC11

implies that if the shorter lifetime κ ($\kappa \sqsubseteq \kappa'$) is alive, the κ' is also alive, and if κ' is dead, then κ must also be dead.

The semi-lattice has a unit ε (LftL-unit-static), which is used to model the static lifetime (`'static` in Rust). Intuitively, the static lifetime is the global lifetime that includes all lifetimes (LftL-incl-static) and is never dead (LftL-static-not-dead).[3] Consequently, we can always acquire a fraction of the static lifetime token (LftL-tok-static).

[3] Intuitively, if the static lifetime is already dead, then the program has ended.

Definition 16.1 (Dynamic Lifetime Inclusion)**.** Lifetime inclusion in RustBelt is in fact defined *semantically* or *dynamically*, using a token trading scheme. A lifetime κ is included in κ' if, given a fraction of the token for κ, we can produce some fraction of the token for κ'.

$$\kappa \sqsubseteq \kappa' ::= \Box \forall q.\, [\kappa]_q \Rrightarrow \Rrightarrow_{\mathcal{N}_{\text{lft}}} \left(\exists q'.\, [\kappa']_{q'} * ([\kappa']_{q'} \Rrightarrow \Rrightarrow_{\mathcal{N}_{\text{lft}}} [\kappa]_q)\right)$$

16.1.4 *Reborrowing*

The rule LftL-reborrow strengthens LftL-full-shorten: it lets us *reborrow* a $\&^{\kappa}_{\mathbf{full}} P$ into a borrow $\&^{\kappa}_{\mathbf{full}}{}' P$ where $\kappa' \sqsubseteq \kappa$. And when the shorter lifetime κ' ends, we get our original full borrow back. As such, while LftL-full-shorten can also be seen as a reborrow, it simply forgets the difference between κ' and κ. On the other hand, LftL-reborrow gives us an inheritance, effectively allowing us to regain and use the original borrow when κ' is already dead but κ is still alive. Note that the inheritance requires the observation of κ''s end locally, *i.e.*, not under a subjective ($\langle \text{subj} \rangle$) modality. LftL-reborrow justifies the RustBelt's type system rule C-reborrow (in [Jun+18a, §3.3]).

The related rule LftL-bor-unnest allows us to turn a full borrow of a full borrow ($\&^{\kappa'}_{\mathbf{full}} \&^{\kappa}_{\mathbf{full}} P$) into a full borrow of the intersected lifetime $\&^{\kappa' \sqcap \kappa}_{\mathbf{full}}$. The catch is that we need a wand step viewshift (Notation 7.8) to strip off an extra later modality that appears between the two borrows.

16.1.5 *Stronger Access Rules*

The rule LftL-full-acc-strong generalizes LftL-full-acc-bd. It allows us to close an access by giving back not just the original resource $\triangleright P$, but some $\triangleright Q$ if we can show that $\triangleright Q$ entails $\triangleright P$ through a view shift. That view shift is only needed when the lifetime ends, *i.e.*, we can assume $\langle \text{subj} \rangle\, [\dagger\kappa]$ when proving that Q entails P. In exchange, we get back a full borrow $\&^{\kappa'}_{\mathbf{full}} Q$ of Q. Intuitively, the rule allows us to turn a full borrow $\&^{\kappa}_{\mathbf{full}} P$ into $\&^{\kappa'}_{\mathbf{full}} Q$ with the access, as long as we can still guarantee that the inheritance will get back the original resource P, and hence the viewshift is only needed at inheritance, once the lifetime κ' is dead ($[\dagger\kappa']$). If LftL-full-acc-strong is invoked multiple times, then the borrow's internal invariant will collect as many such viewshifts, and will apply all of them together at the inheritance to reclaim the original resource with which the very first borrow was created.

Furthermore, the rule exposes the fact that $\kappa \sqsubseteq \kappa'$, which comes from a part of RustBelt's model for the lifetime logic's borrows. That is, the model of a full borrow $\&^{\kappa}_{\mathbf{full}} P$ actually ties the resource P to a bigger lifetime κ', and hence the borrow assertion is downward-closed

with respect to lifetime inclusion, and renders the proofs of rules like LftL-full-shorten easy. This technique is employed quite frequently in the lifetime logic's model.

Finally, the rule LftL-full-acc-atomic-strong provides a way to access a full borrow *without* having a proof that the lifetime is still ongoing. As such, with the access, we will find a disjunction, corresponding to the two cases where the lifetime κ is still alive or is already dead. If it is already dead, we get that fact subjectively, *i.e.*, $\langle \mathsf{subj} \rangle [\dagger \kappa']$. If the lifetime is still alive, we get access to the resource P, and we can close the access in the same way as that of LftL-full-acc-strong. Since we do not need to provide a lifetime token $[\kappa]_q$, the access cannot be non-atomic, because the lifetime κ may be killed during such a non-atomic access. The atomicity of the access is enforced by the mask-changing viewshifts.

Note than in addition to acquiring P at opening and returning Q at closing, we also know and then have to show—at opening and closing of the access—that some resource $P' : \mathsf{iProp}$ holds simultaneously (*i.e.*, with a classical conjunction) with P or Q. The "embed" modality $\lceil \cdot \rceil$ embeds iProp propositions into vProp. This extra requirement is also due to potential relaxed memory effects. Since we do not have a lifetime token at hand to witness to possible changes from P to Q, we instead require that the underlying resource P' that may be tied to some view (hence the use of iProp) needs to be maintained at the same view.

LftL-full-acc-atomic-strong is needed to prove rules for full borrows that should not require a lifetime token, for examples, to prove commutativity with the later modality or the existential quantifier, or to convert a full borrow into a *fractured* borrow—which we will see next.

16.2 Other Forms of Borrows

Full borrows are perfect for modeling mutable references that can only have one user at a time. This is because accesses to full borrows are always sequential: at any moment in time, there can be only one ongoing access to a full borrow. For this reason, however, full borrows are not suitable for modeling types that are meant to be accessed concurrently by multiple threads, *e.g.*, shared references. This motivates two alternatives of full borrows: *fractured* borrows and *atomic* borrows.

Furthermore, full borrows are not persistent—they are unique resources and are withheld during an access so as to ensure at most one access to the underlying protected resource at a time. Non-persistency makes it difficult to build more complex protocols using full borrows. To make the borrows persistent but still maintain unique accesses, one can restrict accesses to a single-threaded manner, by employing non-atomic thread-local invariants (§11.3). This gives rise to *non-atomic* persistent borrows.

All of these different borrows are originally developed in the SC RustBelt work.[4] Table 16.1 compares several of their properties. These differences already exist in RustBelt except for the last column, which is unique to the relaxed-memory setting. (We will come back to that column

[4] Jung et al., "RustBelt: Securing the Foundations of the Rust Programming Language" [Jun+18a].

Borrow type	Access type	Access amount	Persistent	Communication among accesses	Access at local view
Full borrows $\&^{\kappa}_{\mathbf{full}}\, P$	sequential, non-atomic	full	no	yes	yes
Non-atomic borrows $\&^{\kappa/p.\mathcal{N}}_{\mathbf{na}}\, P$	sequential, non-atomic	full	yes	yes	yes
Fractured borrows $\&^{\kappa}_{\mathbf{frac}}\, \Phi$	concurrent, non-atomic	fractions	yes	no	yes
Atomic borrows $\&^{\kappa/\mathcal{N}}_{\mathbf{at}}\, P$	concurrent, atomic	full	yes	yes	no

TABLE 16.1: Comparison of borrow types

in §16.3.) All three forms of borrows—$\&^{\kappa}_{\mathbf{frac}}\, \Phi$, $\&^{\kappa/\mathcal{N}}_{\mathbf{at}}\, P$, and $\&^{\kappa/p.\mathcal{N}}_{\mathbf{na}}\, P$ for fractured, atomic and non-atomic borrows, respectively—are created from a full borrow $\&^{\kappa}_{\mathbf{full}}\, P$. All of their borrow assertions are persistent, so that the same borrow can be referred to and accessed by multiple parties. Fractured and atomic borrows are in fact accessible concurrently by multiple threads, but fractured borrows allow non-atomic accesses to only a *fraction* of the protected resource—ensuring that enough fractions remain for all participants at all times, whereas atomic borrows enforce a strict turn-taking scheme, allowing access to the *full* resource but only for a single atomic step of execution. On the other hand, non-atomic borrows allow non-atomic accesses to the full resource—like full borrows—but the unique access restriction is instead enforced through the non-atomic invariant token $[\mathrm{Na} : p.\mathcal{N}]$ that ties the borrow to the invariant pool $p.\mathcal{N}$.

We now look more closely at each form of borrows, with their respective rules given in Figure 16.2.

16.2.1 *Fractured Borrows*

To meaningfully talk about fractions of resources, fractured borrows assume a *predicate* Φ over fractions that is compatible with fraction addition: $\Phi(q_1 + q_2) \Leftrightarrow \Phi(q_1) * \Phi(q_2)$. With that, LFTL-FULL-FRACTURE (Figure 16.2) allows converting a full borrow $\&^{\kappa}_{\mathbf{full}}\, \Phi(1)$ of the full resource $\Phi(1)$ into a fracture borrow $\&^{\kappa}_{\mathbf{frac}}\, \Phi$. Note that Φ's compatibility with fraction addition can be proven as a persistent vProp fact under a later, as it will only be used once the resource $\Phi(1)$ is stored inside the internal invariant of the fracture borrow (*i.e.*, , stored under a later). As mentioned earlier, the creation of a fracture borrow (and similarly, of an atomic or non-atomic borrow) does not need to know whether κ is alive or not. This demonstrates again the flexibility of the indirect resource reclamation scheme with lifetimes: the inheritance will receive the resource after its associated lifetime ends, regardless of how the original full borrow may have been transformed or used.

With a lifetime token $[\kappa]_q$, the access rule LFTL-FRACT-ACC gives access to $\Phi(q')$ for some fraction q'. Once that exact q' of Φ is returned, we regain the lifetime token that we started the access with.

LFTL-FRACT-SHORTEN allows shortening the lifetime of a fractured bor-

Fractured borrows.

LftL-full-fracture
$$\frac{\triangleright\,\square(\forall q_1, q_2.\, \Phi(q_1 + q_2) \Leftrightarrow \Phi(q_1) * \Phi(q_2))}{\&^{\kappa}_{\mathbf{full}}\,\Phi(1) \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa}_{\mathbf{frac}}\,\Phi}$$

LftL-fract-acc
$$\&^{\kappa}_{\mathbf{frac}}\,\Phi \vdash [\kappa]_q \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \exists q'.\, \triangleright\,\Phi(q') * \left(\triangleright\,\Phi(q') \mathrel{-\!\!*\!\!\Rrightarrow}_{\mathcal{N}_{\mathrm{lft}}} [\kappa]_q\right)$$

LftL-fract-shorten
$$\kappa' \sqsubseteq \kappa \vdash \&^{\kappa}_{\mathbf{frac}}\,\Phi \Rightarrow \&^{\kappa'}_{\mathbf{frac}}\,\Phi$$

LftL-fract-iff
$$\triangleright\,\square(\forall q.\, \Phi(q) \Leftrightarrow \Psi(q)) \vdash \&^{\kappa}_{\mathbf{frac}}\,\Phi \Rightarrow \&^{\kappa}_{\mathbf{frac}}\,\Psi$$

Atomic persistent borrows.

LftL-full-at
$$\frac{\mathcal{N} \# \mathcal{N}_{\mathrm{lft}}}{\&^{\kappa}_{\mathbf{full}}\,P \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa/\mathcal{N}}_{\mathbf{at}}\,P}$$

LftL-at-acc
$$\&^{\kappa/\mathcal{N}}_{\mathbf{at}}\,P \vdash [\kappa]_q \quad {}^{\mathcal{N}_{\mathrm{lft}} \uplus \mathcal{N}}\!\Rrightarrow^{\mathcal{N}_{\mathrm{lft}}} \exists V_b.\, \sqcup_{V_b} \triangleright\,P * \left(\sqcup_{V_b} \triangleright\,P \;{}^{\mathcal{N}_{\mathrm{lft}}}\!\mathrel{-\!\!*\!\!\Rrightarrow}^{\mathcal{N}_{\mathrm{lft}} \uplus \mathcal{N}} [\kappa]_q\right)$$

LftL-at-shorten
$$\kappa' \sqsubseteq \kappa \vdash \&^{\kappa/\mathcal{N}}_{\mathbf{at}}\,P \Rightarrow \&^{\kappa'/\mathcal{N}}_{\mathbf{at}}\,P$$

LftL-at-iff
$$\triangleright\,\square(P \Leftrightarrow Q) \vdash \&^{\kappa/\mathcal{N}}_{\mathbf{at}}\,P \Rightarrow \&^{\kappa/\mathcal{N}}_{\mathbf{at}}\,Q$$

Non-atomic persistent borrows.

LftL-full-na
$$\&^{\kappa}_{\mathbf{full}}\,P \Rrightarrow_{\mathcal{N}_{\mathrm{lft}}} \&^{\kappa/p.\mathcal{N}}_{\mathbf{na}}\,P$$

LftL-na-acc
$$\&^{\kappa/p.\mathcal{N}}_{\mathbf{na}}\,P \vdash [\kappa]_q * [\mathsf{Na} : p.\mathcal{N}] \Rrightarrow_{\mathcal{N}_{\mathrm{lft}} \cup \mathcal{N}} \triangleright\,P * \left(\triangleright\,P \mathrel{-\!\!*\!\!\Rrightarrow}_{\mathcal{N}_{\mathrm{lft}} \cup \mathcal{N}} [\kappa]_q * [\mathsf{Na} : p.\mathcal{N}]\right)$$

LftL-na-shorten
$$\kappa' \sqsubseteq \kappa \vdash \&^{\kappa/p.\mathcal{N}}_{\mathbf{na}}\,P \Rightarrow \&^{\kappa'/p.\mathcal{N}}_{\mathbf{na}}\,P$$

LftL-na-iff
$$\triangleright\,\square(P \Leftrightarrow Q) \vdash \&^{\kappa/p.\mathcal{N}}_{\mathbf{na}}\,P \Rightarrow \&^{\kappa/p.\mathcal{N}}_{\mathbf{na}}\,Q$$

FIGURE 16.2: Selected rules for other borrow alternatives, sound in λ_{Rust} + ORC11

row, like LftL-full-shorten for full borrows. LftL-fract-iff says that fractured borrows are closed under the equivalence of the fractional predicate.

16.2.2 *Atomic Persistent Borrows*

In contrast to fractured borrows, atomic borrows do provide full access to the resources contained within, but only for a single, atomic instruction. This restriction of atomic borrows is encoded in its access rule LftL-at-acc using mask-changing viewshifts, like Inv-acc or CInv-acc. Recall that the namespace $\mathcal{N}_{\mathrm{lft}}$ is used for the internal invariant of the lifetime logic's model. The client of atomic borrows picks a namespace $\mathcal{N}$ *disjoint* from $\mathcal{N}_{\mathrm{lft}}$, at the creation of a atomic borrow from a full borrow using LftL-full-at, to allocate the underlying invariant that will store the resource P for concurrent atomic accesses. Of course, an atomic borrow is still tied to a lifetime specifying its period of validity, so that a lifetime token $[\kappa]_q$ is still required to guarantee that the lifetime is alive during the access.

Atomicity is crucial, for example, when several threads need to modify a shared variable, such as a reference counter for shared pointers. Therefore, while fractured borrows are designed to model *immutable* shared resources, atomic borrows are designed to model *mutable* shared resources in concurrent libraries. Naturally, these libraries use atomic memory accesses whose rules [5] are compatible with LftL-at-acc.

[5] *e.g.*, those in §10

Most importantly, LftL-at-acc only allows accesses to the borrowed resource P under a view-join modality. That is, the access gives us

$\sqcup_{V_b} \triangleright P$ and requires us to return the same $\sqcup_{V_b} \triangleright P$ at the end of the access. This is exactly the same requirement as that of the access rule CINV-ACC for iRC11 cancelable invariants (§11.2.1), for the same reason: we need to maintain soundness of the rules when porting them from Iris-SC to iRC11. The view-join modality helps us prevent unsound accidental synchronization between concurrent accesses.

This leads us back to the last two columns of Table 16.1. Each access of a fractured borrow does not communicate with another, because each access obtains an independent fraction of the borrowed contents, and thus can access that fraction at the accessing thread's local view. Meanwhile, each access of an atomic (or non-atomic or full) borrow obtains the full contents and can modify them and thus can communicate with other accesses. For atomic borrows, it means that without the view-join modality, concurrent accesses would transfer resources from one thread to another without actual physical synchronization, rendering our logic *unsound*! Imagine that if LFTL-AT-ACC without the view-join modality were sound, a thread π would access and modify P (*i.e.*, after the access P holds at π's local view), and immediately in the next step a different thread ρ would get synchronized access to P at its own local view, including the modifications made by π. In that case, the logic would allow synchronization from π to ρ without there being any physical synchronization to support that.

To avoid this unsoundness due to accidental synchronization between concurrent accesses, it is sufficient to protect P with a view-at modality, *i.e.*, $@_{V_b} \triangleright P$, and this would make atomic borrows roughly related to *objective invariants* (§11.1). However, as we also want to reclaim the borrowed resources, we need to employ the view-join modality to enable and maintain the safety of inheritance (Remark 15.3), in the same exact way as how iRC11 cancelable invariants maintains CANCEL-SAFE (§11.2).

Finally, LFTL-AT-SHORTEN and LFTL-AT-IFF say that atomic borrows are closed with respect to lifetime inclusion and predicate equivalence.

16.2.3 *Non-Atomic Persistent Borrows*

A non-atomic borrow $\&^{\kappa/p.\mathcal{N}}_{\mathbf{na}}\, P$ can be created from a full borrow $\&^{\kappa}_{\mathbf{full}}\, P$ using LFTL-FULL-NA (Figure 16.2). Non-atomic borrows offer the same sequential, non-atomic access style as that of full borrows, but manages the unique access restriction through the non-atomic invariant token $[\mathrm{Na} : p.\mathcal{N}]$ (for the namespace $\mathcal{N}$ under the pool p),[6] instead of the borrow assertion $\&^{\kappa}_{\mathbf{full}}\, P$ itself like in the case of full borrows. As such, non-atomic borrow assertions $\&^{\kappa/p.\mathcal{N}}_{\mathbf{na}}\, P$ are persistent and can be owned by multiple parties. Non-atomic borrows are thus useful to model single-threaded "smart pointer" types, *e.g.*, `Rc` or `RefCell`.

[6] see §11.3

The access rule LFTL-NA-ACC thus requires the invariant token $[\mathrm{Na} : p.\mathcal{N}]$ in addition to the lifetime token $[\kappa]_q$. Under the hood, the rule is supported by the non-atomic invariant access rule NAINV-ACC and the access rules for borrows.

LFTL-NA-SHORTEN and LFTL-NA-IFF say that non-atomic borrows are also closed with respect to lifetime inclusion and predicate equivalence.

16.3 Adaption of the Lifetime Logic's Model in iRC11

In this section, we briefly discuss the adaptation of the lifetime logic's model on top of iRC11. This work was spearheaded by Jacques-Henri Jourdan, one of the designers of the original Iris-SC lifetime logic. Therefore, the discussion here is not a contribution of this dissertation, and is only provided for the sake of completeness.

Concept 16.2 (The Invariant for Lifetimes and Borrows)**.** The model of the lifetime logic sets up a global invariant that governs the protocols for all lifetimes and their associated borrows. At a very high-level, the global invariant $\mathsf{LftInv}(\kappa)$ for each lifetime κ is roughly as follows.

- κ is either alive or dead: $\mathsf{LftInv}(\kappa) = \mathsf{LftAlive}(\kappa) \lor \mathsf{LftDead}(\kappa)$.
- For each borrow $\&^\kappa P$ associated with κ, when κ is already dead, $\mathsf{LftDead}(\kappa)$ tracks if the inheritance $\mathsf{Inh}(\kappa, P)$[7] has been used. For borrows whose inheritance have not been used, $\mathsf{LftDead}(\kappa)$ carries the resources of those borrows. This part of the protocol is needed for LftL-full-inh.[8]
- When κ is still alive, $\mathsf{LftAlive}(\kappa)$ tracks the states of each borrow $\&^\kappa P$ associated with κ. The borrow can be in one of three states:
  - The borrow is not being accessed, and the resource P is still being hold by $\mathsf{LftAlive}(\kappa)$.
  - The borrow is being opened by someone, so the resource P has been taken out of $\mathsf{LftAlive}(\kappa)$. On the other hand, as part of the borrowing process, some fraction q of the lifetime token $[\kappa]_q$ must have been put in $\mathsf{LftAlive}(\kappa)$ as a deposit. This corresponds to the rules LftL-full-acc and LftL-full-ret.[9]
  - The borrow has been reborrowed at a strictly shorter lifetime κ'. $\mathsf{LftAlive}(\kappa)$ then needs to track how to reclaim resources for $\&^\kappa P$ from the reborrows when κ is killed. This part of the protocol models the reborrowing rule LftL-reborrow.[10]

[7] see §15.2

[8] see Figure 15.1

[9] see Figure 15.1

[10] see Figure 16.1

As typical for logics built in Iris, the lifetime invariant $\mathsf{LftInv}(\kappa)$ together with the lifetime and borrow assertions (*e.g.*, the lifetime token $[\kappa]_q$ or the borrow assertion $\&^\kappa_{\mathbf{full}} P$, as listed in §15.2) are modeled with user-defined ghost state that encodes the desirable properties needed by the protocols. For example, in the Iris-SC model of the lifetime logic, $[\kappa]_q$ and $[\dagger\kappa]$ are defined purely with disjoint ghost elements (so as to satisfy LftL-tok-not-dead),[11] and $\&^\kappa_{\mathbf{full}} P$ is defined with an exclusive ghost element, together with some invariant that ties P to the global invariant $\mathsf{LftInv}(\kappa)$.

[11] see Figure 15.1

The Iris-SC lifetime logic, however, models the lifetime token $[\kappa]_q$ as a view-agnostic assertion, simply asserting the ghost ownership of some fraction q of the ghost location κ:

$$[\![[\kappa]_q]\!] ::= \boxed{q}^\kappa \qquad \text{(LftL-tok-SC-model)}$$

This model is insufficient for the RMC lifetime logic, because it does not guarantee the safety of inheritance (Remark 15.3) in the presence of concurrent borrow accesses that do not establish synchronization. In $\mathsf{RB}_{\mathtt{rlx}}$, we instead need to enrich this model so that it depends on the view at which $[\kappa]_q$ is asserted:

$$[\![[\kappa]_q]\!] ::= \exists V_{\mathtt{tok}}. \boxed{(q, V_{\mathtt{tok}})}^{\kappa} * \sqsupseteq V_{\mathtt{tok}} \qquad \text{(LFT-TOK-RLX-MODEL)}$$

In this model, the ghost element is no longer just a fraction q, but a pair of the fraction and the *token view* $V_{\mathtt{tok}}$. The token view $V_{\mathtt{tok}}$ represents what this particular fraction of the token has *observed*, *i.e.*, what borrow accesses the token has participated in. The model requires that the local view at which the token is interpreted has also at least observed what $[\kappa]_q$ has observed: $\sqsupseteq V_{\mathtt{tok}}$.

Note that this model of lifetime token is exactly the same as the model of cancelable invariant token $\heartsuit^{\gamma}_{q}$, given by CINV-MODEL-TOK,[12] because they share the same goal: guaranteeing the safety of inheritance/cancelation. The protocol for borrow inheritance, however, is quite more elaborate than invariant cancelation, because a lifetime can be associated with multiple borrows, who in turn can be reborrowed. In the following, we sketch the proofs for the inheritance and access rules, for full, fractured, and atomic borrows in iRC11. As we will see shortly, the key idea of the proofs is to associate views not only with the lifetime token assertions, but with all the other assertions that play a role in the lifetime logic.

[12] see Definition 11.4, §11.2.2

16.3.1 *Full Borrows*

PROVING INHERITANCE. To prove LFTL-FULL-INH (Figure 15.1) sound for $\mathsf{RB}_{\mathtt{rlx}}$, we get to assume $[\dagger\kappa]$ and $\mathsf{Inh}(\kappa, P)$ at the thread's current local view—call it V, and we need to produce $\triangleright P$ at that same view V. Now, the lifetime logic is responsible for controlling ownership of the content of the borrow, P; so let us assume that, when no threads are accessing the borrow, P is maintained at a view V_b, which we call the *content view*. Furthermore, by owning $[\dagger\kappa]$, we know that the global lifetime invariant $\mathsf{LftInv}(\kappa)$ is in the $\mathsf{LftDead}(\kappa)$ disjunct, and by owning $\mathsf{Inh}(\kappa, P)$, we know that the inheritance has not been applied yet (we are the one to apply it), so $\mathsf{LftDead}(\kappa)$ is still holding on to P at the view V_b. In other words, $\mathsf{LftDead}(\kappa)$ is holding $@_{V_b} \triangleright P$.

When we apply the inheritance, we will therefore be trading $\mathsf{Inh}(\kappa, P)$ for $@_{V_b} \triangleright P$ from $\mathsf{LftDead}(\kappa)$. If we can show that $\sqsupseteq V_b$, *i.e.*, the current thread has locally observed V_b, we can use VA-ELIM (Figure 8.3) to obtain $\triangleright P$ and finish the proof.

The key to proving the goal $\sqsupseteq V_b$ is to:

- associate with each lifetime logic assertion a view that represents what the assertion has observed, *i.e.*, what activities it has been involved in; and then
- establish and maintain for those associated views sufficiently strong invariants in $\mathsf{LftInv}(\kappa)$ so that we can prove our goal.

Below, we summarize a list of some associated views for assertions that interact with a full borrow $\&^{\kappa}_{\mathbf{full}} P$:

- the *content view* V_b at which the lifetime invariant keeps P;
- the *token views* $V_{\mathtt{tok}}$, one for every $[\kappa]_q$;
- the *full token view* V_κ of the full token $[\kappa]_1$, defined as the join of all token views $V_{\mathtt{tok}}$;
- the *dead token view* $V_\dagger \sqsupseteq V_\kappa$; and
- the *borrow view* V_B of the borrow assertion $\&^{\kappa}_{\mathbf{full}} P$.

In order to prove $\sqsupseteq V_b$ for LftL-full-inh, we enforce the following property in $\mathsf{LftInv}(\kappa)$ for the borrow in question:

$$V_b \sqsubseteq V_\kappa \qquad \text{(LftL-full-bor-inv-1)}$$

By owning $[\dagger\kappa]$, we have for the current thread $\sqsupseteq V_\dagger$. By view-monotonicity (*i.e.*, VS-mono, Figure 8.3), the current thread must have observed $\sqsupseteq V_\kappa$ and $\sqsupseteq V_b$. □

This shows that LftL-full-bor-inv-1 allows us to prove LftL-full-inh. But what is the intuition for this invariant? As hinted at before, each piece of a lifetime token is intended to bear "witness" to any access to the borrow that the token is used for. Ultimately, the full token $[\kappa]_1$—which is the join of all tokens—must have witnessed *all* accesses to the borrow ($V_b \sqsubseteq V_\kappa$). Therefore, a thread owning the full token should have observed all modifications made to P by all of those accesses, so it can safely kill the lifetime and produce the dead token $[\dagger\kappa]$. Effectively, a thread owning $[\dagger\kappa]$ must have observed all modifications made to P ($V_b \sqsubseteq V_\kappa \sqsubseteq V_\dagger$), so it can use the inheritance to reclaim P at its local view.

How do we maintain LftL-full-bor-inv-1? $V_b \sqsubseteq V_\kappa$ is maintained by the rule LftL-full-ret. Note that the lifetime token $[\kappa]_q$ is withheld during the access. When the access finishes and P is returned to the borrow at an updated view V'_C, the rule uses V'_C to update the view of the withheld token $[\kappa]_q$ from $V_{\mathtt{tok}}$ to $V_{\mathtt{tok}} \sqcup V'_C$ before returning it to the user. Since V_κ is the join of all lifetime tokens, this effectively updates V_κ to $V_\kappa \sqcup V'_C \sqsupseteq V'_C$. With this, the invariant is re-established. Note that this line of reasoning mirrors that of the proof of the cancelable invariant access rule CInv-acc in §11.2.2.

Proving accesses. To prove the rule for accessing full borrows, LftL-full-acc, we assume the lifetime token and the borrow assertion, and we need to provide the user with synchronized access to P at the thread's local view. Assuming that we can prove the return policy $\mathsf{Ret}(\kappa, P, q)$, we still need to ensure that P *holds* locally, *i.e.*, we have $\sqsupseteq V_b$. For this, we rely on the following invariant:

$$V_b \sqsubseteq V_B \qquad \text{(LftL-full-bor-inv-2)}$$

That is, the content view V_b of P is always included in the borrow view V_B of $\&^{\kappa}_{\mathbf{full}} P$. In other words, owning $\&^{\kappa}_{\mathbf{full}} P$ should imply $\sqsupseteq V_B$, which

in turn implies $\sqsupseteq V_b$. Like LftL-full-bor-inv-1, LftL-full-bor-inv-2 is straightforward to maintain: the borrow assertion $\&^{\kappa}_{\mathbf{full}} P$ is withheld during the access; and when an access is returned, we update the borrow assertion's V_B with the new content view V'_C. □

16.3.2 *Fractured Borrows*

Fractured borrows, like all borrows, need to guarantee that all accesses happen-before the inheritance is applied. However, unlike full borrows where all accesses are ordered, accesses to a fractured borrow can happen independently from one another. Thus, for fractured borrows, we need to maintain that independent changes to fractions of Φ made by independent accesses are all observed by the thread performing the inheritance. The key to achieve this is to recognize that the content view of Φ is no longer a single view, but consists of two views:

(1) the view V_{YTBA} of the *yet-to-be-accessed* portion of Φ,

(2) the view V_{AA} of the *already-accessed* portion of Φ.

V_{YTBA} is the view of the chunk of Φ that has not been given out to any access, as well as the view at which the fractured borrow was created. Meanwhile, V_{AA} tracks all the changes made by the accesses to all the chunks that have been given out to those accesses.

With that in mind, we repeat what we did for full borrows: defining the invariants for the inheritance and the accesses of fractured borrows.

PROVING INHERITANCE. We enforce an invariant on the two views of a fractured borrow $\&^{\kappa}_{\mathbf{frac}} \Phi$:

$$V_{\mathrm{YTBA}} \sqcup V_{\mathrm{AA}} \sqsubseteq V_{\kappa} \qquad \text{(LftL-fract-bor-inv-1)}$$

That is, instead of using a single content view V_b like in LftL-full-bor-inv-1, we use the view $V_{\mathrm{YTBA}} \sqcup V_{\mathrm{AA}}$ which is the view of the full resource $\Phi(1)$, and require that it is always included in the full token view κ. Thus, by similar reasoning to LftL-full-bor-inv-1, any changes to the fractured borrow's contents are guaranteed to happen-before the moment the inheritance is applied.

PROVING ACCESSES. We maintain a second invariant:

$$V_{\mathrm{YTBA}} \sqsubseteq V_B \qquad \text{(LftL-fract-bor-inv-2)}$$

This invariant enables the synchronized access to a fraction of Φ in the accessing rule LftL-fract-acc: when the rule is applied with a thread owning a borrow assertion $\&^{\kappa}_{\mathbf{frac}} \Phi$ with some borrow view V_B, we have that the thread has observed V_B, *i.e.*, $\sqsupseteq V_B$. Consequently, the thread has $\sqsupseteq V_{\mathrm{YTBA}}$, and it can then obtain some portion $\Phi(q')$ from the yet-to-be-accessed chunk at its local view.

16.3.3 *Atomic Persistent Borrows*

The challenge of porting atomic borrows is the same one of porting cancelable invariants to RMC: atomic borrows allow concurrent accesses

to the full contents of the borrow, where each access can modify the contents and thus can communicate with other accesses. While concurrent accesses are also possible with fractured borrows, this situation does not apply to them because each access of a fractured borrow obtains an independent fraction of the underlying contents.

The content view V_b of the borrow content P can be constantly changed by different threads with atomic accesses to the borrow $\&_{\mathbf{at}}^{\kappa/\mathcal{N}} P$, and there is in general no relationship between threads' local views and the content view V_b. This is why, like the cancelable invariant access rule CINV-ACC, the atomic borrow access rule LFTL-AT-ACC only provides the borrow content protected under a view-join modality, in the form $\sqcup_{V_b} \triangleright P$. It is then the obligation of the clients of the atomic borrow to eliminate the view-join modality in order to get actual access to P. The main role of the view-join modality is to maintain safety of inheritance: we still have that $V_b \sqsubseteq V_\dagger$, so with $\sqcup_{V_b} \triangleright P$ and $\sqsupseteq V_\dagger$, we can use VJ-ELIM (Figure 8.3) to safely inherit $\triangleright P$. The proofs of inheritance and accesses for atomic borrows are therefore very similar to the proofs for cancelable invariants (see §11.2.2).

CHAPTER SUMMARY. In this chapter, we have reviewed the adaptation of the RustBelt lifetime logic to RMC, on top of our logic iRC11. This is sufficient to establish "Task 2: re-proving the safety of the λ_{Rust} type system under RMC", due to the fact that the lifetime logic interface only changes in the atomic borrow access rule. In this part's remaining chapters, we demonstrate some of the results in "Task 1", where we rely on the facilities of iRC11 as well as atomic persistent borrows to re-verify the safety of several Rust libraries, considering their real, relaxed-memory implementations.

17

GPS Single-Location Protocols

In this chapter, we present how to encode GPS single-location protocols[1] in iRC11, using atomics points-to (Chapter 10). From an organizational perspective, this encoding should have been introduced as part of iRC11. However, the encoding was mainly developed to perform verifications of relaxed memory concurrent libraries for $\mathsf{RB}_{\mathtt{rlx}}$ (*i.e.*, for Task 2, see §14.2). In Chapter 18 and Chapter 19, we will demonstrate how GPS protocols are used to verify respectively, the reader-writer lock and the atomic reference counting against their RustBelt's semantic types.

[1] Turon et al., "GPS: navigating weak memory with ghosts, protocols, and separation" [TVD14]; Kaiser et al., "Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris" [Kai+17].

We will start in §17.1 with the interfaces of several different kinds of GPS protocols in iRC11. The kinds of GPS protocols differ on (1) how long they can be accessed, and on (2) how much concurrency they allow. For (2), there are *concurrent* protocols which allow *arbitrary* concurrent accesses, and *single-writer* protocols which allow either single-writer or CAS-only writers. For (1), there are *persistent* protocols which stay alive forever, *cancelable* protocols which stay alive as long as the cancelable invariant token is still available, and *atomic-borrows*-based protocols whose lifecycle is tied to a lifetime. Atomic-borrows-based protocols will be used in the verification of the reader-writer lock (Chapter 18), while cancelable protocols will be used in the verification of the atomic reference counting (Chapter 19).

In §17.2, we provide intermediate-level interfaces of GPS protocols, which we call the *middleware*. We show how middleware GPS protocols are a common interface that can be combined with different types of invariants to derive the surface-level GPS protocols. Specifically, we define, as examples, the models of persistent/concurrent protocols, of cancelable/single-writer protocols, and of atomic-borrows-based/single-writer protocols using the middleware protocols respectively together with objective invariants (§11.1), cancelable invariants (§11.2), and atomic borrows (§16.2).

Finally, in §17.3, we briefly explain the model of middleware protocols, which supports both concurrent and single-writer protocols, and which is built upon the atomic points-to assertion.

17.1 Surface-level GPS Protocols in iRC11

Definition 17.1 (GPS Protocol Type)**.** A GPS protocol for a single location ℓ restricts how ℓ's history can grow. The setup therefore lets the user

pick a type of *protocol states*, and then pick a state for every write in the history. The restriction is that, as the history grows, the states picked can only grow following a *pre-order* (*i.e.*, it is reflexive and transitive) that is also picked up front by the user. More concretely, the user needs to pick $S \in \mathit{ProtoState} ::= (T_S, \sqsubseteq)$ where T_S is the state type that enjoys the pre-order $\sqsubseteq$.

Example 17.2 (GPS Protocol Types)**.** We provide a list of common protocol types.

$$\begin{aligned} S_{()} &::= ((), \lambda\_.\, \mathsf{True}) \\ S_{\mathbb{B}} &::= (\mathbb{B}, \lambda s_1, s_2.\, s_2 = \texttt{true} \vee s_1 = \texttt{false}) \\ S_{\mathbb{N}} &::= (\mathbb{N}, \leq) \\ S_{\wp(A)} &::= (\wp(A), \subseteq) \\ S_{\mathsf{List}(A)} &::= (\mathsf{List}(A), \mathsf{prefix}) \end{aligned}$$

The unit protocol $S_{()}$ has a single state and a trivial order. The boolean protocol $S_{\mathbb{B}}$ has two states **`false`** and **`true`** and can only grow from the former to the latter. The natural protocol $S_{\mathbb{N}}$ has states as natural numbers and can only grow following the natural number order. The set protocol $S_{\wp(A)}$ has states as sets of elements from the type A, and states can only grow by adding more elements. The list protocol $S_{\mathsf{List}(A)}$ has states as lists of elements from A, and lists can only grow by appending.

17.1.1 *Persistent Concurrent Protocols*

The interface of persistent concurrent GPS protocols is given in Figure 17.1 and Figure 17.2.

Definition 17.3 (Persistent Concurrent Protocol Assertion)**.** The protocol assertion $\boxed{(\ell, \gamma) : (t, s, v) \mid \mathcal{I}}^{\mathcal{N}}$ says that the location ℓ is persistently (permanently) governed by a GPS protocol *interpretation* $\mathcal{I}$, under the namespace $\mathcal{N}$ and the ghost location γ.

- The namespace $\mathcal{N}$ is picked by the creator (user) of the protocol to house the protocol invariant. As GPS protocols are built with Iris invariants, namespaces allow the user to atomically access multiple invariants (housed in disjoint namespaces). This ability is particularly important for deterministic pointer comparison in the semantics of ORC11: when performing a compare-and-swap of location values,[2] if the two compared values are locations governed by persistent concurrent protocols in disjoint namespaces, we know atomically that the two locations are both alive, and hence their comparison is deterministic.

- The ghost location γ is needed to store the ghost state of the protocol's model. It is important in the case the protocol is cancelable: multiple protocols can be tied to one location (but only at most one can be alive at any moment in time), the ghost location γ uniquely identifies a protocol instance. For persistent concurrent protocols that are not cancelable, γ never changes and can be safely ignored.

[2] for example, see GPS-CON-CAS-LOC in Figure 17.2

- The protocol interpretation $\mathcal{I}$ is a pair of predicates $(\mathcal{I}_r, \mathcal{I}_w)$ called the *read* and *write* interpretations, respectively. Both predicates are of the type $(\mathit{Loc} \times \mathit{GName} \times \mathit{Time} \times T_S \times \mathit{Val}) \to \mathsf{vProp}$, where S is the protocol type picked by the user. $\mathcal{I}_r(\ell, \gamma, t, s, v)$ represents the resources one would get from the protocol when reading the location's write message of timestamp t and value v. Meanwhile, $\mathcal{I}_w(\ell, \gamma, t, s, v)$ represents the resources one needs to give up to the protocol when writing a message of of timestamp t and value v to ℓ and tying the state s to that message.
- In addition to asserting that the protocol exists persistently, the assertion $\boxed{(\ell, \gamma) : (t, s, v) \mid \mathcal{I}}^{\mathcal{N}}$ also says that the current thread has observed the write message of timestamp t, value v, and state s.

We can now look more closely at some of the rules for persistent concurrent protocols, first in Figure 17.1 and then in Figure 17.2. We explain first some simple rules.

- GPS-CON-AGREE says that the pre-order of protocol states are *total per protocol*, as it follows the timestamp order—which encodes the modification order mo—of a location's history.
- GPS-CON-ATOM-PTSTO says that, by knowing that a persistent protocol exists for ℓ, we can *atomically*—due to the mask-changing fancy update—access the primitive atomic points-to $\ell \mapsto h$ of ℓ subjectively. This is sufficient to deduce that ℓ is still alive, and can be useful in deterministic pointer comparison.
- GPS-CON-INIT allows one to allocate a new GPS protocol from a non-atomic points-to $\ell \mapsto v$ and the write interpretation $\mathcal{I}_w$ of the latest write (with value v).

The rule GPS-CON-READ shows how persistent GPS protocols allow us to read the location ℓ with the access mode o. Albeit looking simple, the rule requires a bit of explanation.

- The read must use an atomic access mode for read, *i.e.*, $o \in \{\mathbf{rlx}, \mathbf{acq}\}$. o can also be **sc**, but we do not provide a stronger rule for SC accesses in iRC11.
- The rule can be used with the mask $\mathcal{E}$ that includes the protocol's namespace $\mathcal{N}$, *i.e.*, when the protocol's invariant is enabled.
- If the reading thread π's observation for ℓ was at least the message (t, s, v), then the read will return a message (t', s', v') (and its observation) that is no-mo-earlier than t.
- Additionally, if the user can prove a fancy update $\mathsf{Extract}_{\mathcal{I}}$ that can extract the resource $R(t', s', v')$ from either the read ($\mathcal{I}_r$) or write ($\mathcal{I}_w$) interpretation of the protocol for the read message t', then the user can get back R in the post-condition.

$$\frac{\mathsf{persistent}(\mathcal{R}(\ell,\gamma,t,s,v))\quad \mathsf{timeless}(\mathcal{R}(\ell,\gamma,t,s,v))}{\mathsf{persistent}(\boxed{(\ell,\gamma):(t,s,v)\mid\mathcal{I}}^{\mathcal{N}})}$$

GPS-CON-RDR
$$\boxed{(\ell,\gamma):(t,s,v)\mid\mathcal{I}}^{\mathcal{N}} \vdash \mathcal{R}(\ell,\gamma,t,s,v)$$

GPS-CON-AGREE
$$\boxed{(\ell,\gamma):(t_1,s_1,v_1)\mid\mathcal{I}}^{\mathcal{N}} * \boxed{(\ell,\gamma):(t_2,s_2,v_2)\mid\mathcal{I}}^{\mathcal{N}} \Rrightarrow_{\mathcal{E}} (t_1\le t_2 \Rightarrow s_1\sqsubseteq s_2)\wedge(t_2\le t_1\Rightarrow s_2\sqsubseteq s_1)$$

GPS-CON-ATOM-PTSTO
$$\boxed{(\ell,\gamma):(t,s,v)\mid\mathcal{I}}^{\mathcal{N}} \quad \mathcal{E}\Rrightarrow^{\mathcal{E}\setminus\uparrow\mathcal{N}} \exists h.\, \triangleright\langle\mathsf{subj}\rangle\,\ell\mapsto h$$

GPS-CON-INIT
$$\ell\mapsto v * (\forall\gamma, t.\, \triangleright\mathcal{I}_w(\ell,\gamma,t,s,v)) \Rrightarrow_{\mathcal{E}} \exists\gamma,t.\,\boxed{(\ell,\gamma):(t,s,v)\mid\mathcal{I}}^{\mathcal{N}}$$

$$\nabla_\pi^{o?\mathtt{rlx}} P ::= \mathbf{if}\,(o=\mathtt{rlx})\,\mathbf{then}\,\nabla_\pi P\,\mathbf{else}\,P$$
$$\Delta_\pi^{o?\mathtt{rlx}} P ::= \mathbf{if}\,(o=\mathtt{rlx})\,\mathbf{then}\,\Delta_\pi P\,\mathbf{else}\,P$$
$$\mathsf{Extract}_{\mathcal{I}}(\ell,\gamma,t',s',v',R,\mathcal{E},\mathcal{N}) ::= \langle\mathsf{obj}\rangle \wedge \begin{cases}\mathcal{I}_r(\ell,\gamma,t',s',v') \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \mathcal{I}_r(\ell,\gamma,t',s',v') * R(t',s',v')\\ \mathcal{I}_w(\ell,\gamma,t',s',v') \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \mathcal{I}_w(\ell,\gamma,t',s',v') * R(t',s',v')\end{cases}$$

GPS-CON-READ
$$\frac{\mathtt{rlx}\sqsubseteq o \qquad \uparrow\mathcal{N}\subseteq\mathcal{E}}{\left\{\boxed{(\ell,\gamma):(t,s,v)\mid\mathcal{I}}^{\mathcal{N}} * \forall t'\ge t, s'\sqsupseteq s, v'.\,\mathsf{Extract}_{\mathcal{I}}(\ell,\gamma,t',s',v',R,\mathcal{E},\mathcal{N})\right\}\ {}^{*o}\ell\ \mathsf{in}\ \pi\ \left\{v'.\,\exists t'\ge t, s'\sqsupseteq s.\,\boxed{(\ell,\gamma):(t',s',v')\mid\mathcal{I}}^{\mathcal{N}} * \nabla_\pi^{o?\mathtt{rlx}} R(t',s',v')\right\}_{\mathcal{E}}}$$

GPS-CON-WRITE
$$\frac{\mathtt{rlx}\sqsubseteq o \qquad \uparrow\mathcal{N}\subseteq\mathcal{E} \qquad \forall s.\, s\sqsubseteq s'}{\left\{\boxed{(\ell,\gamma):(t,s,v)\mid\mathcal{I}}^{\mathcal{N}} * \Delta_\pi^{o?\mathtt{rlx}}\left(\forall t'>t.\,\mathcal{R}(\ell,\gamma,t',s',v') \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \mathcal{I}_w(\ell,\gamma,t',s',v')\right)\right\}\ \ell :=_o v'\ \mathsf{in}\ \pi\ \left\{☣.\,\exists t'>t.\,\boxed{(\ell,\gamma):(t',s',v')\mid\mathcal{I}}^{\mathcal{N}}\right\}_{\mathcal{E}}}$$

FIGURE 17.1: Rules for GPS Persistent Concurrent Protocols

- The extraction must be proven *objectively*,[3] without changing the interpretations.

[3] see §8.4

- If the read access mode is at least **acq**, then R is acquired as-is. If the read is only relaxed (**rlx**), R will be returned protected by the acquire modality ∇,[4] which can later be eliminated by with an acquire fence. To succinctly encapsulate both cases in the rule, we use the conditional notation $\nabla_\pi^{o?\mathbf{rlx}}$ for the acquire modality.

[4] see §8.3

GPS-CON-WRITE allows us to write a value v' to ℓ and tie the state s' to that write.

- The write access mode o should be $\{\mathbf{rlx}, \mathbf{rel}\}$.
- The value v' will be tied to the timestamp t' strictly greater than the writing thread's observation of timestamp t for ℓ.
- As the protocol states need to grow with respect to the pre-order $\sqsubseteq$, the user needs to guarantee that the new state s' is greater than the protocol's current state. However, as the protocol allows concurrent writes, it is difficult to know what the current state is. Consequently, the rule requires the user to prove that s' is be greater than any state ($\forall s.\, s \sqsubseteq s'$). This is quite a strong requirement, and expectedly, protocols for arbitrarily concurrent locations are often trivial.
- Most importantly, the user has to provide the *write interpretation* $\mathcal{I}_w$ for the new write message (t', s', v'). If the write access mode is at least **rel**, then the user only has to provide $\mathcal{I}_w$ right before the write. If the mode is relaxed, then the user has to provide $\mathcal{I}_w$ at the most recent release fence, *i.e.*, provide $\mathcal{I}_w$ under the release modality Δ.[5] Similarly to the read rule, we use the conditional notation $\Delta_\pi^{o?\mathbf{rlx}}$ here to combine the two cases.

[5] see §8.3

- Additionally, in proving $\mathcal{I}_w$, the user can assume the *pure-ghost* observation $\mathcal{R}(\ell, \gamma, t', s', v')$ of the new message v' with the state s'. That is, the write interpretation can include the fact that the write has been registered in the protocol. Intuitively, the observation $\mathcal{R}(\ell, \gamma, t', s', v')$ is simply the assertion $\boxed{(\ell, \gamma) : (t', s', v') \mid \mathcal{I}}^{\mathcal{N}}$ *without* asserting the existence of the protocol invariant. As such, $\mathcal{R}$ is both persistent and timeless. The relation between the two observations is shown in the rule GPS-CON-RDR.

We now look at the rule GPS-CON-CAS-INT in Figure 17.2, which allows us to perform a compare-exchange from an *integer* value v_r to a value v_w.

- Recall that o_r and o_w are respectively the read and write access modes in the successful exchange case. Meanwhile, o_f is the read access mode in the failure case. Like the read and write rules, these access modes must be at least **rlx**, and dictate whether the resources the user needs to provide or can acquire will be protected by the release or acquire modality, respectively. As in the read and

GPS-CON-CAS-INT

$$\mathbf{rlx} \sqsubseteq o_f, o_r, o_w \qquad \uparrow\mathcal{N} \subseteq \mathcal{E} \qquad v_r \in \mathbb{Z}$$

$$\left\{ \begin{array}{l} \boxed{(\ell,\gamma) : (t,s,v) \;\middle|\; \mathcal{I}}^{\mathcal{N}} * \\ \triangleright \langle \mathsf{obj} \rangle \left(\forall t' \geq t, s' \sqsupseteq s, v'. \left(\mathcal{I}_r(\ell,\gamma,t',s',v') \vee \mathcal{I}_w(\ell,\gamma,t',s',v')\right) \mathrel{-\!\!*} \vdash v' =^? v_r \right) * \\ \Delta_\pi^{o_w ?\mathbf{rlx}}\, P * \\ \Delta_\pi^{o_w ?\mathbf{rlx}} \forall t' \geq t, s' \sqsupseteq s. \wedge \left\{ \begin{array}{l} \triangleright (\forall v' \neq v_r. \mathsf{Extract}_{\mathcal{I}}(\ell,\gamma,t',s',v',R,\mathcal{E},\mathcal{N})) \\ * \left\{ \begin{array}{l} \langle \mathsf{obj} \rangle \left(\triangleright \mathcal{I}_w(\ell,\gamma,t',s',v_r) \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \left(\triangleright Q_1(t',s') * \triangleright Q_2(t',s')\right)\right) \\ \left(P * \triangleright Q_2(t',s')\right) \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \exists s'' \sqsupseteq s'. \forall t'' > t'. \\ \quad \triangleright \mathcal{R}(\ell,\gamma,t'',s'',v_w) \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \\ \quad * \left\{ \begin{array}{l} \langle \mathsf{obj} \rangle \left(\triangleright Q_1(t',s') \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \mathcal{I}_r(\ell,\gamma,t',s',v')\right) \\ \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \triangleright \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \left(Q(t'',s'') * \mathcal{I}_w(\ell,\gamma,t'',s'',v_w)\right) \end{array} \right. \end{array} \right. \end{array} \right. \end{array} \right\}$$

$\mathbf{CAS}^{o_f,o_r,o_w}(\ell, v_r, v_w)$ in π

$$\left\{ b. \exists t', v', s' \sqsupseteq s. \vee \left\{ \begin{array}{ll} v_r \neq v' * t \leq t' * \boxed{(\ell,\gamma) : (t',s',v') \;\middle|\; \mathcal{I}}^{\mathcal{N}} * \nabla_\pi^{o_f ?\mathbf{rlx}} R(t',s',v') * \nabla_\pi^{o_w ?\mathbf{rlx}} P & b = \mathbf{false} \\ v_r = v' * t < t' * \boxed{(\ell,\gamma) : (t',s',v_w) \;\middle|\; \mathcal{I}}^{\mathcal{N}} * \nabla_\pi^{o_r ?\mathbf{rlx}} Q(t',s') & b = \mathbf{true} \end{array} \right. \right\}_{\mathcal{E}}$$

GPS-CON-CAS-LOC

$$\mathbf{rlx} \sqsubseteq o_f, o_r, o_w \qquad \uparrow\mathcal{N} \subseteq \mathcal{E} \qquad \ell_r \in \mathit{Loc}$$

$$\left\{ \begin{array}{l} \boxed{(\ell,\gamma) : (t,s,v) \;\middle|\; \mathcal{I}}^{\mathcal{N}} * \\ P_{\mathrm{cmp}} * \Box(P_{\mathrm{cmp}} \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \exists q, h_r. \triangleright \langle \mathsf{subj} \rangle\, \ell_r \overset{q}{\mapsto} h_r) * \\ \triangleright \langle \mathsf{obj} \rangle \left(\forall t' \geq t, s' \sqsupseteq s, v'. \left(\mathcal{I}_r(\ell,\gamma,t',s',v') \vee \mathcal{I}_w(\ell,\gamma,t',s',v')\right) \mathrel{-\!\!*} \vdash v' =^? \ell_r \right) * \\ \Delta_\pi^{o_w ?\mathbf{rlx}}\, P * \\ \Delta_\pi^{o_w ?\mathbf{rlx}} \forall t' \geq t, s' \sqsupseteq s. \wedge \left\{ \begin{array}{l} \triangleright (\forall v' \neq \ell_r. \mathsf{Extract}_{\mathcal{I}}(\ell,\gamma,t',s',v',R,\mathcal{E},\mathcal{N})) \\ * \left\{ \begin{array}{l} \langle \mathsf{obj} \rangle \left(\forall \ell' \neq \ell_r. \triangleright \mathcal{I}_w(\ell,\gamma,t',s',\ell') \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \exists q, h'. \triangleright \ell' \overset{q}{\mapsto} h' \right) \\ \langle \mathsf{obj} \rangle \left(\triangleright \mathcal{I}_w(\ell,\gamma,t',s',\ell_r) \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \left(\triangleright Q_1(t',s') * \triangleright Q_2(t',s')\right)\right) \\ \left(P * \triangleright Q_2(t',s')\right) \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \exists s'' \sqsupseteq s'. \forall t'' > t'. \\ \quad \triangleright \mathcal{R}(\ell,\gamma,t'',s'',v_w) \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \\ \quad * \left\{ \begin{array}{l} \langle \mathsf{obj} \rangle \left(\triangleright Q_1(t',s') \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \mathcal{I}_r(\ell,\gamma,t',s',v')\right) \\ \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \triangleright \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \left(Q(t'',s'') * \mathcal{I}_w(\ell,\gamma,t'',s'',v_w)\right) \end{array} \right. \end{array} \right. \end{array} \right. \end{array} \right\}$$

$\mathbf{CAS}^{o_f,o_r,o_w}(\ell, \ell_r, v_w)$ in π

$$\left\{ b. \exists t', v', s' \sqsupseteq s. P_{\mathrm{cmp}} * \vee \left\{ \begin{array}{ll} \ell_r \neq v' * t \leq t' * \boxed{(\ell,\gamma) : (t',s',v') \;\middle|\; \mathcal{I}}^{\mathcal{N}} * \nabla_\pi^{o_f ?\mathbf{rlx}} R(t',s',v') * \Delta_\pi^{o_w ?\mathbf{rlx}} P & b = \mathbf{false} \\ \ell_r = v' * t < t' * \boxed{(\ell,\gamma) : (t',s',v_w) \;\middle|\; \mathcal{I}}^{\mathcal{N}} * \nabla_\pi^{o_r ?\mathbf{rlx}} Q(t',s') & b = \mathbf{true} \end{array} \right. \right\}_{\mathcal{E}}$$

FIGURE 17.2: CAS Rules for GPS Persistent Concurrent Protocols

write rules, we use the conditional notations $\Delta_\pi^{o?\texttt{rlx}}$ and $\nabla_\pi^{o?\texttt{rlx}}$ to denote these requirements.

- Putting the usual pre-condition of the protocol assertion aside, the first pre-condition requires the user to show, objectively, that any message (t', s', v') the CAS can read from would have the value v' that is *comparable* with the expected integer v_r, *i.e.*, $\vdash v' =^? v_r$.[6] In proving comparability, the user can assume more resources, specifically the read or write interpretation about the read message (t', s', v').

[6] see Definition 4.9

- As the second pre-condition, the user is required to pick and prove some predicate P—either now or at the most recent release fence—that will be released with the write effect if the CAS succeeds ($b = \texttt{true}$ in the post-condition). In case the CAS fails ($b = \texttt{false}$), P is returned unchanged in the post-condition.

- The last pre-condition, also needed to be proven either now or at the most recent release fence depending on o_w, requires that the user can resolve two obligations that represent the failure and success cases of the CAS, and that consequently are tied together by the classical conjunction $\wedge$.

  (i) If the CAS fails, the situation is similar to that of a read with the mode o_f: the user should prove a fancy update $\mathsf{Extract}_{\mathcal{I}}$ that can extract the resource $R(t', s', v')$ from either the read ($\mathcal{I}_r$) or write ($\mathcal{I}_w$) interpretation for the read message t', whose value v' is definitely not the expected integer v_r.

  (ii) If the CAS succeeds, it has a combined effect of both a read of the message (t', s', v_r) *and* a write of the message (t'', s'', v_w). In this obligation, the user can consume the write interpretation $\mathcal{I}_w$ of the *read* message to produce the write interpretation $\mathcal{I}_w$ of the new write message. The obligation is split into several steps:

  - first, the user shows, objectively, that the write interpretation $\mathcal{I}_w$ of the read message (t', s', v_r) can be split into two resources Q_1 and Q_2;
  - second, the user shows that, with the pre-condition P and the 2nd part Q_2, the user can *pick* the new state s'' that extends s' for the new write of timestamp t'';
  - finally; knowing the observation $\mathcal{R}(\ell, \gamma, t'', s'', v_w)$ that the protocol has been extended with the new state, the user proves both (a) the read interpretation $\mathcal{I}_r$ for the read message, assuming Q_1, and (b) the write interpretation $\mathcal{I}_w$ for the new write message (t'', s'', v_w), assuming the rest of the resources. If there are still resources left, the user can choose to keep them in Q, which will be returned in the post-condition when the CAS succeeds ($b = \texttt{true}$).

- The laters and fancy updates are put in positions that provide maximum flexibility for the user in accessing invariants, performing ghost updates, and eliminating laters.

- The timestamp t' and state s' returned in the post-condition correspond, in the CAS failure case, to the read message's timestamp t' and s', and in the CAS success case, to the write message's timestamp t'' and state s''.

The rule GPS-CON-CAS-LOC (Figure 17.2) is very similar to GPS-CON-CAS-INT, except that it allows to use a *pointer* (location) ℓ_r as the expected value for the comparison. Consequently, the rule has extra obligations to achieve deterministic pointer comparison. Specifically, the rule now requires, as the first condition, some resource P_{cmp} that can be used to show that ℓ_r is alive, by showing its primitive atomic points-to $\ell_r \mapsto h_r$, subjectively. The resource P_{cmp}, however, is only used to for this purpose and will not be consumed, and will be returned unchanged in the post-condition. Furthermore, the last pre-condition requires the user to show that the location value ℓ' that is to be compared with ℓ_r is also alive, also by showing its primitive atomic points-to. Note that, interestingly, this requirement only applies for ℓ' and ℓ_r that are *definitionally* inequal, because definitionally equal locations are always compared equal and never inequal,[7] regardless of their liveness status.

[7] see Definition 4.7 and Definition 4.8

17.1.2 *Cancelable Single-Writer Protocols*

As can be seen from GPS-CON-WRITE, allowing arbitrarily concurrent writes results in a weak write rule which is often only applicable to a trivial protocol order, because it is impossible to know what the current up-to-date state of the protocol is. In more interesting protocols, one needs more control on how states can change, *i.e.*, how writes can happen. To provide stronger reasoning principles for some of such interesting scenarios, iGPS[8] introduces *single-writer* protocols, where only one thread can write to the location, while other threads can concurrently read from it. Intuitively, the setup is that writes can only be performed with the unique *writer permission*, which must be transferred explicitly among multiple threads to write to it, one at a time. As such, the writer always knows exactly what the current state of the protocol is.

[8] Kaiser et al., "Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris" [Kai+17].

iRC11's single-writer protocols build upon those of iGPS, but extends the notion of single-writer to also include *CAS-only* accesses: when multiple threads concurrently try to write to the same location ℓ, if they can resolve their contentions by using only CASes, then the protocol still maintains a single writer, albeit only *atomically*. In the following, we present the *cancelable* variant of iRC11 single-writer protocols, *i.e.*, they also allow the user to switch protocols or switch to non-atomic reasoning if need be.

Definition 17.4 (Cancelable Single-Writer Protocol Assertions)**.** There are four kinds of assertions.

- The reader assertion $\boxed{(\ell, \gamma_i, \gamma) : (t, s, v) \mid \mathcal{I}}^{\mathcal{N}}$ says that a cancelable protocol instance γ with the interpretation $\mathcal{I}$ has been established for the location ℓ, under the namespace $\mathcal{N}$. It is similar to the assertion $\boxed{(\ell, \gamma) : (t, s, v) \mid \mathcal{I}}^{\mathcal{N}}$ of persistent concurrent protocols,[9] as it also says that the current thread has observed the write message of timestamp t, value v, and state s of the protocol instance γ. However, it says nothing about the *liveness* of that instance.

[9] see Definition 17.3

- The cancelable token $\heartsuit_q^{\gamma_i}$ asserts that the protocol has not been canceled, *i.e.*, it is still alive. The token is tied to other protocol assertions by the ghost location γ_i, and like the cancelable invariant token,[10] it is needed for any access to the location ℓ through the protocol instance γ that is tied to γ_i.[11]

[10] see §11.2.1

[11] Consequently, the pair (γ_i, γ) uniquely identifies the protocol instance, and we should always use the pair as a single name for the instance. However, here we choose to state the pair explicitly, to easily show the similarity with the persistent concurrent protocol assertion (through γ) and the cancelable invariant assertions (through γ_i).

- The single-writer assertion $\boxed{(\ell, \gamma_i, \gamma) : (t, s, v) \mid \mathcal{I}}^{\mathcal{N}}_{W}$ asserts the unique permission to perform writes on ℓ. It additionally says that the latest write to ℓ is exactly the message (t, s, v), and that the owner thread has observed that write.

- The CAS-only assertion $\boxed{(\ell, \gamma_i, \gamma) : (t, s, v) \mid \mathcal{I}}^{\mathcal{N}}_{q}$ asserts the permission to perform CASes on ℓ. The assertion is fractional, as it is indexed by a fraction q. Therefore it can be split and joined, so as to allow multiple threads to concurrently perform CASes on ℓ. However, the CAS-only assertion cannot co-exist with the single-writer assertion, enforcing that the protocol is in the CAS-only mode, where writes can only be done with CASes.

- Last but not least, in order to support a strong single-writer rule, we need to extend the protocol interpretation $\mathcal{I}$ into a triple of predicates $(\mathcal{I}_r, \mathcal{I}_w, \mathcal{I}_m)$ which are the read, write, and *moved* interpretations, respectively. The new moved interpretation $\mathcal{I}_m$ denotes the *leftover* resources of a write message that has been overwritten by a single-writer write. We will see the use of this new interpretation in the single-writer write rule GPS-SW-WRITE-REL in Figure 17.4.

Some of the relations among the assertions can be found in Figure 17.3. For example, GPS-SW-W and GPS-SW-RSHR respectively show that the single-writer assertion and the CAS-only assertion both imply the reader assertion, so they can both be used to read from the location ℓ. GPS-SW-W-WSHR-RSHR shows how to convert the single-writer assertion into the full fraction of the CAS-only assertion, effectively switching the protocol from the single-writer mode to the CAS-only mode. Dually, GPS-SW-W-REVERT shows how to convert the full fraction of the CAS-only assertion back to the single-writer assertion. To fully understand the rules in Figure 17.3, we need to introduce some new auxillary assertions.

Definition 17.5 (Auxillary Single-Writer Protocol Assertions)**.** The auxillary assertions are all pure ghost-state assertions, so they are *timeless* and can be put into invariants and can bypass all step-indexing restrictions (manifested in the later modality). In fact, they are only needed specifically for this purpose.

$\mathsf{persistent}(\mathcal{R}(\ell,\gamma,t,s,v))$
$\mathsf{timeless}(\mathcal{R}(\ell,\gamma,t,s,v))$
$\mathsf{persistent}(\boxed{(\ell,\gamma_i,\gamma):(t,s,v)\mid\mathcal{I}}^{\mathcal{N}})$

$\mathsf{timeless}(\mathcal{W}(\ell,\gamma,t,s,v))$
$\mathsf{timeless}(\mathcal{W}_{\mathsf{shr}}(\ell,\gamma,t,s,v))$
$\mathsf{timeless}(\mathcal{R}^q_{\mathsf{shr}}(\ell,\gamma,t,s,v))$

GPS-SW-W
$$\boxed{(\ell,\gamma_i,\gamma):(t,s,v)\mid\mathcal{I}}^{\mathcal{N}}_{W} \dashv\vdash \mathcal{W}(\ell,\gamma,t,s,v) * \boxed{(\ell,\gamma_i,\gamma):(t,s,v)\mid\mathcal{I}}^{\mathcal{N}}$$

GPS-SW-RSHR
$$\boxed{(\ell,\gamma_i,\gamma):(t,s,v)\mid\mathcal{I}}^{\mathcal{N}}_{q} \dashv\vdash \mathcal{R}^q_{\mathsf{shr}}(\ell,\gamma,t,s,v) * \boxed{(\ell,\gamma_i,\gamma):(t,s,v)\mid\mathcal{I}}^{\mathcal{N}}$$

GPS-SW-R
$$\boxed{(\ell,\gamma_i,\gamma):(t,s,v)\mid\mathcal{I}}^{\mathcal{N}} \vdash \mathcal{R}(\ell,\gamma,t,s,v)$$

GPS-W-R
$\mathcal{W}(\ell,\gamma,t,s,v) \vdash \mathcal{R}(\ell,\gamma,t,s,v)$

GPS-WSHR-R
$\mathcal{W}_{\mathsf{shr}}(\ell,\gamma,t,s,v) \vdash \mathcal{R}(\ell,\gamma,t,s,v)$

GPS-RSHR-R
$\mathcal{R}^q_{\mathsf{shr}}(\ell,\gamma,t,s,v) \vdash \mathcal{R}(\ell,\gamma,t,s,v)$

GPS-W-EXCL
$\mathcal{W}(\ell,\gamma,t,s,v) * \mathcal{W}(\ell,\gamma,t',s',v') \vdash \mathsf{False}$

GPS-WSHR-EXCL
$\mathcal{W}_{\mathsf{shr}}(\ell,\gamma,t,s,v) * \mathcal{W}_{\mathsf{shr}}(\ell,\gamma,t',s',v') \vdash \mathsf{False}$

GPS-W-WSHR-EXCL
$\mathcal{W}(\ell,\gamma,t,s,v) * \mathcal{W}_{\mathsf{shr}}(\ell,\gamma,t',s',v') \vdash \mathsf{False}$

GPS-W-RSHR-EXCL
$\mathcal{W}(\ell,\gamma,t,s,v) * \mathcal{R}^q_{\mathsf{shr}}(\ell,\gamma,t',s',v') \vdash \mathsf{False}$

GPS-RSHR-FRAC-1
$\mathcal{R}^q_{\mathsf{shr}}(\ell,\gamma,t,s,v) * \langle\mathsf{subj}\rangle\, \mathcal{R}^{q'}_{\mathsf{shr}}(\ell,\gamma,t',s',v') \vdash \mathcal{R}^{q+q'}_{\mathsf{shr}}(\ell,\gamma,t,s,v)$

GPS-RSHR-FRAC-2
$\mathcal{R}^{q+q'}_{\mathsf{shr}}(\ell,\gamma,t,s,v) \vdash \mathcal{R}^q_{\mathsf{shr}}(\ell,\gamma,t,s,v) * \mathcal{R}^{q'}_{\mathsf{shr}}(\ell,\gamma,t,s,v)$

GPS-R-RSHR-JOIN
$\mathcal{R}(\ell,\gamma,t,s,v) * \langle\mathsf{subj}\rangle\, \mathcal{R}^q_{\mathsf{shr}}(\ell,\gamma,t',s',v') \vdash \mathcal{R}^q_{\mathsf{shr}}(\ell,\gamma,t,s,v)$

GPS-WSHR-RSHR-UPDATE
$\mathcal{W}_{\mathsf{shr}}(\ell,\gamma,t,s,v) * \mathcal{R}^q_{\mathsf{shr}}(\ell,\gamma,t',s',v') \vdash \mathcal{W}_{\mathsf{shr}}(\ell,\gamma,t,s,v) * \mathcal{R}^q_{\mathsf{shr}}(\ell,\gamma,t,s,v)$

GPS-W-WSHR-RSHR
$\mathcal{W}(\ell,\gamma,t,s,v) \vdash \mathcal{W}_{\mathsf{shr}}(\ell,\gamma,t,s,v) * \mathcal{R}^1_{\mathsf{shr}}(\ell,\gamma,t,s,v)$

GPS-SW-W-WSHR-RSHR
$$\boxed{(\ell,\gamma_i,\gamma):(t,s,v)\mid\mathcal{I}}^{\mathcal{N}}_{W} \vdash \mathcal{W}_{\mathsf{shr}}(\ell,\gamma,t,s,v) * \boxed{(\ell,\gamma_i,\gamma):(t,s,v)\mid\mathcal{I}}^{\mathcal{N}}_{1}$$

GPS-SW-W-REVERT
$$\frac{\uparrow\mathcal{N} \subseteq \mathcal{E}}{\heartsuit^{\gamma_i}_q * \mathcal{W}_{\mathsf{shr}}(\ell,\gamma,t,s,v) * \boxed{(\ell,\gamma_i,\gamma):(t',s',v')\mid\mathcal{I}}^{\mathcal{N}}_{1} \Rrightarrow_{\mathcal{E}} \heartsuit^{\gamma_i}_q * \boxed{(\ell,\gamma_i,\gamma):(t,s,v)\mid\mathcal{I}}^{\mathcal{N}}_{W}}$$

FIGURE 17.3: Rules for auxillary assertions of GPS Single-Writer Protocols

- The read observation $\mathcal{R}(\ell, \gamma, t, s, v)$, which we already see in the rule GPS-CON-WRITE, is persistent and asserts that the message (t, s, v) has been registered in the protocol instance γ.

- The single-writer permission $\mathcal{W}(\ell, \gamma, t, s, v)$ is the unique ghost permission needed to perform writes when the protocol is in single-writer mode.

- The shared-writer permission $\mathcal{W}_{\text{shr}}(\ell, \gamma, t, s, v)$ and the shared-reader permission $\mathcal{R}^q_{\text{shr}}(\ell, \gamma, t, s, v)$ are needed to perform CASes when the protocol is in CAS-only mode. Intuitively, every participant locally owns a fraction of the shared-reader permission and globally shares the shared-writer permission in order to perform CASes.

We can now explain the rules in Figure 17.3 in more details. GPS-SW-W says explicitly that the single-writer assertion is in fact the ghost single-writer permission combined with the reader assertion—the ghost single-writer permission maintains that the protocol is in the single-writer mode and that the observed message (t, s, v) is the latest write. Meanwhile GPS-SW-RSHR says that the CAS-only assertion is the ghost shared-reader permission combined with the reader assertion—the ghost shared-reader permission maintains that the protocol is still in CAS-only mode. Furthermore, GPS-SW-R says that the reader assertion simply implies the read observation (in addition to the fact that an invariant exists for the protocol). The rules GPS-W-R, GPS-WSHR-R, and GPS-RSHR-R say that all ghost permissions always include the read observation of their message (t, s, v).

GPS-W-EXCL and GPS-WSHR-EXCL respectively say that the ghost single-writer permission and shared-writer permission are exclusive. GPS-W-WSHR-EXCL and GPS-W-RSHR-EXCL say that the single-writer permission cannot co-exist with the shared-writer permission or the shared-reader permission, because the protocol can only be either in the single-writer mode or in the CAS-only mode at a time. GPS-RSHR-FRAC-1 and GPS-RSHR-FRAC-2 say that the shared-reader permission can be split and join fractionally. GPS-R-RSHR-JOIN says that we can update the shared-reader permission $\mathcal{R}^q_{\text{shr}}(\ell, \gamma, t', s', v')$ with the local observation $\mathcal{R}q(\ell, \gamma, t, s, v)$. Meanwhile, GPS-WSHR-RSHR-UPDATE says that we can update the observation of the shared-reader permission with that of the shared-writer permission, which always carries the latest write to the protocol.

The rule GPS-W-WSHR-RSHR crucially shows how we can convert the single-writer permission into the shared-writer permission and the full fraction of the shared-reader permission. This is the key rule to switch the protocol from the single-writer mode to the CAS-only mode, and in fact it establishes the soundness of GPS-SW-W-WSHR-RSHR, which applies the switch at the level of protocol assertions. The reverse switch—from CAS-only mode back to single-writer mode—is stated in GPS-SW-W-REVERT and additionally requires a fancy update as well as a fraction of cancelable token $\heartsuit^{\gamma_i}_q$. We need these two facilities because the reverse switch needs to access the internal invariant that supports the model of cancelable

single-writer protocols. We will explain this in more details when we look at the model of cancelable single-writer protocols in §17.2.

We now turn our eyes to an excerpt of the *operational* rules for cancelable single-writer protocols in Figure 17.4.

- GPS-SW-INIT allows us to allocate a new cancelable single-writer GPS protocol for ℓ provided the non-atomic points-to $\ell \mapsto v$ and the write interpretation $\mathcal{I}_w$ for the latest message (t, s, v). The rule is very similar to the rule GPS-CON-INIT of persistent concurrent protocols, but in this case we get back the single-writer assertion as well as the *full* cancelable token $\heartsuit_1^{\gamma_i}$.

- GPS-SW-READ shows how we can read from the location using only the reader assertion. Again, this is almost the same as GPS-CON-READ, except that we need a fraction $\heartsuit_q^{\gamma_i}$ of the token to know that the protocol instance is still alive. GPS-SW-WRITER-READ shows a stronger read rule with the single-writer assertion: we always read the latest write message.

- GPS-SW-WRITE-REL is a release write rule with the single-writer assertion. The rule is much stronger than the concurrent write rule GPS-CON-WRITE,[12] because it gets access to the latest write. More specifically, we can use the write interpretation $\mathcal{I}_w$ of the latest write (t, s, v) to establish the write interpretation $\mathcal{I}_w$ of the *next* latest write (t', s', v'), as follows.

  - The protocol state must grow, *i.e.*, $s \sqsubseteq s'$.
  - $\mathcal{I}_w$ of (t, s, v) can be split into Q_1 and Q_2, where Q_1 is the *leftover* resource that can be used to establish the new *moved* interpretation $\mathcal{I}_m$ of the overwritten message (t, s, v).
  - Q_2 can be used to establish the write interpretation $\mathcal{I}_w$ of the next latest write (t', s', v').
  - Additionally, the ghost writer permission $\mathcal{W}(\ell, \gamma, t', s', v')$ (updated with the next latest write) as well as the cancelable token $\heartsuit_q^{\gamma_i}$ (needed to know that the protocol is still alive) can also be used for $\mathcal{I}_w$ of the next write (t', s', v').
  - Any unused resources can be returned in $Q(t')$. For example, if the ghost writer permission $\mathcal{W}(\ell, \gamma, t', s', v')$ is returned in $Q(t')$, we can get back the writer assertion with the updated latest write, *i.e.*, $\boxed{(\ell, \gamma_i, \gamma) : (t', s', v') \mid \mathcal{I}}^{\mathcal{N}}_{W}$. The same also applies to the cancelable fraction $\heartsuit_q^{\gamma_i}$.

- GPS-SW-DEALLOC is the cancelation rule for the single-writer protocol. It is derived from CINV-CANCEL,[13] so it naturally requires the *full* cancelable token. The rule then will return the full non-atomic points-to of the location, as well as the write interpretation $\mathcal{I}_w$ for the latest write. Note that it is sufficient to have only the reader assertion of the protocol to cancel it. However, we also have a variant of the rule (elided here) with the writer assertion, where we know that we will get back the non-atomic points-to with v

[12] see Figure 17.1. In fact, it has the flavor of a CAS rule, like GPS-CON-CAS-INT (in Figure 17.2).

[13] see Chapter 11

GPS-SW-INIT

$$\ell \mapsto v * (\forall \gamma_i, \gamma, t.\ \triangleright \mathcal{I}_w(\ell, \gamma_i, \gamma, t, s, v)) \Rrightarrow_{\mathcal{E}} \exists \gamma_i, \gamma, t.\ \heartsuit_1^{\gamma_i} * \boxed{(\ell, \gamma_i, \gamma) : (t, s, v) \mid \mathcal{I}}^{\mathcal{N}}_{W}$$

$$\mathsf{Extract}_{\mathcal{I}}(\ell, \gamma_i, \gamma, t', s', v', R, \mathcal{E}, \mathcal{N}) ::= \langle \mathsf{obj} \rangle \wedge \begin{cases} \mathcal{I}_r(\ell, \gamma_i, \gamma, t', s', v') \Rrightarrow_{\mathcal{E} \setminus \uparrow \mathcal{N}} \mathcal{I}_r(\ell, \gamma_i, \gamma, t', s', v') * R(t', s', v') \\ \mathcal{I}_w(\ell, \gamma_i, \gamma, t', s', v') \Rrightarrow_{\mathcal{E} \setminus \uparrow \mathcal{N}} \mathcal{I}_w(\ell, \gamma_i, \gamma, t', s', v') * R(t', s', v') \\ \mathcal{I}_m(\ell, \gamma_i, \gamma, t', s', v') \Rrightarrow_{\mathcal{E} \setminus \uparrow \mathcal{N}} \mathcal{I}_m(\ell, \gamma_i, \gamma, t', s', v') * R(t', s', v') \end{cases}$$

GPS-SW-READ

$$\frac{\mathbf{rlx} \sqsubseteq o \qquad \uparrow \mathcal{N} \subseteq \mathcal{E}}{\left\{ \heartsuit_q^{\gamma_i} * \boxed{(\ell, \gamma_i, \gamma) : (t, s, v) \mid \mathcal{I}}^{\mathcal{N}} * (\forall t' \geq t, s' \sqsupseteq s, v'.\ \mathsf{Extract}_{\mathcal{I}}(\ell, \gamma_i, \gamma, t', s', v', R, \mathcal{E}, \mathcal{N})) \right\} \ {}^{*o}\ell \ \mathsf{in}\ \pi \ \left\{ v'.\ \exists t' \geq t, s' \sqsupseteq s.\ \heartsuit_q^{\gamma_i} * \boxed{(\ell, \gamma_i, \gamma) : (t', s', v') \mid \mathcal{I}}^{\mathcal{N}} * \nabla_{\pi}^{o ? \mathbf{rlx}} R(t', s', v') \right\}_{\mathcal{E}}}$$

GPS-SW-WRITER-READ

$$\frac{\mathbf{rlx} \sqsubseteq o \qquad \uparrow \mathcal{N} \subseteq \mathcal{E}}{\left\{ \heartsuit_q^{\gamma_i} * \boxed{(\ell, \gamma_i, \gamma) : (t, s, v) \mid \mathcal{I}}^{\mathcal{N}}_{W} * \langle \mathsf{obj} \rangle \left(\mathcal{I}_w(\ell, \gamma_i, \gamma, t, s, v) \Rrightarrow_{\mathcal{E} \setminus \uparrow \mathcal{N}} \mathcal{I}_w(\ell, \gamma_i, \gamma, t, s, v) * R \right) \right\} \ {}^{*o}\ell \ \mathsf{in}\ \pi \ \left\{ v.\ \heartsuit_q^{\gamma_i} * \boxed{(\ell, \gamma_i, \gamma) : (t, s, v) \mid \mathcal{I}}^{\mathcal{N}}_{W} * R \right\}_{\mathcal{E}}}$$

GPS-SW-WRITE-REL

$$\frac{\uparrow \mathcal{N} \subseteq \mathcal{E} \qquad s \sqsubseteq s'}{\left\{ \begin{array}{l} \heartsuit_q^{\gamma_i} * \boxed{(\ell, \gamma_i, \gamma) : (t, s, v) \mid \mathcal{I}}^{\mathcal{N}}_{W} * \triangleright \langle \mathsf{obj} \rangle \left(\mathcal{I}_w(\ell, \gamma_i, \gamma, t, s, v) \Rrightarrow_{\mathcal{E} \setminus \uparrow \mathcal{N}} Q_1 * Q_2 \right) * \\ \forall t' > t.\ \left(\mathcal{W}(\ell, \gamma, t', s', v') * Q_2 * \heartsuit_q^{\gamma_i} \right) \Rrightarrow_{\mathcal{E} \setminus \uparrow \mathcal{N}} * \begin{cases} \langle \mathsf{obj} \rangle\, Q_1 \Rrightarrow_{\mathcal{E} \setminus \uparrow \mathcal{N}} \mathcal{I}_m(\ell, \gamma_i, \gamma, t, s, v) \\ \models\!\!\Rrightarrow_{\mathcal{E} \setminus \uparrow \mathcal{N}} \mathcal{I}_w(\ell, \gamma_i, \gamma, t', s', v') * Q(t') \end{cases} \end{array} \right\} \ \ell :=_{\mathbf{rel}} v' \ \mathsf{in}\ \pi \ \left\{ \text{☠}.\ \exists t' > t. \boxed{(\ell, \gamma_i, \gamma) : (t', s', v') \mid \mathcal{I}}^{\mathcal{N}} * Q(t') \right\}_{\mathcal{E}}}$$

GPS-SW-DEALLOC

$$\frac{\uparrow \mathcal{N} \subseteq \mathcal{E}}{\heartsuit_1^{\gamma_i} * \boxed{(\ell, \gamma_i, \gamma) : (t', s', v') \mid \mathcal{I}}^{\mathcal{N}} \Rrightarrow_{\mathcal{E}} \exists t, s, v.\ \ell \mapsto v * \triangleright \mathcal{I}_w(\ell, \gamma_i, \gamma, t, s, v)}$$

GPS-SW-READ-ACQ-DEALLOC

$$\frac{\mathbf{rlx} \sqsubseteq o \qquad \uparrow \mathcal{N} \subseteq \mathcal{E}}{\left\{ \begin{array}{l} \heartsuit_q^{\gamma_i} * \boxed{(\ell, \gamma_i, \gamma) : (t, s, v) \mid \mathcal{I}}^{\mathcal{N}} * (\forall t' \geq t, s' \sqsupseteq s, v'.\ \mathsf{Extract}_{\mathcal{I}}(\ell, \gamma_i, \gamma, t', s', v', R, \mathcal{E}, \mathcal{N})) * \\ P * \left(\forall t' \geq t, s' \sqsupseteq s, v'.\ \left(P * \heartsuit_q^{\gamma_i} * R(t', s', v_d) \right) \Rrightarrow_{\mathcal{E} \setminus \uparrow \mathcal{N}} \heartsuit_1^{\gamma_i} * Q(t', s') \right) \end{array} \right\} \ {}^{*\mathbf{acq}}\ell \ \mathsf{in}\ \pi \ \left\{ v'.\ \exists t' \geq t, s' \sqsupseteq s.\ \boxed{(\ell, \gamma_i, \gamma) : (t', s', v') \mid \mathcal{I}}^{\mathcal{N}} * \begin{cases} Q(t', s') * \exists t, s, v.\ \ell \mapsto v * \triangleright \mathcal{I}_w(\ell, \gamma_i, \gamma, t, s, v) & v' = v_d \\ \heartsuit_q^{\gamma_i} * R(t', s', v') * P & v' \neq v_d \end{cases} \right\}_{\mathcal{E}}}$$

FIGURE 17.4: Selected rules for Cancelable Single-Writer GPS Protocols

being the latest write from the writer assertion (instead of being existentially quantified here).

- GPS-SW-READ-ACQ-DEALLOC is a very specific read rule that extends GPS-SW-READ with the ability to cancel the protocol if it happens to be the case that the reader is the *only* party accessing the protocol. To do so, the user first picks a special value v_d that signals the end-of-life of the protocol. That is, the reader should prepare to cancel the protocol if it happens to read v_d. Specifically, the reader should provide as extra pre-conditions (1) some resource P and (2) a proof that P together with the cancelable fraction $\heartsuit_q^{\gamma_i}$ (used for the read) *and* the acquired resource $R(t', s', v_d)$ can be used to reconstruct the full cancelable token $\heartsuit_1^{\gamma_i}$. In that case, the cancelation is performed in the same way as GPS-SW-DEALLOC: we get back the full non-atomic points-to of the location and the latest write interpretation $\mathcal{I}_w$. Furthermore, we also get out whatever leftover resource $Q(t', s')$ that comes out of (2). In case that we do not read the signal value v_d, we simply get back the acquired resource $R(t', s', v')$ with the original P and the cancelable fraction.

17.1.3 *Atomic-Borrows-based Protocols*

Finally, we introduce a variant of GPS single-writer protocols that is tied to a lifetime κ, who effectively is the upper-bound of the protocol's lifetime.

Definition 17.6 (Atomic-Borrows-based Single-Writer Protocol Assertions)**.** There are three kinds of assertions that mirror those of cancelable single-writer protocols. They all simply tie the protocol to a lifetime κ, instead of to a ghost location γ_i for the cancelable token.

- The reader assertion $\boxed{\&^{\kappa}\,(\ell, \gamma) : (t, s, v) \mid \mathcal{I}}^{\mathcal{N}}$
- The single-writer assertion $\boxed{\&^{\kappa}\,(\ell, \gamma) : (t, s, v) \mid \mathcal{I}}^{\mathcal{N}}_{W}$
- The CAS-only assertion $\boxed{\&^{\kappa}\,(\ell, \gamma) : (t, s, v) \mid \mathcal{I}}^{\mathcal{N}}_{q}$

We present only a small selection of rules for atomic-borrows-based protocols in Figure 17.5 and Figure 17.6.

- GPS-ATBOR-R, GPS-ATBOR-W, and GPS-ATBOR-RSHR are similar to the rules GPS-SW-R, GPS-SW-W, and GPS-SW-RSHR in Figure 17.3, respectively. They show the relations among the protocol assertions and the ghost permissions (introduced in Definition 17.5).
- GPS-ATBOR-W-WSHR-RSHR is the atomic-borrow-based variant of GPS-SW-W-WSHR-RSHR, which shows how to convert the single-writer assertion into the full fraction of the CAS-only assertion, effectively switching the protocol from the single-writer mode to the CAS-only mode.

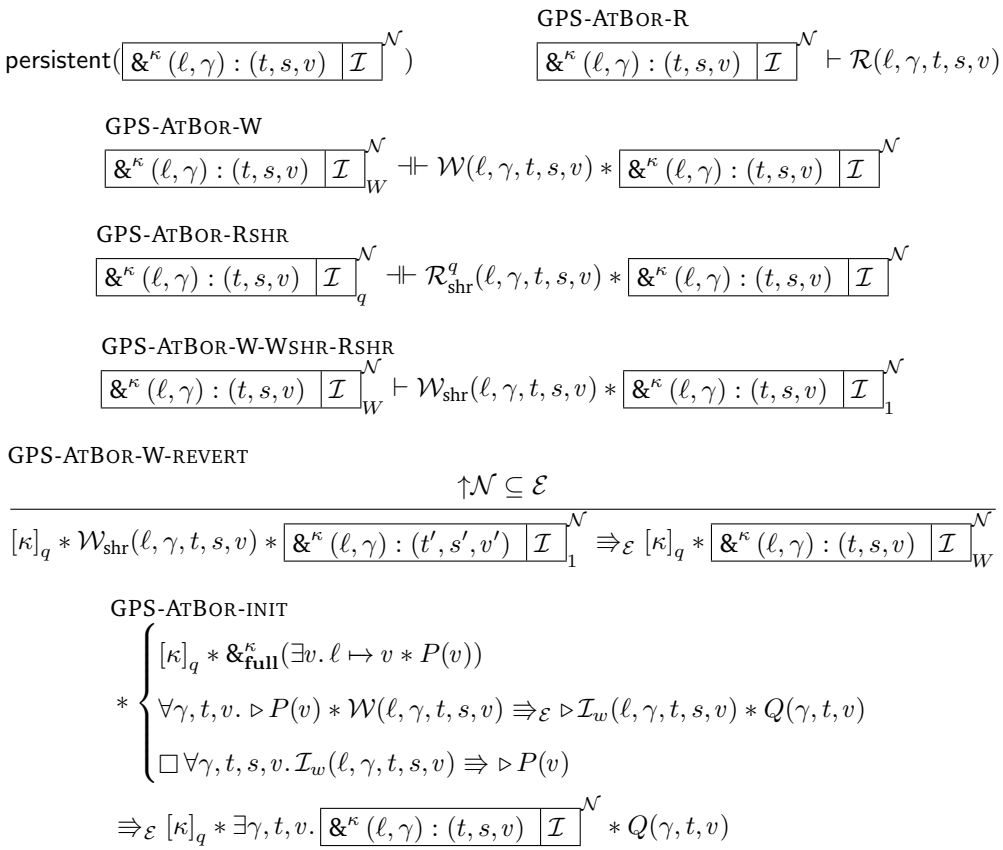


FIGURE 17.5: Selected basic rules for Atomic-Borrows-based GPS Protocols

- The reverse rule GPS-AtBor-W-revert, like GPS-SW-W-revert, shows how to switch the protocol from CAS-only mode back to single-writer mode. Similar to that of cancelable single-writer protocols, this reverse switch needs to know the internal invariant that supports the model of GPS protocols is still alive, so it needs the fancy update and a fraction of the lifetime token. (If the lifetime is still alive, then the invariant is also still alive.)

- GPS-AtBor-init allows us to initialize a atomic-borrows-based protocol for ℓ. The rule is similar to the rule GPS-SW-init of cancelable protocols,[14] but here we cannot simply allocate a new cancelable invariant (and so a cancelable token) for every new protocol instance. Instead, we need to tie the new protocol instance to some existing lifetime κ, and therefore we need to integrate the more complex (but also more general) management of lifetimes and borrows into the rule.

  - First, we need to know that the lifetime κ is alive, so we require a fraction of the lifetime token $[\kappa]_q$, which will we return after the initialization.

  - Second, like in GPS-SW-init, we need the non-atomic points-to $\ell \mapsto v$ and the write interpretation $\mathcal{I}_w$ for the write message (t, s, v). However, unlike GPS-SW-init, we make one general-

[14] see Figure 17.4

ization, and we must establish an atomic borrow tied to κ for the new protocol instance.

– The generalization is the first fancy update where we allow the write interpretation $\mathcal{I}_w$ to use the single-writer ghost permission $\mathcal{W}(\ell, \gamma, t, s, v)$ immediately at initialization point. Any unused resources, which can possibly include the writer permission, is returned in $Q(\gamma, t, v)$ after the initialization, together with the read assertion $\boxed{\&^{\kappa}\,(\ell,\gamma) : (t,s,v) \mid \mathcal{I}}^{\mathcal{N}}$. If the writer permission $\mathcal{W}(\ell, \gamma, t, s, v)$ is indeed unused for $\mathcal{I}_w$ and returned in $Q(\gamma, t, v)$, then after the initialization the user can get the single-writer assertion $\boxed{\&^{\kappa}\,(\ell,\gamma) : (t,s,v) \mid \mathcal{I}}^{\mathcal{N}}_{W}$, using GPS-AtBor-W.[15]

– To create an atomic borrow for our protocol, we need to use LftL-full-at [16] to turn a *full* borrow into an atomic borrow. As such, our rule here requires that the needed resources have been put into a full borrow, *i.e.*, $\&^{\kappa}_{\mathbf{full}}(\exists v.\, \ell \mapsto v * P(v))$. However, before applying LftL-full-at to create an atomic borrow, we need to transform the non-atomic points-to into the protocol's resources *under* a full borrow. For that purpose, we need to apply the strong access rule LftL-full-acc-strong and then show that the transformation can be reversed, *i.e.*, from the protocol resources we can get back the original resources. Since the rule itself takes care of giving back the non-atomic points-to, it only requires the user to show the remaining obligation, in the second fancy update, that the write interpretation $\mathcal{I}_w(\ell, \gamma, t, s, v)$ can be used to reconstruct the original resource $P(v)$.

[15] This generalization has also been applied in GPS-SW-write-rel in Figure 17.4, and will also be applied in GPS-AtBor-write-rel in Figure 17.6.

[16] see Figure 16.2

Intuitively, this requirement in LftL-full-acc-strong is to ensure the lifetime inheritance rule LftL-full-inh,[17] where we need to maintain that once the lifetime is dead, we can reclaim the original resource picked when the borrow is created with LftL-full-bor. As such, any update to a borrow's resource needs to show that we can go back to the original resource. Effectively, we see that GPS-AtBor-init has the flavor of both an initialization rule (like GPS-SW-init) *and* a cancelation rule (like GPS-SW-dealloc). Specifically, atomic-borrows-based protocols do not have an actual cancelation rule. Instead, once the lifetime κ is dead, the user will reclaim the resource $\exists v.\, \ell \mapsto v * P(v)$ that they originally put into the full borrow, regardless of whether that full borrow has ever been used to construct a GPS protocol.

[17] see Figure 15.1

- GPS-AtBor-read and GPS-AtBor-write-rel are essentially the same as their cancelable protocol counterparts GPS-SW-read and GPS-SW-write-rel in Figure 17.4, respectively. The only difference is that they require a fraction of the lifetime token $[\kappa]_q$ instead of a fraction of the cancelable token.

$$\mathsf{Extract}_{\mathcal{I}}(\ell,\gamma,t',s',v',R,\mathcal{E},\mathcal{N}) ::= \langle \mathsf{obj}\rangle \wedge \begin{cases} \mathcal{I}_r(\ell,\gamma,t',s',v') \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \mathcal{I}_r(\ell,\gamma,t',s',v') * R(t',s',v') \\ \mathcal{I}_w(\ell,\gamma,t',s',v') \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \mathcal{I}_w(\ell,\gamma,t',s',v') * R(t',s',v') \\ \mathcal{I}_m(\ell,\gamma,t',s',v') \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \mathcal{I}_m(\ell,\gamma,t',s',v') * R(t',s',v') \end{cases}$$

GPS-AtBor-read

$$\frac{\mathtt{rlx} \sqsubseteq o \qquad \uparrow\mathcal{N} \subseteq \mathcal{E}}{\left\{ [\kappa]_q * \boxed{\&^{\kappa}\,(\ell,\gamma) : (t,s,v) \;\middle|\; \mathcal{I}}^{\mathcal{N}} * (\forall t' \geq t, s' \sqsupseteq s, v'.\, \mathsf{Extract}_{\mathcal{I}}(\ell,\gamma,t',s',v',R,\mathcal{E},\mathcal{N})) \right\}}$$
$$*^{o}\ell \text{ in } \pi$$
$$\left\{ v'.\, \exists t' \geq t, s' \sqsupseteq s.\, [\kappa]_q * \boxed{\&^{\kappa}\,(\ell,\gamma) : (t',s',v') \;\middle|\; \mathcal{I}}^{\mathcal{N}} * \nabla_{\pi}^{o?\mathtt{rlx}} R(t',s',v') \right\}_{\mathcal{E}}$$

GPS-AtBor-write-rel

$$\frac{\uparrow\mathcal{N} \subseteq \mathcal{E} \qquad s \sqsubseteq s'}{\left\{ \begin{array}{l} [\kappa]_q * \boxed{\&^{\kappa}\,(\ell,\gamma) : (t,s,v) \;\middle|\; \mathcal{I}}^{\mathcal{N}}_{W} * \triangleright \langle \mathsf{obj}\rangle \left(\mathcal{I}_w(\ell,\gamma,t,s,v) \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} Q_1 * Q_2\right) * \\ \forall t' > t.\ (\mathcal{W}(\ell,\gamma,t',s',v') * Q_2) \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} * \begin{cases} \langle \mathsf{obj}\rangle\, Q_1 \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \mathcal{I}_m(\ell,\gamma,t,s,v) \\ \Rightarrow\!\!\!\!\Rightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \mathcal{I}_w(\ell,\gamma,t',s',v') * Q(t') \end{cases} \end{array} \right\}}$$
$$\ell :=_{\mathtt{rel}} v' \text{ in } \pi$$
$$\left\{ \lightning.\, \exists t' > t.\, [\kappa]_q * \boxed{\&^{\kappa}\,(\ell,\gamma) : (t',s',v') \;\middle|\; \mathcal{I}}^{\mathcal{N}} * Q(t') \right\}_{\mathcal{E}}$$

GPS-AtBor-write

$$\frac{\uparrow\mathcal{N} \subseteq \mathcal{E} \qquad s \sqsubseteq s'}{\left\{ \begin{array}{l} [\kappa]_q * \boxed{\&^{\kappa}\,(\ell,\gamma) : (t,s,v) \;\middle|\; \mathcal{I}}^{\mathcal{N}}_{W} * \triangleright \Delta_{\pi}^{o?\mathtt{rlx}} \left(\forall t' > t. \Rightarrow\!\!\!\!\Rightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \mathcal{I}_w(\ell,\gamma,t',s',v')\right) * \\ \triangleright \langle \mathsf{obj}\rangle \left(\mathcal{I}_w(\ell,\gamma,t,s,v) \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \mathcal{I}_m(\ell,\gamma,t,s,v) * R\right) \end{array} \right\}}$$
$$\ell :=_{o} v' \text{ in } \pi$$
$$\left\{ \lightning.\, \exists t' > t.\, [\kappa]_q * \boxed{\&^{\kappa}\,(\ell,\gamma) : (t',s',v') \;\middle|\; \mathcal{I}}^{\mathcal{N}}_{W} * R \right\}_{\mathcal{E}}$$

FIGURE 17.6: Selected read and write rules for Atomic-Borrows-based GPS Protocols

- GPS-AtBor-write is a strong write rule that supports also relaxed access mode, using the single-writer assertion.

Finally, Figure 17.7 gives a CAS rule for the atomic-borrows-based single-writer protocol, which can only be used when the protocol is in CAS-only mode. GPS-AtBor-Wshr-cas-int is similar to the persistent concurrent protocol CAS rule GPS-Con-cas-int, except for the following points.

- The rule requires a fraction $[\kappa]_{q'}$ to know that that the lifetime κ is still alive, and a fraction $\boxed{\&^{\kappa}\,(\ell,\gamma) : (t,s,v) \;\middle|\; \mathcal{I}}^{\mathcal{N}}_{q}$ of the CAS-only assertion. These resources are returned (and potentially updated) in the post-condition.
- In case of a failing CAS, the extraction $\mathsf{Extract}_{\mathcal{I}}$ does not require a

$$\mathsf{Extract}_{\mathcal{I}}(\ell,\gamma,t',s',v',R,\mathcal{E},\mathcal{N}) ::= \langle \mathsf{obj}\rangle \wedge \begin{cases} \mathcal{I}_r(\ell,\gamma,t',s',v') \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \mathcal{I}_r(\ell,\gamma,t',s',v') * R(t',s',v') \\ \mathcal{I}_w(\ell,\gamma,t',s',v') \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \mathcal{I}_w(\ell,\gamma,t',s',v') * R(t',s',v') \end{cases}$$

GPS-ATBOR-WSHR-CAS-INT

$$\mathbf{rlx} \sqsubseteq o_f, o_r, o_w \qquad \uparrow\mathcal{N} \subseteq \mathcal{E} \qquad v_r \in \mathbb{Z}$$

$$\left\{ \begin{array}{l} [\kappa]_{q'} * \boxed{\&^{\kappa}\,(\ell,\gamma) : (t,s,v) \mid \mathcal{I}}^{\mathcal{N}}_{q} * \\ \triangleright \langle \mathsf{obj}\rangle \left(\forall t' \geq t, s' \sqsupseteq s, v'.\, (\mathcal{I}_r(\ell,\gamma,t',s',v') \vee \mathcal{I}_w(\ell,\gamma,t',s',v')) \mathrel{-\!\!*} \vdash v' =^{?} v_r\right) * \\ \Delta_{\pi}^{o_w ?\mathbf{rlx}}\, P * \\ \Delta_{\pi}^{o_w ?\mathbf{rlx}} \forall t' \geq t, s' \sqsupseteq s.\ \wedge \begin{cases} \triangleright (\forall v' \neq v_r.\, \mathsf{Extract}_{\mathcal{I}}(\ell,\gamma,t',s',v',R,\mathcal{E},\mathcal{N})) \\ * \begin{cases} \langle \mathsf{obj}\rangle \left(\triangleright \mathcal{I}_w(\ell,\gamma,t',s',v_r) \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} (\triangleright Q_1(t',s') * \triangleright Q_2(t',s'))\right) \\ (P * \triangleright Q_2(t',s')) \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \mathcal{W}_{\mathsf{shr}}(\ell,\gamma,t',s',v_r) * \exists s'' \sqsupseteq s'.\, \forall t'' > t'. \\ \quad \triangleright \mathcal{W}_{\mathsf{shr}}(\ell,\gamma,t'',s'',v_w) \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \\ \quad * \begin{cases} \langle \mathsf{obj}\rangle \left(\triangleright Q_1(t',s') \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \mathcal{I}_r(\ell,\gamma,t',s',v')\right) \\ \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} \triangleright \Rrightarrow_{\mathcal{E}\setminus\uparrow\mathcal{N}} (Q(t'',s'') * \mathcal{I}_w(\ell,\gamma,t'',s'',v_w)) \end{cases} \end{cases} \end{array} \right\}$$

$$\mathbf{CAS}^{o_f,o_r,o_w}(\ell, v_r, v_w) \text{ in } \pi$$

$$\left\{ b.\, [\kappa]_{q'} * \exists t', v', s' \sqsupseteq s.\ \vee \begin{cases} b = \mathbf{false} * v_r \neq v' * t \leq t' * \boxed{\&^{\kappa}\,(\ell,\gamma) : (t',s',v') \mid \mathcal{I}}^{\mathcal{N}}_{q} * \nabla_{\pi}^{o_f ?\mathbf{rlx}} R(t',s',v') * \nabla_{\pi}^{o_w ?\mathbf{rlx}} P \\ b = \mathbf{true} * v_r = v' * t < t' * \boxed{\&^{\kappa}\,(\ell,\gamma) : (t',s',v_w) \mid \mathcal{I}}^{\mathcal{N}}_{q} * \nabla_{\pi}^{o_r ?\mathbf{rlx}} Q(t',s') \end{cases} \right\}_{\mathcal{E}}$$

FIGURE 17.7: A CAS rule for Atomic-Borrows-based GPS Protocols

case for the moved interpretation $\mathcal{I}_m$, because there cannot be a single-writer write in the CAS-only mode.

- In case of a successful CAS, the user needs to provide the shared-writer permission $\mathcal{W}_{\mathsf{shr}}(\ell,\gamma,t',s',v_r)$ for the read message, and will receive the updated shared-writer permission $\mathcal{W}_{\mathsf{shr}}(\ell,\gamma,t'',s'',v_w)$ for the new write message afterwards.

Intuitively, when in CAS-only mode, every party who wants to perform a CAS with the protocol needs a fraction of the CAS-only assertion, and all parties need to share the shared-writer permission $\mathcal{W}_{\mathsf{shr}}$ that maintains the total order of updates to the location's history.

17.2 Middleware GPS Protocols in iRC11

In this section, we introduce middleware GPS protocols as a common interface that can be combined with different types of invariants to derive the various versions of surface-level GPS protocols, some of which we have seen in the previous section. We first will introduce the assertions of middleware GPS protocols, and then show how the surface-level protocol assertions are modeled using those middleware assertions.

Definition 17.7 (Middleware GPS Protocol Assertions)**.** The middleware assertions include all auxillary ghost permissions in Definition 17.5,

i.e., the read observation $\mathcal{R}(\ell,\gamma,t,s,v)$, the single-writer permission $\mathcal{W}(\ell,\gamma,t,s,v)$, the shared-writer permission $\mathcal{W}_{\text{shr}}(\ell,\gamma,t,s,v)$ and the shared-reader permission $\mathcal{R}^q_{\text{shr}}(\ell,\gamma,t,s,v)$.

Most importantly, these ghost permissions are tied together to the *core* GPS invariant construction $\mathsf{GPS}^\theta(\ell,\gamma,\mathcal{I})$ where $\theta \in \{\mathtt{con},\mathtt{sw}\}$. Intuitively, the core GPS construction $\mathsf{GPS}^\theta(\ell,\gamma,\mathcal{I})$ owns the atomic points-to of ℓ[18] and enforces that ℓ's history can only change within a GPS protocol's restrictions, who are manifested in the auxillary ghost permissions. The core construction will then be put in an invariant or an atomic borrow, so that it can be accessed concurrently by threads that own the ghost permissions.

[18] where γ is the atomic points-to's atomic period identifier—see also Definition 10.1.

Definition 17.8 (Assertion Model of GPS Persistent Concurrent Protocols)**.** Using the middleware assertions, the model of GPS persistent concurrent protocol assertions simply (1) picks the construction that supports arbitrarily concurrent accesses, *i.e.*, $\mathsf{GPS}^{\mathtt{con}}(\ell,\gamma,\mathcal{I})$; and (2) shares the construction in a persistent, *objective invariant*.[19] The subjective modality $\langle\mathsf{subj}\rangle$ [20] turns the core construction into an objective assertion such that it can be put inside the invariant.

[19] see §11.1

[20] see §8.6

$$\boxed{(\ell,\gamma):(t,s,v)\ \boxed{\mathcal{I}}}^{\mathcal{N}} ::= \mathcal{R}(\ell,\gamma,t,s,v) * \boxed{\langle\mathsf{subj}\rangle\,\mathsf{GPS}^{\mathtt{con}}(\ell,\gamma,\mathcal{I})}^{\mathcal{N}}$$

Definition 17.9 (Assertion Model of Cancelable Single-Writer GPS Protocols)**.** Similarly, the model of GPS cancelable single-writer protocol assertions (1) picks the construction $\mathsf{GPS}^{\mathtt{sw}}(\ell,\gamma,\mathcal{I})$ that supports single-writer accesses; and (2) shares the construction in a cancelable invariants;[21] and (3) pairs up the invariant assertion with the corresponding ghost permissions.

[21] see §11.2

$$\boxed{(\ell,\gamma_i,\gamma):(t,s,v)\ \boxed{\mathcal{I}}}^{\mathcal{N}} ::= \mathcal{R}(\ell,\gamma,t,s,v) * \boxed{\mathsf{GPS}^{\mathtt{sw}}(\ell,\gamma,\mathcal{I})}^{\gamma_i,\mathcal{N}}$$

$$\boxed{(\ell,\gamma_i,\gamma):(t,s,v)\ \boxed{\mathcal{I}}}^{\mathcal{N}}_{W} ::= \mathcal{W}(\ell,\gamma,t,s,v) * \boxed{\mathsf{GPS}^{\mathtt{sw}}(\ell,\gamma,\mathcal{I})}^{\gamma_i,\mathcal{N}}$$

$$\boxed{(\ell,\gamma_i,\gamma):(t,s,v)\ \boxed{\mathcal{I}}}^{\mathcal{N}}_{q} ::= \mathcal{R}^q_{\text{shr}}(\ell,\gamma,t,s,v) * \boxed{\mathsf{GPS}^{\mathtt{sw}}(\ell,\gamma,\mathcal{I})}^{\gamma_i,\mathcal{N}}$$

Definition 17.10 (Assertion Model of Atomic-Borrows-based Single-Writer GPS Protocols)**.** The model of GPS atomic-borrows-based single-writer protocol assertions is also similar to that of cancelable single-writer protocol assertions, but instead of a cancelable invariant, we put the core construction $\mathsf{GPS}^{\mathtt{sw}}(\ell,\gamma,\mathcal{I})$ in an atomic borrow.[22]

[22] see §16.2.2

$$\boxed{\&^\kappa\,(\ell,\gamma):(t,s,v)\ \boxed{\mathcal{I}}}^{\mathcal{N}} ::= \mathcal{R}(\ell,\gamma,t,s,v) * \&^{\kappa/\mathcal{N}}_{\mathbf{at}}\,\mathsf{GPS}^{\mathtt{sw}}(\ell,\gamma,\mathcal{I})$$

$$\boxed{\&^\kappa\,(\ell,\gamma):(t,s,v)\ \boxed{\mathcal{I}}}^{\mathcal{N}}_{W} ::= \mathcal{W}(\ell,\gamma,t,s,v) * \&^{\kappa/\mathcal{N}}_{\mathbf{at}}\,\mathsf{GPS}^{\mathtt{sw}}(\ell,\gamma,\mathcal{I})$$

$$\boxed{\&^\kappa\,(\ell,\gamma):(t,s,v)\ \boxed{\mathcal{I}}}^{\mathcal{N}}_{q} ::= \mathcal{R}^q_{\text{shr}}(\ell,\gamma,t,s,v) * \&^{\kappa/\mathcal{N}}_{\mathbf{at}}\,\mathsf{GPS}^{\mathtt{sw}}(\ell,\gamma,\mathcal{I})$$

From these models, it is easy to see the soundness of rules that relate the various assertions and the ghost permissions, that is, the rules GPS-SW-R, GPS-SW-W, GPS-SW-RSHR, GPS-SW-W-WSHR-RSHR, GPS-ATBOR-R, GPS-ATBOR-W, GPS-ATBOR-RSHR, and GPS-ATBOR-W-WSHR-RSHR.

GPS-MID-HIST
$$\mathsf{GPS}^{\theta}(\ell,\gamma,\mathcal{I}) \vdash \exists h.\, \ell \mapsto_{\theta} h$$

GPS-MID-CON-INIT
$$\ell \mapsto v * (\forall \gamma, t.\, \triangleright \mathcal{I}_w(\ell,\gamma,t,s,v)) \Rrightarrow_{\mathcal{E}} \exists \gamma, t.\, \triangleright \mathsf{GPS}^{\mathtt{sw}}(\ell,\gamma,\mathcal{I}) * \mathcal{R}(\ell,\gamma,t,s,v)$$

GPS-MID-SW-INIT
$$\ell \mapsto v * (\forall \gamma, t.\, \triangleright \mathcal{I}_w(\ell,\gamma,t,s,v)) \Rrightarrow_{\mathcal{E}} \exists \gamma, t.\, \triangleright \mathsf{GPS}^{\mathtt{sw}}(\ell,\gamma,\mathcal{I}) * \mathcal{W}(\ell,\gamma,t,s,v)$$

GPS-MID-SW-INIT-STRONG
$$\ell \mapsto v * (\forall \gamma, t.\, \mathcal{W}(\ell,\gamma,t,s,v) \Rrightarrow_{\mathcal{E}} \triangleright \mathcal{I}_w(\ell,\gamma,t,s,v) * Q(\gamma,t)) \Rrightarrow_{\mathcal{E}} \exists \gamma, t.\, \triangleright \mathsf{GPS}^{\mathtt{sw}}(\ell,\gamma,\mathcal{I}) * Q(\gamma,t)$$

GPS-MID-DEALLOC
$$\triangleright\, \mathsf{GPS}^{-}(\ell,\gamma,\mathcal{I}) \Rrightarrow_{\mathcal{E}} \exists t, s, v.\, \ell \mapsto v * \triangleright \mathcal{I}_w(\ell,\gamma,t,s,v)$$

GPS-MID-SW-DEALLOC
$$\triangleright\, \mathsf{GPS}^{\mathtt{sw}}(\ell,\gamma,\mathcal{I}) * \mathcal{W}(\ell,\gamma,t,s,v) \Rrightarrow_{\mathcal{E}} \ell \mapsto v * \triangleright \mathcal{I}_w(\ell,\gamma,t,s,v)$$

$$\frac{P}{Q}\ [R] \ ::= P * R \vdash Q * R$$

GPS-R-AGREE
$$\frac{\mathcal{R}(\ell,\gamma,t_1,s_1,v_1) * \mathcal{R}(\ell,\gamma,t_2,s_2,v_2)}{\Rrightarrow_{\mathcal{E}} (t_1 \le t_2 \Rightarrow s_1 \sqsubseteq s_2) \wedge (t_2 \le t_1 \Rightarrow s_2 \sqsubseteq s_1)}\ [\sqcup_V \triangleright \mathsf{GPS}^{-}(\ell,\gamma,\mathcal{I})]$$

GPS-RSHR-AGREE
$$\frac{\mathcal{R}^{q_1}_{\mathrm{shr}}(\ell,\gamma,t_1,s_1,v_1) * \mathcal{R}^{q_2}_{\mathrm{shr}}(\ell,\gamma,t_2,s_2,v_2)}{\Rrightarrow_{\mathcal{E}} (t_1 \le t_2 \Rightarrow s_1 \sqsubseteq s_2) \wedge (t_2 \le t_1 \Rightarrow s_2 \sqsubseteq s_1)}\ [\sqcup_V \triangleright \mathsf{GPS}^{\mathtt{sw}}(\ell,\gamma,\mathcal{I})]$$

GPS-R-RSHR-AGREE
$$\frac{\mathcal{R}(\ell,\gamma,t_1,s_1,v_1) * \mathcal{R}^{q_2}_{\mathrm{shr}}(\ell,\gamma,t_2,s_2,v_2)}{\Rrightarrow_{\mathcal{E}} (t_1 \le t_2 \Rightarrow s_1 \sqsubseteq s_2) \wedge (t_2 \le t_1 \Rightarrow s_2 \sqsubseteq s_1)}\ [\sqcup_V \triangleright \mathsf{GPS}^{\mathtt{sw}}(\ell,\gamma,\mathcal{I})]$$

GPS-W-LATEST
$$\frac{\mathcal{R}(\ell,\gamma,t_2,s_2,v_2)}{\Rrightarrow_{\mathcal{E}} (t_2 \le t_1 \wedge s_2 \sqsubseteq s_1)}\ [\mathcal{W}(\ell,\gamma,t_1,s_1,v_1) * \sqcup_V \triangleright \mathsf{GPS}^{\mathtt{sw}}(\ell,\gamma,\mathcal{I})]$$

GPS-MID-SW-W-REVERT
$$\frac{\mathcal{W}_{\mathrm{shr}}(\ell,\gamma,t_1,s_1,v_1) * \langle \mathsf{subj} \rangle\, \mathcal{R}^{1}_{\mathrm{shr}}(\ell,\gamma,t_2,s_2,v_2)}{\Rrightarrow_{\mathcal{E}} \mathcal{W}(\ell,\gamma,t_1,s_1,v_1)}\ [\sqcup_V \triangleright \mathsf{GPS}^{\mathtt{sw}}(\ell,\gamma,\mathcal{I})]$$

FIGURE 17.8: Selected rules for assertions of middleware GPS protocols

We now look at several structural rules for middleware GPS assertions in Figure 17.8 to informally justify the soundness of the other structural rules for surface-level assertions. *We will not attempt to justify the soundness of operational rules here, due to their complexity.* (Their soundness proofs are checked in Coq.)

- GPS-MID-HIST says that the core GPS construction for ℓ owns the atomic points-to of ℓ. Therefore the core construction is naturally shared to allow multiple threads to use the protocol. Furthermore,

the rule justifies the rule GPS-CON-ATOM-PTSTO (and similar rules for other protocol variants) that allows us to atomically peek at the atomic points-to of ℓ and subsequently learn that ℓ is still alive. Such liveness fact is needed for deterministic pointer comparison (such as in CASes).

- GPS-MID-CON-INIT intuitively performs the creation of the core GPS construction for concurrent protocols, by allocating the GPS ghost state (which will be explained in §17.3) at the ghost location γ. The rule also suggests that the core construction contains all interpretations ($\mathcal{I}_w$, $\mathcal{I}_r$, and $\mathcal{I}_m$). The rule justifies GPS-CON-INIT.

- GPS-MID-SW-INIT and GPS-MID-SW-INIT-STRONG similarly perform the creation of the core GPS construction for single-writer protocols. The rules respectively justifies GPS-SW-INIT and GPS-ATBOR-INIT.

- GPS-MID-DEALLOC and GPS-MID-SW-DEALLOC justify surface-level deallocation rules, such as GPS-SW-DEALLOC, GPS-SW-READ-ACQ-DEALLOC, and GPS-ATBOR-INIT.[23]

- GPS-R-AGREE says that two read observations are in a total order, *provided* that we get access to the core construction GPS in the frame. This is because the two read observations are related through the ghost state stored in the core construction. The core construction GPS only needs to be provided under a later modality and a view-join modality[24] $\sqcup_V$ for some view V, which makes the rule compatible with cancelable invariant access rules, *e.g.*, CINV-ACC.[25] The fancy update is needed to eliminate the later modality on the ghost state stored in GPS. GPS-R-AGREE justifies the rule GPS-CON-AGREE and its variants.

- GPS-RSHR-AGREE, GPS-R-RSHR-AGREE, and GPS-W-LATEST work similarly to GPS-R-AGREE, and respectively justify surface-level rules that relate different ghost permissions.[26]

- GPS-MID-SW-W-REVERT shows how to switch the GPS ghost state from the shared-writer and shared-reader permissions to the single-writer permission, effectively switching from CAS-only mode to single-writer mode. The rule justifies GPS-SW-W-REVERT and GPS-ATBOR-W-REVERT.

[23] Recall that GPS-ATBOR-INIT incorporates both an initialization rule and a deallocation rule—see the explanation in §17.1.3.

[24] see §8.5

[25] see §11.2

[26] Those rules are elided in this chapter.

17.3 The Model of GPS Protocols

In this section, we briefly give the model for GPS middleware protocol assertions (Definition 17.7). More specifically, we give the model of the ghost reader, writer, shared-writer, and shared-reader permissions, as well as the model of the core GPS construction GPS. We start with the resource algebra needed for the GPS ghost state.

Definition 17.11 (Resource Algebra for GPS Protocol States)**.** Assuming the protocol state type T_S we need the following RA to records how a

write event—identified by a timestamp—is tied to a protocol state.

$$\textsc{gpsr} ::= \textsc{Auth}(\textsc{Map}(\mathit{Time}, \textsc{Ag}(T_S)))$$

The fragmentary elements of GPSR will be used to model the ghost permissions, while the authoritative element will be used in the model of the core GPS construction GPS.

Definition 17.12 (Model of GPS Ghost Permissions)**.**

$$\begin{aligned}
\mathcal{R}(\ell, \gamma, t, s, v) ::= {} & \exists \gamma_i, \gamma_\ell.\, \gamma = (\gamma_i, \gamma_\ell) * \boxed{\circ\, [t \leftarrow s]}^{\gamma_i} * \\
& \exists V.\, \ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sn}} [t \leftarrow (v, V)] \\
\mathcal{W}(\ell, \gamma, t, s, v) ::= {} & \exists \gamma_i, \gamma_\ell.\, \gamma = (\gamma_i, \gamma_\ell) * \boxed{\circ\, [t \leftarrow s]}^{\gamma_i} * \\
& \exists h.\, \ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sw}} h * h(t) = (v, _) * \mathtt{max}(\mathtt{dom}(h)) = t \\
\mathcal{W}_{\text{shr}}(\ell, \gamma, t, s, v) ::= {} & \exists \gamma_i, \gamma_\ell.\, \gamma = (\gamma_i, \gamma_\ell) * \boxed{\circ\, [t \leftarrow s]}^{\gamma_i} * \\
& \exists h.\, \mathsf{atWriter}^{\gamma_\ell}(h) * \ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sy}} h * \\
& h(t) = (v, _) * \mathtt{max}(\mathtt{dom}(h)) = t \\
\mathcal{R}^q_{\text{shr}}(\ell, \gamma, t, s, v) ::= {} & \exists \gamma_i, \gamma_\ell.\, \gamma = (\gamma_i, \gamma_\ell) * \boxed{\circ\, [t \leftarrow s]}^{\gamma_i} * \\
& \exists V.\, \ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sn}} [t \leftarrow (v, V)] * \\
& \exists h, t_x \leq t, V'.\, @_{V'} \ell \sqsupseteq^{\gamma_\ell, t_x, q}_{\mathtt{cas}} h
\end{aligned}$$

The model of GPS middleware assertions for a protocol instance γ has the following pattern:

- The instance identifier γ uniquely identifies (1) the ghost location γ_i needed to store the GPS ghost state and (2) the current atomic period identifier γ_ℓ for ℓ's atomic points-to.[27]
- Each assertion owns a fragmentary element of the GPS ghost state $\boxed{\circ\, [t \leftarrow s]}^{\gamma_i}$ that tracks that the timestamp t is assigned the protocol state s.
- Each assertion owns the appropriate assertion for ℓ's atomic points-to.

More specifically:

- The reader observation $\mathcal{R}(\ell, \gamma, t, s, v)$ additionally only carries a history-seen observation[28] $\ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sn}} [t \leftarrow (v, V)]$ of the write (t, v, V).
- Meanwhile, the single-writer permission carries the single-writer ownership $\ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sw}} h$ of ℓ's atomic points-to, with $(t, v, \_)$ being the latest write in ℓ's history h.
- The shared-writer permission $\mathcal{W}_{\text{shr}}(\ell, \gamma, t, s, v)$ owns the atomic points-to's *ghost writer permission* $\mathsf{atWriter}^{\gamma_\ell}(h)$,[29] which is the exclusive permission needed to change ℓ's current history h. The shared-writer permission also requires the history-sync observation of h, and requires that t is indeed the latest write.

[27] see Definition 10.1

[28] see Definition 10.2

[29] see Definition 10.7

- The shared-reader permission is the reader observation extended with the fractional CAS ownership $\ell \sqsupseteq^{\gamma_\ell, t, q}_{\mathtt{cas}} h$ asserted at some arbitrary view V'. That the view V' can be arbitrary is because we do not need the fractional CAS ownership assertion's observation—we already have the observation $\ell \sqsupseteq^{\gamma_\ell}_{\mathtt{sn}} [t \leftarrow (v, V)]$ locally. On the other hand, we need the fractional CAS ownership to maintain that the most recent exclusive write with the timestamp t_x is frozen, and that our observation t is at least t_x. This is to ensure that the shared-reader permission prevents any single-writer writes when the protocol is in CAS-only mode. And, as we have seen in the rule GPS-AtBor-Wshr-cas-int in Figure 17.7, the shared-reader permission can only be used in conjunction with the shared-writer permission to perform CASes in the GPS single-writer protocol setup.

We elide the proofs that the model supports various rules on the relations among the ghost permissions, *e.g.*, GPS-W-R or GPS-Wshr-R.[30]

[30] see Figure 17.3

Definition 17.13 (Model of GPS Core Construction)**.**

$$
\begin{aligned}
&\mathsf{GPS}^{\theta}(\ell, \gamma, \mathcal{I}) ::= \\
&\quad \exists \gamma_i, \gamma_\ell.\, \gamma = (\gamma_i, \gamma_\ell) * \exists \mu.\, \boxed{\bullet \mu}^{\gamma_i} * \\
&\quad \exists h, t_x.\, \ell \overset{t_x}{\longmapsto}^{\gamma_\ell}_{\theta} h * \\
&\qquad \ast_{(t,v,V) \in \mathsf{block\_ends}(h)} @_V((t_x \le t)\, ?\, \mathcal{I}_w : \mathcal{I}_m)(\ell, \gamma, t, s, v) * \\
&\qquad \ast_{(t,v,V) \in (h \setminus \mathsf{block\_ends}(h))} @_V((t_x \le t)\, ?\, \mathcal{I}_{w|r} : \mathcal{I}_{w|r|m})(\ell, \gamma, t, s, v) * \\
&\quad \mathtt{dom}(\mu) = \mathtt{dom}(h) * \mathsf{state\_sorted}(\mu) * && \text{(GPS-core-wf)} \\
&\quad \forall t \in \mathsf{disconnected\_from}(\mu, t_x).\, \mathsf{final}(\mu(t)) * && \text{(GPS-core-fin)} \\
&\quad \begin{cases} \theta = \mathtt{sw} & \forall t \in \mathtt{dom}(h).\, t_x \le t \Rightarrow \\ & \qquad t \notin \mathsf{disconnected\_from}(\mu, t_x) \\ \theta = \mathtt{con} & \forall t \in \mathtt{dom}(h).\, t_x \le t \end{cases} && \text{(GPS-core-sw-con)}
\end{aligned}
$$

where

$$
\begin{aligned}
\mathsf{block\_ends}(h) &::= \{(t, v, V) \in h \mid t + 1 \notin \mathtt{dom}(h)\} \\
\mathcal{I}_{w|r}(\ell, \gamma, t, s, v) &::= \mathcal{I}_w(\ell, \gamma, t, s, v) \vee \mathcal{I}_r(\ell, \gamma, t, s, v) \\
\mathcal{I}_{w|r|m}(\ell, \gamma, t, s, v) &::= \mathcal{I}_{w|r}(\ell, \gamma, t, s, v) \vee \mathcal{I}_m(\ell, \gamma, t, s, v) \\
\mathsf{state\_sorted}(\mu) &::= \forall t_1, t_2.\, t_1 \le t_2 \Rightarrow \mu(t_1) \sqsubseteq \mu(t_2) \\
\mathsf{final}(s) &::= \forall s' \in T_S.\, s' \sqsubseteq s \\
\mathsf{disconnected\_from}(\mu, t_x) &::= \{t \in \mathtt{dom}(\mu) \mid \exists t_d \notin \mathtt{dom}(\mu).\, t_x \le t_d \le t\}
\end{aligned}
$$

The core $\mathsf{GPS}^{\theta}(\ell, \gamma, \mathcal{I})$ construction is rather simple—it is composed of several components that are tied together to establish a protocol instance γ on the location ℓ with the interpretation $\mathcal{I}$.

- the authoritative GPS ghost state $\bullet \mu$ at the ghost location γ_i;

- the atomic points-to $\ell \overset{t_x}{\longmapsto}{}^{\gamma_\ell}_{\theta}\ h$;
- the interpretation resources ($\mathcal{I}_w$, $\mathcal{I}_r$, and $\mathcal{I}_m$) for the write messages in ℓ's history h;
- the various properties (GPS-CORE-WF, GPS-CORE-FIN, and GPS-CORE-SW-CON) that ties μ and h together.

More specifically,

[31] see Definition 6.11

[32] see Definition 17.13

- Thanks to the authoritative algebra,[31] the map μ from timestamps to protocol states is the upper bound for all ghost permissions.[32]
- The atomic points-to has the same mode θ as the protocol's mode, and its most recent exclusive write t_x dictates which interpretation resources are variable for a write message in h.
- The write messages in h are split into *block-ends* and *non block-ends*. Intuitively, the messages in h are grouped into *contiguous blocks*: a write can always initiate a new block, while a CAS always extends an existing block. As a CAS can take the write interpretation $\mathcal{I}_w$ of the message it reads from in order to establish $\mathcal{I}_w$ for its write message, $\mathcal{I}_w$ is guaranteed for block ends, *unless* it has been taken away ("moved") by an exclusive single-writer write.[33]. In other words, for block-ends that are later than the most recent exclusive write t_x we have $\mathcal{I}_w$, and for block-ends that are earlier we only have the moved interpretation $\mathcal{I}_m$. Meanwhile, for non block-ends that are later than t_x we can have either $\mathcal{I}_w$ or $\mathcal{I}_r$ (in case it has been CAS-ed on), and for non block-ends that are earlier than t_x we can have any case of $\mathcal{I}_w$, $\mathcal{I}_r$, or $\mathcal{I}_m$.

[33] see, *e.g.*, GPS-SW-WRITE-REL in Figure 17.4

- GPS-CORE-WF requires that μ and h must agree on the current set of timestamps, and that μ enforces monotonicity on states.
- GPS-CORE-FIN requires that any write *disconnected from*—*i.e.*, later than and not from the same block with—t_x (must be *final*. This property is needed for state monotonicity, as can be seen in the concurrent write rule GPS-CON-WRITE, but it is not very restrictive, so as to support single-write rules, *e.g.*, GPS-SW-WRITE-REL.
- GPS-CORE-SW-CON requires that, in a single-writer protocol, any writes later than t_x are not disconnected from t_x—effectively enforces that ony single-writer writes or CASes can be performed. Meanwhile, a concurrent protocol forbids single-writer writes: any writes are later than the most recent exclusive write t_x.

CHAPTER SUMMARY. In this chapter, we presented the interfaces and the models of various GPS protocols. A GPS protocol is built atop the atomic points-to, and provides some abstraction of write messages into protocol states that can only grow monotonically. The construction of a GPS protocol model is done in multiple layers, allowing us to attach it with different invariant types, including atomic borrows. In the next chapters, we will see how GPS protocols can be used in $\mathsf{RB}_{\mathtt{rlx}}$ to verify concurrent Rust libraries that use `unsafe` blocks.

18

Verification of RwLock

In this chapter, we demonstrate a concrete verification of the Rust's reader-writer lock library RwLock as part of RB$_{\texttt{r1x}}$'s Task 1—proving the safety of Rust libraries under relaxed memory concurrency. As with the original RustBelt work,[1] we perform the verification with a hand-translated λ_{Rust} version of RwLock. In the framework of RustBelt, the obligations of verifying that a type τ satisfies the Rust type system pin down to (1) giving a separation-logic *semantic interpretation* of τ that satisfies certain requirements, and (2) proving that the type's operations *maintain* the semantic interpretation. More specifically, the semantic interpretation $[\![\tau]\!]$ for τ is a triple $([\![\tau]\!].\text{size}, [\![\tau]\!].\text{own}, [\![\tau]\!].\text{shr})$ that respectively represents (1) the memory size of τ, (2) the separation-logic unique ownership of an *owned* object or a *mutable reference* to an object of type τ, and (3) the separation-logic shared ownership of an *immutable reference* to an object of type τ. We will give more details on the requirements of the predicates when we get to define them for RwLock. However, we would like to note that for concurrently libraries (which are our focus here), the most relevant predicate is the shared predicate $[\![\tau]\!].\text{shr}$ which arbitrates concurrent accesses to the underlying resources of the type, and which encodes the *interior mutability* of the library through immutable references. For RwLock, we will use atomic-borrows-based single-writer GPS protocols[2] to define $[\![\text{RwLock}]\!].\text{shr}$.

[1] Jung et al., "RustBelt: Securing the Foundations of the Rust Programming Language" [Jun+18a].

[2] see §17.1.3

In §18.1, we will present the relaxed memory λ_{Rust} version of RwLock<T>, which uses the same relaxed memory operations as that of the Rust's library. We then give the library the semantic model $[\![\text{RwLock}]\!]$ in §18.2 and sketch how the operations maintain the model in §18.3.

18.1 RMC Implementation of a Reader-Writer Lock

The RMC λ_{Rust} implementation of RwLock<T> and its related types—*lock guards*—RwLockReadGuard<T> and RwLockWriteGuard<T> are given in Figure 18.1. Intuitively, the reader-writer lock RwLock<T> protects an underlying object of type T. When acquiring a reader lock or a writer lock, a corresponding lock guard of type RwLockReadGuard<T> or RwLockWriteGuard<T> respectively is returned. The lock guard follows the "Resource Acquisition is initialization" (RAII) idiom, where the lifetime of the lock guard is the duration that a client holds on to the lock: when the lock is acquired, the lock guard object is created, and when the

```
RwLock<T>::          new :  fn(T) -> RwLock<T>
RwLock<T>::   into_inner :  fn(self) -> T
RwLock<T>::     get_mut :  fn(&mut self) -> &mut T
RwLock<T>::    try_read :  fn(&self) -> Result<RwLockReadGuard<T>>
RwLock<T>::   try_write :  fn(&self) -> Result<RwLockWriteGuard<T>>
RwLockReadGuard<T>::       deref :  fn(&self) -> &T
RwLockReadGuard<T>::        drop :  fn(&mut self)
RwLockWriteGuard<T>::      deref :  fn(&self) -> &T
RwLockWriteGuard<T>::  deref_mut :  fn(&mut self) -> &mut T
RwLockWriteGuard<T>::       drop :  fn(&mut self)
```

TABLE 18.1: A summary of Rust types for `RwLock<T>` and its lock guards

lock guard object is destroyed, the lock is released.

Table 18.1 gives a summary of the Rust types for the functions in Figure 18.1.

- **`new_lock`**(τ) is the λ_{Rust} version of `fn new(x: T) -> RwLock<T>` which consumes an object of type `T` and returns a reader-writer lock for `T`. The type τ is the λ_{Rust} version of `T`.

- **`into_inner`**(τ) consumes the reader-writer lock and returns the underlying object of type `T`. **`self`** is the Rust idiom to mention the current object, which, in this case, is of type `RwLock<T>`.

- **`get_mut`** borrows a mutable reference to the underlying object from a mutable reference to the lock. There is no need to acquire the lock, because the mutable reference to the lock should guarantee that no one holds the lock.

- **`try_read`** tries to acquire a reader lock from an *immutable* reference to the lock. The `Result` type has two cases: a `None` means the function fails to acquire the reader lock, and a `Some` means the reader lock has been acquired through a reader lock guard. In the Rust version, this function's body uses **`unsafe`**.

- **`try_write`** tries to acquire a writer lock from an immutable reference to the lock. If the writer has been acquired then a writer lock guard is returned. In the Rust version, this function's body uses **`unsafe`**.

- **`rdguard_deref`** borrows an immutable reference to the underlying object `T` of the lock from a reader lock guard. (**`self`** is of type `RwLockReadGuard<T>` here.) In the Rust version, this function's body uses **`unsafe`**.

- **`rdguard_drop`** is the destructor of `RwLockReadGuard<T>` that also releases a reader lock. In the Rust version, this function's body uses **`unsafe`**.

- **wrguard_deref** and **wrguard_deref_mut** respectively borrow an immutable or mutable reference to the underlying object `T` of the lock from a writer lock guard. (**self** is of type `RwLockWriteGuard<T>` here.) In the Rust version, this function's body uses **unsafe**.
- **wrguard_drop** is the destructor of `RwLockWriteGuard<T>` that also releases a writer lock. In the Rust version, this function's body uses **unsafe**.

We now look more closely at the λ_{Rust} implementations in Figure 18.1.

- **new_lock**(τ) shows that the memory layout of a `RwLock<T>` is a counter at offset 0 and the memory of τ itself at offset 1. The counter represents the lock state: if it is -1, the writer lock is being held, while if it is positive, the number is the number of reader locks being held. The lock is unlocked when the counter is 0.

  Since λ_{Rust} is call-by-value, it models consumption of the object of type τ (addressed by the value of x) as mem-copying (non-atomically) the contents of τ, and then deleting the original input object.[3]
- Similarly, **into_inner**(τ) non-atomically mem-copies the underlying object τ from offset 1 of the lock (addressed by the value of x) into a fresh memory region (of size τ.size), and then deletes the whole lock object.
- **get_mut** reads the address y to the lock stored in the argument x, and updates (non-atomically writes to) x with the address to the underlying τ (offset by 1 from the lock's address).
- **try_read** returns r as the address to a memory region of size 2 for the type `Result<RwLockReadGuard<T>>`. The first memory cell of the memory region stores the variant of the `Result` type (None or Some), and the second memory cell in the Some case stores the value of the type `RwLockReadGuard<T>`, which is simply the address y to the lock (stored in the argument x). A reader lock can only be acquired using an acquire CAS if either the lock is unlocked (the counter is 0) or there are reader locks (the counter is positive). The function only retries in case the CAS fails due to contention with other threads trying to acquire reader locks.[4]
- **try_write** does not retry at all. A writer lock can only be acquired when the lock is unlocked, also by using an acquire CAS.
- **rdguard_deref** simply returns r that stores the address to the underlying object of type τ.
- **rdguard_drop** keeps trying until successful decrement the counter.
- **wrguard_deref** and **wrguard_deref_mut** are exactly the same as **rdguard_deref**—they only differ by types.
- **wrguard_drop** simply resets the counter to 0 with a release write and thus releases the writer lock.

[3] The implementations of **new**, **delete**, and mem-copy have been provided in Figure 4.2.

[4] The implementation of **letcont** is also provided in provided in Figure 4.2.

```
new_lock(τ) ::=
  λ[x]. 1: let r := new[τ.size + 1] in
        2: r + 0 := 0;
        3: r + 1 :=_{τ.size} *x;
        3: delete[τ.size, x];
        4: r

into_inner(τ) ::=
  λ[x]. 1: let r := new[τ.size] in
        2: r :=_{τ.size} *(x + 1);
        3: delete[τ.size + 1, x];
        4: r

get_mut ::=
  λ[x]. 1: let y := *x in
        2: x := y + 1;
        3: x

rdguard_deref ::=
  λ[x]. 1: let y := *x in
        2: let r := new[1] in
        3: r := *(y + 1);
        4: delete[1, x];
        5: r

rdguard_drop ::=
  λ[x]. 1: let y := *x in
        2: letcont loop([]) :=
        3:   let n := *^rlx y in
        4:   if CAS^rel(y, n, n − 1)
        5:    then
        6:      delete[1, x]
        7:    else loop([])
        8:  in loop([])

try_read ::=
  λ[x]. 1: let r := new[2] in
        2: let y := *x in
        3: letcont k([]) := delete[1, x]; r in
        4: letcont loop([]) :=
        5:   let n := *^rlx y in
        6:   if n ≤ −1
        7:    then r :=^{inj 0} (); k([]) // returns None
        8:    else
        9:      if CAS^acq(y, n, n + 1)
       10:       then r :=^{inj 1} y; k([]) // returns Some
       11:       else loop([])
       12:  in loop([])

try_write ::=
  λ[s]. 1: letcont k([r]) := delete[1, x]; r in
        2: let r := new[2] in
        3: let y := *x in
        4: if CAS^acq(y, 0, −1)
        5:  then r :=^{inj 1} y; k([r]) // returns Some
        6:  else r :=^{inj 0} (); k([r]) // returns None

wrguard_deref ::=
  λ[x]. 1: let y := *x in
        2: let r := new[1] in
        3: r := *(y + 1);
        4: delete[1, x];
        5: r

wrguard_deref_mut ::=
  λ[x]. 1: let y := *x in
        2: let r := new[1] in
        3: r := *(y + 1);
        4: delete[1, x];
        5: r

wrguard_drop ::=
  λ[x]. 1: let y := *x in
        2: y :=_rel 0;
        3: delete[1, x]
```

FIGURE 18.1: A λ_{Rust} version of Rust's RMC RwLock<T>

18.2 The Semantic Model of the Reader-Writer Lock Type

The general formula of semantically type checking an expression e to have the type τ is to show that, under an assignment of semantically well-typed values for all free variables in e, the substituted expression of e executes without errors, and if the execution terminates, it terminates with a value that satisfies the semantic interpretation of τ. More formally, a semantic typing judgment $[\![\Gamma \vdash e : \tau]\!]$ is typically defined as:

$$[\![\Gamma \vdash e : \tau]\!] ::= \forall \gamma \in [\![\Gamma]\!].\, \mathsf{wp}\, \gamma e \left\{ v.\, [\![\tau]\!]_\gamma(v) \right\}$$

where γ is the assignment satisfying the context Γ for free variables in e, and γe is the substitution of e with γ. The semantic typing for the function type may also read as:

$$[\![\Gamma \vdash \lambda x.\, e : \tau_1 \to \tau_2]\!] ::= \\ \forall \gamma \in [\![\Gamma]\!], v.\, [\![\tau_1]\!]_\gamma(v) \mathrel{-\!\!*} \mathsf{wp}\, \gamma((\lambda x.\, e)v) \left\{ v'.\, [\![\tau_2]\!]_\gamma(v') \right\}$$

That is, for $\lambda x.\, e$ to be semantically well-typed if when provided with the interpretation of the arguments $[\![\tau_1]\!]_\gamma(v)$, the resulting function application is also semantically well-typed.

Therefore, to show that the reader-writer lock library is semantically well-typed, we first need to provide the semantic interpretations for `RwLock<T>`, `RwLockReadGuard<T>`, and `RwLockWriteGuard<T>`. Then we need to show that the library's functions are also semantically well-typed, which, as explained above, translates to verifications of function bodies in our separation logic.

The semantic typing judgment of RustBelt is more complicated than the general intuition above: the type system needs to account for lifetimes and continuations. However, in the following, we focus on the type interpretations, and only sketch the verifications of the function bodies, and avoid going into the details of the semantic typing judgment.

Definition 18.1 (RA for RwLock)**.** We use the unit protocol $S_{()}$ for `RwLock`. That is, the protocol is trivial, so we always ignore the state (which is only unit). Instead, we will use the richer ghost state of the algebra RWLOCKR to model the lock states.

$$\text{RWLOCKR} ::= \text{AUTH}(\text{OPTION}(\text{SUM}(\text{EX}(1), \text{AG}(\mathit{Lft}) \times \text{FRAC} \times \text{POS})))$$

This algebra is already used in the original RustBelt work. The lock's ghost state is an option of a sum of two types of sub states. The None case models the unlocked state. The Some case with the left-hand side of the sum models the writer lock state, while a Some case with the right-hand side of the sum models a reader lock state. The writer lock state needs to be exclusive (EX), while a reader lock state needs to track the lifetimes (using agreement AG) of all reader locks, the fractions (FRAC) of the lock being held by all reader locks, and the number (POS—positive numbers) of reader locks.

$$\mathsf{Writing} ::= \circ\, \mathsf{Some}(\mathsf{left}())$$
$$\mathsf{Reading}(q, \nu) ::= \circ\, \mathsf{Some}(\mathsf{right}(\mathsf{ag}(\nu), q, 1))$$

The writer state Writing will be owned by the one holding the writer lock. Every piece of reader state Reading(q, ν) will be held by one (1) reader where q is the fraction of the lock held by the reader, and q is the lifetime for the reader lock guard.

Definition 18.2 (GPS Protocol for RwLock)**.** We will not rely on the timestamps exposed by our version of GPS protocols. As we are working with a concurrent Rust type, it is natural to use an atomic-borrows-based variant of GPS protocols. We use the single-writer variant introduced in §17.1.3—the writer lock relies on single-writer writes, while reader locks rely on CAS-only writes.

$$
\begin{aligned}
\mathcal{I} &::= (\mathcal{I}_w, \mathcal{I}_r, \mathcal{I}_m) \\
\mathcal{I}_r &::= \lambda \ell, \gamma, \_, \_, v.\, \mathsf{True} \\
\mathcal{I}_m &::= \lambda \ell, \gamma, \_, \_, v.\, v = -1 \\
\mathcal{I}_w(\tau, \alpha, \gamma_s, t) &::= \lambda \ell, \gamma, \_, \_, v.\, \exists a.\, \boxed{\bullet a}^{\gamma_s} * \\
&\qquad \begin{cases}
a = \mathsf{None} * v = 0 \\
\quad * \ \&^{\alpha}_{\mathbf{full}} (\ell + 1 \mapsto [\![\tau]\!].\mathrm{own}(t)) \\
\quad * \ \mathcal{W}_{\mathrm{shr}}(\ell, \gamma, \_, \_, 0) * \mathcal{R}^{1}_{\mathrm{shr}}(\ell, \gamma, \_, \_, 0) \\
a = \mathsf{Some}(\mathsf{right}(\nu, q, n)) * v = n \\
\quad * \ \mathcal{W}_{\mathrm{shr}}(\ell, \gamma, \_, \_, n) \\
\quad * \ [\![\tau]\!].\mathrm{shr}(\alpha \sqcap \nu, t, \ell + 1) \\
\quad * \ \square\, \mathsf{Kill}(\nu) \\
\quad * \ \mathsf{Inh}(\nu, \&^{\alpha}_{\mathbf{full}} (\ell + 1 \mapsto [\![\tau]\!].\mathrm{own}(t))) \\
\quad * \ \exists q'.\, q + q' = 1 * [\nu]_{q'} * \mathcal{R}^{q'}_{\mathrm{shr}}(\ell, \gamma, \_, \_, n) \\
a = \mathsf{Some}(\mathsf{left}()) * v = -1
\end{cases}
\end{aligned}
$$

where

$$\ell \mapsto \Phi ::= \exists \overline{v}.\, \ell \mapsto \overline{v} * \Phi(\overline{v})$$

[5] See Definition 17.3

Recall that we need to define the read, write, and move interpretations $(\mathcal{I}_r, \mathcal{I}_w, \mathcal{I}_m)$.[5] Since we do not have any interesting read operation for the lock itself (acquiring and releasing locks are writes), $\mathcal{I}_r$ is trivial. The move interpretation $\mathcal{I}_m$ encodes the obligation for single-writer writes, and as we only use single-writer writes to release the writer lock—switching the lock from -1 to 0—we only require that only writer lock writes (-1) can be overwritten by single-writer writes that release the writer lock.

The write interpretation $\mathcal{I}_w$ is the most important. It is parameterized by the underlying type τ, the lifetime upper-bound α of the lock guards (the upper-bound of *e.g.*, `'a` in `RwLockReadGuard<'a,T>`), the ghost location γ_s for the lock ghost state (whose type is the RWLOCKR algebra), and the owner's thread id t. $\mathcal{I}_w$ always holds on to the current authoritative lock state $\bullet a$ (while the clients of the lock hold the fragmentary parts, in Reading or Writing). $\mathcal{I}_w$ enforces that the *physical* lock state v stored at ℓ must agree with the ghost state a: if a is None, the lock is unlocked and $v = 0$; if the lock is write-locked, a is in the left-hand side of the sum and

$v = -1$; if the lock is read-locked by n readers, a is in the right-hand side of the sum and $v = n$.

Furthermore, when the lock is unlocked, the invariant holds (1) the resources for the underlying type τ, in form of a full borrow[6] with the *own* part $[\![\tau]\!]$.own of the semantic interpretation of τ (see more below); (2) the shared-writer and shared-reader permissions of the protocol,[7] both with value 0. These resources will be given to those acquiring a lock, so that they can access the underlying type τ as well as permissions to release the lock.

[6] see §15.2

[7] see Definition 17.5

The notation $\ell \mapsto \Phi$ is short for $\exists \overline{v}.\, \ell \mapsto \overline{v} * \Phi(\overline{v})$, which represents the resources that Φ holds for the list of values v that are stored at ℓ. Effectively, when the lock is unlocked, the invariant holds the points-to of τ with values $\overline{v}$ as well as $[\![\tau]\!]$.own$(t, \overline{v})$—the resources of τ that hold at the owner's thread t and $\overline{v}$.

When the lock is write-locked, all of the resources in (1) and (2) will be held by the one holding the lock, so the invariant has nothing left. The one holding the writer lock will have access to $[\![\tau]\!]$.own, which is the full resources of the underlying type τ.

When the lock is read-locked, the invariant holds (1) the shared-writer permission $\mathcal{W}_{\text{shr}}(\ell, \gamma, \_, \_, n)$ needed to perform CASes (to acquire reader locks), with n being the current number of reader locks; (2) the *share* part $[\![\tau]\!]$.shr of τ's semantic interpretation, which will be given to any one acquiring a reader lock, so that they can access τ's contents in *shared* mode; (3) the ability to kill the shared lifetime ν ($\mathsf{Kill}(\nu)$) and to inherit the own resources of τ, which will be used to switch the lock from sharing (read-lock) mode to unlocked mode; (4) the remaining available resources that will be given to new reader locks, which include a fraction q' of the lifetime token for ν ($[\nu]_{q'}$) and the same fraction q' for the shared-reader permission ($\mathcal{R}^{q'}_{\text{shr}}(\ell, \gamma, \_, \_, n)$) needed to release a reader lock (performing a CAS). q' together with q—the fraction of those resources that have been given out to existing reader lock holders—must sum up to 1.

The semantic type interpretations for `RwLock` and its lock guards are provided by Definition 18.3. In the RustBelt framework, an interpretation of a type τ is in the form Type$\{\text{size} := \ldots; \text{own} := \ldots; \text{shr} := \ldots\}$. $[\![\tau]\!]$.size is the number of values that τ has, *i.e.*, the length of a list of values $\overline{v}$ of τ. $[\![\tau]\!]$.own is a predicate of type $\mathit{Thread} \times \mathit{Val}^* \to \mathsf{vProp}$, where $[\![\tau]\!]$.own$(t, \overline{v})$ should encode what it means for the thread t to *uniquely own* an object of type τ (*i.e.*, owning a `T`) or a *mutable* reference to an object of type τ (*i.e.*, own a `&mut T`), and the object has the values $\overline{v}$. $[\![\tau]\!]$.shr is a predicate of type $\mathit{Lft} \times \mathit{Thread} \times \mathit{Loc} \to \mathsf{vProp}$, where $[\![\tau]\!]$.shr(κ, t, ℓ) should encode what it means for the thread t to own, at the location ℓ, a *shared* reference, bounded by the lifetime κ, to an object of type τ (*i.e.*, own a `&T`). Indeed, the predicate is used for the own predicate of a shared reference:

$$[\![\&^{\kappa}_{\mathbf{shr}}\, \tau]\!].\text{size} ::= 1$$
$$[\![\&^{\kappa}_{\mathbf{shr}}\, \tau]\!].\text{own}(t, \overline{v}) ::= \exists \ell.\, \overline{v} = [\ell] * [\![\tau]\!].\text{shr}(\kappa, t, \ell)$$

$[\![\tau]\!].\mathrm{shr}(\kappa, t, \ell)$ needs to satisfy several properties: it needs to be persistent and it needs to be closed with respect to shorter lifetimes, because shared references should be easily copied (shared) for smaller lifetimes. But most importantly, the predicate needs to satisfy the property that shared references can be created from mutable references:

$$\left(\&^{\kappa}_{\mathbf{full}}\left(\exists \overline{w}.\, \ell \mapsto \overline{w} * [\![\tau]\!].\mathrm{own}(t, \overline{w})\right) * [\kappa]_q\right) \Rrightarrow \left([\![\tau]\!].\mathrm{shr}(\kappa, t, \ell) * [\kappa]_q\right) \tag{TY-SHARE}$$

Definition 18.3 (Type Interpretations)**.**

$[\![\mathbf{rwlock}(\tau)]\!] ::=$

Type {size $:= 1 + [\![\tau]\!].\mathrm{size}$;

own $:= \lambda t, \overline{v}.\, \exists n \geq -1, \overline{w}.\, \overline{v} = [n] +\!\!+ \overline{w} * [\![\tau]\!].\mathrm{own}(t, \overline{w})$;

shr $:= \lambda \kappa, t, \ell.\, \exists \alpha \sqsupseteq \kappa, \gamma, \gamma_s.\, \boxed{\&^{\alpha}\,(\ell, \gamma) : \_ \mid \mathcal{I}(\tau, \alpha, \gamma_s, t)}^{\mathcal{N}}$ }

$[\![\mathbf{rwlockreadguard}(\alpha, \tau)]\!] ::=$

Type {size $:= 1$;

own $:= \lambda t, \overline{v}.\, \exists \ell.\, \overline{v} = [\ell] * \exists \nu, q, \beta \sqsupseteq \alpha, t_o, \gamma, \gamma_s.$

$[\![\tau]\!].\mathrm{shr}(\alpha \sqcap \nu, t, \ell + 1) * [\nu]_q * \square\, \mathsf{Kill}(\nu) *$

$\boxed{\mathsf{Reading}(q, \nu)}^{\gamma_s} * \boxed{\&^{\beta}\,(\ell, \gamma) : \_ \mid \mathcal{I}(\tau, \beta, \gamma_s, t)}^{\mathcal{N}}_{q}$;

shr $:= \lambda \kappa, t, \ell.\, \exists \ell'.\, \&^{\kappa}_{\mathbf{frac}}(\lambda q.\, \ell \overset{q}{\mapsto} [\ell']) * \triangleright\, [\![\tau]\!].\mathrm{shr}(\alpha \sqcap \kappa, t, \ell' + 1)$}

$[\![\mathbf{rwlockwriteguard}(\alpha, \tau)]\!] ::=$

Type {size $:= 1$;

own $:= \lambda t, \overline{v}.\, \exists \ell.\, \overline{v} = [\ell] * \exists \beta \sqsupseteq \alpha, \gamma, \gamma_s.$

$\&^{\beta}_{\mathbf{full}}\,(\ell + 1 \mapsto [\![\tau]\!].\mathrm{own}(t)) *$

$\boxed{\mathsf{Writing}}^{\gamma_s} * \boxed{\&^{\beta}\,(\ell, \gamma) : (\_, \_, -1) \mid \mathcal{I}(\tau, \beta, \gamma_s, t)}^{\mathcal{N}}_{W}$;

shr $:= \lambda \kappa, t, \ell.\, \exists \ell'.\, \&^{\kappa}_{\mathbf{frac}}(\lambda q.\, \ell \overset{q}{\mapsto} [\ell']) * \triangleright\, [\![\tau]\!].\mathrm{shr}(\alpha \sqcap \kappa, t, \ell' + 1)$}

The interpretations of reader-writer lock types are as follows.

- For $\mathbf{rwlock}(\tau)$, its size is the size of the underlying type τ plus one for the counter. Its own interpretation says that the current list of values $\overline{v}$ (of length $[\![\tau]\!].\mathrm{size}$) starts with the first value being the number of locks n, which must always be at least -1. (Recall that -1 stands for the writer-locked state, 0 for the unlocked state, and $n > 0$ for the reader-locked state with n readers.) The rest of the list $\overline{w}$ satisfies the underlying type τ's own interpretation ($[\![\tau]\!].\mathrm{own}(t, \overline{w})$).

  $\mathbf{rwlock}(\tau)$'s shared interpretation for a shared reference to the lock bound by the lifetime κ simply says that the lock state must be governed by an atomic-borrowed-based GPS protocol—with the protocol given by Definition 18.2—bound by a lifetime α that is at least κ. The lifetime inclusion relation $\alpha \sqsupseteq \kappa$ is typically used to make sure that the interpretation is closed under smaller lifetimes, so that it can allow for reborrowing. In case of $[\![\mathbf{rwlock}(\tau)]\!].\mathrm{shr}$,

this means that the lock is already governed by a protocol bound by α, and is then available for any smaller lifetime κ. Note that when proving TY-SHARE for **rwlock**(τ), we perform the allocation of the GPS protocol. Reversely, when the sharing ends with the end of the lifetime κ, we perform the *de*allocation of the GPS protocol.

- For **rwlockreadguard**(α, τ), the size is simply one because it only records the base location ℓ of the lock. The own interpretation simply owns the resources that are acquired by a reader lock, as already mentioned in Definition 18.2: a reader lock owns (1) the shared interpretation $[\![\tau]\!]$.shr at the intersected lifetime $\alpha \sqcap \nu$, so that one can access τ's contents in shared mode; (2) a fraction q of the lifetime token for ν ($[\nu]_q$) to maintain that ν is alive; and (3) the ability to kill the shared lifetime ν ($\mathsf{Kill}(\nu)$) and to inherit the own resources of τ, which will be used to switch the lock from sharing (read-lock) mode to unlocked mode if this is the only existing reader lock; (4) the same fraction q for the reader state $\boxed{\mathsf{Reading}(q,\nu)}^{\gamma_s}$ as well as the shared-reader permission of the GPS protocol bound by the lifetime β that is at least the lifetime α that has computed statically by the type system for the guard.

  The shared interpretation of the lock guard itself is just sharing the ownership of the storage to the base location of the lock (now quantified as ℓ') and the shared interpretation of the underlying type τ bound by both the static lifetime α of the lock guard and the lifetime κ of the shared reference (to the lock guard).

- For **rwlockwriteguard**(α, τ), the size and the shared interpretation are similar to those of reader guards.

  The own interpretation is rather simple compared to that of the reader lock guard. In addition to owning the writer state $\boxed{\mathsf{Writing}}^{\gamma_s}$ as well as the writer permission of the GPS protocol—bound by β, and with the writer-lock state value -1, the interpretation also holds the own interpretation of the underlying type τ, but under a full borrow bound by β. Intuitively, the resources $\ell{+}1 \mapsto [\![\tau]\!].\mathsf{own}(t)$ is under a borrow at β because it is indeed borrowed from the own interpretation of **rwlock**(τ) for the writer lock guard when the lock is acquired (when the writer lock guard is created).

18.3 Proof Sketches of the Library's Operations

In this section we sketch the proofs that the implementation of the reader-writer lock's functions given in Figure 18.1, which uses **unsafe** code, is semantically well-typed. That is, we show that the implementation satisfies the semantic interpretations for their corresponding Rust types given in Table 18.1, using the core interpretations of Definition 18.3. More specifically, the proof obligation for a function is typically that, given the interpretations of the function's arguments, we need to produce the interpretation of the function's return value, while maintaining all

invariants. Below, we only sketch the proofs for the most interesting functions.

Proof sketch of `new_lock`(τ). The type is `fn(T) -> RwLock<T>`, so we get to assume $[\![\tau]\!].\mathrm{own}(t, \overline{w})$ where t is the calling thread and w are the values of the argument of type τ, and we need to produce $[\![\mathbf{rwlock}(\tau)]\!].\mathrm{own}(t, \overline{v})$. This is easy because $[\![\mathbf{rwlock}(\tau)]\!].\mathrm{own}(t, \overline{v})$ contains $[\![\tau]\!].\mathrm{own}(t, \overline{w})$, and the implementation only needs to maintain that the lock counter n is at least -1. In fact, `new_lock`(τ) initializes the lock counter to 0 (the unlocked state). Recall that the allocation of the GPS protocol is done only when one gets a shared reference to the lock, and we have performed that obligation in TY-SHARE. □

Proof sketch of `try_read`. `try_read` takes in a `&self`, a shared reference of $\mathbf{rwlock}(\tau)$, and returns a `Result<RwLockReadGuard<T>>`. We only consider the successful case, where the `Result` is a `Some`. In that case, we get to assume $[\![\mathbf{rwlock}(\tau)]\!].\mathrm{shr}(\kappa, t, \ell)$ where κ is the lifetime of the shared reference and ℓ is the base location of the lock, and we need to produce $[\![\mathbf{rwlockreadguard}(\alpha, \tau)]\!].\mathrm{own}(t, [\ell])$, knowing that $\alpha \sqsubseteq \kappa$. We therefore can pick κ to instantiate β in $[\![\mathbf{rwlockreadguard}(\alpha, \tau)]\!].\mathrm{own}(t, [\ell])$.

The success case means that the CAS in line 9 of `try_read` (see Figure 18.1) has successfully increased the lock counter by 1. In the proof, we rely on the GPS protocol assertion $\boxed{\&^{\alpha}(\ell, \gamma) : \_ \mid \mathcal{I}(\tau, \alpha, \gamma_s, t)}^{\wedge}$ from $[\![\mathbf{rwlock}(\tau)]\!].\mathrm{shr}(\kappa, t, \ell)$ to verify the CAS (using *e.g.*, GPS-ATBOR-WSHR-CAS-INT in Figure 17.7). While doing so, we interact with the write interpretation $\mathcal{I}_w(\tau, \alpha, \gamma_s, t)$ given in Definition 18.2, knowing that the state a is $\mathsf{Some}(\mathsf{right}(\nu, q, n))$. We can then acquire all of the resources needed for $[\![\mathbf{rwlockreadguard}(\alpha, \tau)]\!].\mathrm{shr}(\kappa, t, \ell)$, by carving out some fraction q'' from the remaining fraction q' in the write interpretation, and then updating the state a to $\mathsf{Some}(\mathsf{right}(\nu, q + q'', n + 1))$. □

Proof sketch of `try_write`. The proof is similar to that of `try_read`: when performing the CAS from 0 to -1 in 4 of `try_write`, we also interact with the write interpretation $\mathcal{I}_w(\tau, \alpha, \gamma_s, t)$, but this time we know that $a = \mathsf{None}$. We then update the state a to $\mathsf{Some}(\mathsf{left}())$ and acquire $\boxed{\mathsf{Writing}}^{\gamma_s}$. Together with the remaining resources from $\mathcal{I}_w(\tau, \alpha, \gamma_s, t)$, we can establish $[\![\mathbf{rwlockwriteguard}(\alpha, \tau)]\!].\mathrm{own}(t, [\ell])$. □

Proof sketch of `rdguard_deref`. With the type `fn(&self) -> &T`, we get to assume $[\![\mathbf{rwlockreadguard}(\alpha, \tau)]\!].\mathrm{shr}(\kappa, t, \ell)$ and we need to produce $[\![\tau]\!].\mathrm{shr}(\kappa, t, \ell' + 1)$. This is straightforward because the latter is contained in the the former. □

Proof sketch of `wrguard_deref_mut`. With the argument `&mut self`, we get to assume $[\![\mathbf{rwlockwriteguard}(\alpha, \tau)]\!].\mathrm{own}(t, [\ell])$ under a full borrow bounded by some lifetime κ, and we need to produce, for `&mut T`, $[\![\tau]\!].\mathrm{own}(t, \ell + 1)$ also under a κ-bounded full borrow. Recall that in RustBelt, $[\![\&^{\kappa}_{\mathbf{mut}}\, \tau]\!].\mathrm{own}(t, \overline{v}) ::= \exists \ell.\, \overline{v} = [\ell] * \&^{\kappa}_{\mathbf{full}}(\ell \mapsto [\![\tau]\!].\mathrm{own}(t))$.

We know that $\kappa \sqsubseteq_{\mathrm{l}} \alpha$. Looking at the own interpretation for the writer lock guard, we have also $\alpha \sqsubseteq_{\mathrm{l}} \beta$ and $\&^{\beta}_{\mathbf{full}}(\ell + 1 \mapsto [\![\tau]\!].\mathrm{own}(t))$. So

we can *reborrow* that into $\&^{\kappa}_{\mathbf{full}}(\ell + 1 \mapsto [\![\tau]\!].\mathrm{own}(t))$. The remaining resources of the writer lock guard will be withheld by the inheritance of the reborrow, so that when the reborrow ends, those resources are returned so that we can reconstruct the own interpretation of the writer lock guard. (The RustBelt framework requires us to establish such reconstruction proof when performing the reborrow.) □

Proof sketch of **`rdguard_drop`**. We get to consume the reader lock guard, that is $[\![\mathbf{rwlockreadguard}(\alpha, \tau)]\!].\mathrm{own}(t, [\ell])$. This is the reverse of the **`try_read`** proof: we use the GPS protocol resources in the own interpretation for the lock guard to perform the CAS that decrements the lock counter, and we then return all resources from the lock guard's own interpretation to the protocol's write interpretation $\mathcal{I}_w(\tau, \alpha, \gamma_s, t)$. Additionally, we need to update the ghost state to decrement the ghost counter. In doing so, we deallocate our reader state $\boxed{\mathsf{Reading}(q, \nu)}^{\gamma_s}$.

In case we are the only existing reader lock guard, we kill the lifetime ν (using $\square\,\mathsf{Kill}(\nu)$), so that we can inherit $\&^{\alpha}_{\mathbf{full}}(\ell + 1 \mapsto [\![\tau]\!].\mathrm{own}(t))$, *i.e.*, switching the access mode of the underlying type τ from shared to owned. Note that in this case we also regain the full read and write permissions of the GPS protocol. All of those resources allow us to set the protocol ghost state back to $a = \mathsf{None}$, *i.e.*, to unlock the lock. □

Proof sketch of **`wrguard_drop`**. We get to consume the writer lock guard, $[\![\mathbf{rwlockwriteguard}(\alpha, \tau)]\!].\mathrm{own}(t, [\ell])$. The GPS protocol writer permission allows us to know that the protocol is in state $a = \mathsf{Some}(\mathsf{left}())$, as well as to simply perform a release write of 0 to release the lock. From the writer lock guard's own interpretation, we have all the resources needed to set the protocol ghost state back to $a = \mathsf{None}$. When performing the ghost update, we also deallocate our writer state $\boxed{\mathsf{Writing}}^{\gamma_s}$. □

CHAPTER SUMMARY. In this chapter, we sketched the semantic type checking of a λ_{Rust} version of Rust's reader-writer lock library (which employs **`unsafe`** blocks), using a combination of the relaxed lifetime logic and iRC11 GPS protocols.

19

Verification of `Arc`

The verification of the `Arc` library is by far the most challenging library verification in $\mathsf{RB}_{\mathtt{rlx}}$. In the original RustBelt work, the verification of `Arc` is split into 2 steps: first, the implementation is verified against a sufficiently strong abstract specification, and then the semantic type checking of `Arc`'s functions is done using only that abstract specification. While the latter task is no less challenging, we managed to reuse that part of proofs entirely for $\mathsf{RB}_{\mathtt{rlx}}$. Therefore, in this chapter, we focus on the first task: reverifying the now-relaxed implementation against the strong abstract specification. Fortunately, the specification is reasonably balanced to also admit the relaxed implementation, so we did not need to change it—we only need to redo the proofs in iRC11.

Regardless, to make the verification go through, we needed to strengthen two atomic reads from **`rlx`** to **`acq`** in the implementations of `Arc::get_mut` and `Arc::make_mut`. We conjecture that the relaxed access in `Arc::make_mut` is indeed sound but verifying this would have required a significantly more complex invariant. The relaxed access in `Arc::get_mut` turned out to be a bug. We provide more details about this bug in §19.4.

In §19.1, we present the core `Arc` implementation in λ_{Rust} + ORC11 and sketch how to verify it using cancelable GPS protocols in §19.2. In §19.3, we present the *full* APIs of `Arc`, which involves the associate type `Weak`, and sketch how to verify the full APIs. The synchronization bug and a bit of history how it was found is mentioned in §19.4.

19.1 The Core `Arc` library

`Arc<T>`, short for *Atomically Reference Counted*, is used to share atomically an object of type `T`, whose mutation is disabled by default. To mutate `T`, one needs `T` to support thread-safe mutability, for example with `T` being an atomic type, or with `T` wrapped inside a lock (*e.g.*, `Mutex<T>`).

Example 19.1 (A Client of `Arc`)**.** The following Rust example instantiates `Arc` with an atomic integer `AtomicUsize` and demonstrates how `Arc` is typically used:

```
1  let arc1 = Arc::new(AtomicUsize::new(5)); // create the first Arc
2  let arc2 = Arc::clone(&arc1); // clone the second Arc
3  thread::spawn(move || { // give arc2 to child thread
4    println!("child: {:?}", arc2.fetch_add(1, Ordering::Relaxed));
```

```
5     // drop(arc2);
6   });
7   println!("main: {:?}", arc1.fetch_add(2, Ordering::Relaxed));
8   // drop(arc1);
```

In line 1 in the main thread, a new Arc pointer arc1 is created to govern an atomic integer allocated in shared memory. The Arc's internal counter field for the number of references to the content is set to 1. An Arc pointer acts almost like its underlying content, so in line 4 we can call fetch_add on arc1 as if on the atomic integer itself. To share the content with the child thread, we create another arc2 by clone-ing arc1 (line 2), which effectively increments the internal counter to 2: there are now 2 pointers sharing the atomic integer. Unsurprisingly, to allow concurrent updates by multiple threads, the internal counter field is implemented with an atomic integer.

When the Arc pointers go out of scope, in lines 5 and 8, their destructors—the drop function—are called and the counter field is decremented accordingly. The last call of drop will deallocate the underlying content *and* the counter field.

The λ_{Rust} + ORC11 implementation of Core Arc's functions is given in Figure 19.1. The new function allocates two locations, one for the counter field and one for the data field, then initializes them. Note that we present Core Arc as a simplification of the actual implementation verified in our Coq development: we assume here that the type T only has size 1, but the actual implementation allocates $[\![\tau]\!]$.size locations (memory cells) to be able to store T. Furthermore, the actual implementation needs to consume all of those memory cells (copying them into the allocated memory and freeing the memory for the argument of type T), instead of just assuming that the argument is just a simple value v.

The deref function provides access to the data field, effectively allowing an Arc<T> to behave like its content T. The clone function does a relaxed (**rlx**) *fetch-and-add* (**FAA**) by 1 to increment counter and then return a copy of a.

Finally, the drop function does a *release* (**rel**) **FAA** to decrement counter. If the value of counter was 1 before the decrement (*i.e.*, this is the last drop), drop additionally does an acquire (**acq**) fence before deallocating both the counter and data fields. The acquire fence is needed because the release **FAA**, although being a release write, is only a relaxed read.

CORRECTNESS. Intuitively, the main correctness guarantee of Core Arc is that the deallocation of its data and counter fields is synchronized with (happens-after) *all* accesses to those fields. Those accesses happen between (and including) the construction of an Arc pointer, either by new or clone, and its destruction by drop. Therefore, the correctness guarantee translates to making sure that the deallocation done by the last drop is synchronized with all previous drop's. In this case, that synchronization is established between the release **FAA**'s of all previous drop's and the acquire fence of the last drop.

$$
\begin{aligned}
\texttt{new}(v) ::= {} & \textbf{let}\ \texttt{a} := \textbf{alloc}(2)\ \textbf{in} \\
& \texttt{a.counter} :=_{\textbf{na}} 1; \\
& \texttt{a.data} :=_{\textbf{na}} v; \\
& \texttt{a} \\
\texttt{deref}(\texttt{a}) ::= {} & {}^{*\textbf{na}}\texttt{a.data}
\end{aligned}
\qquad
\begin{aligned}
\texttt{drop}(\texttt{a}) ::= {} & \textbf{if}\ \textbf{FAA}^{\textbf{rel}}(\texttt{a.counter}, -1) == 1 \\
& \quad \textbf{fence}_{\textbf{acq}}; \\
& \quad \textbf{free}(\texttt{a}, 2) \\
\texttt{clone}(\texttt{a}) ::= {} & \textbf{FAA}^{\textbf{rlx}}(\texttt{a.counter}, 1); \\
& \texttt{a}
\end{aligned}
$$

FIGURE 19.1: Implementation of Core Arc

19.2 Verification of Core Arc with Cancelable GPS Protocols

We demonstrate the verification of the most important functions of Core Arc: new, clone and drop. For clone, we need to guarantee that any newly-created pointer arc to an object a can *non-atomically read* its data field a.data (so that the deref function can be called on arc), and perform *atomic* **FAA**'s on its counter field a.counter (so that clone and drop can be called on arc). This means that both fields must be *shared* for concurrent accesses by multiple threads.

For drop, we instead show that this sharing of the fields must have been finished before the deallocation is called. To deallocate a block a of two locations using $\textbf{free}(\texttt{a}, 2)$, the deallocation rule NA-DEALLOC (Figure 9.1, §9.1) requires us to have the full ownership of the whole block *i.e.*, both $\texttt{a.data} \mapsto \_$ and $\texttt{a.counter} \mapsto \_$.

In short, we start out with the full ownership $\texttt{a.data} \mapsto v$ and $\texttt{a.counter} \mapsto 1$ in the new function, then we share both a.data and a.counter for concurrent accesses, and at the end reclaim both $\texttt{a.data} \mapsto \_$ and $\texttt{a.counter} \mapsto \_$ in the last drop for deallocation. Our job is to set up the sharing to satisfy this scheme. Because the data field only needs concurrent *read* accesses, we employ *fractional* ownership[1] on the points-to assertion $\texttt{a.data} \mapsto v$.[2] That is, we start out with the full fraction $\texttt{a.data} \mapsto v = \texttt{a.data} \mapsto v$ and for every newly-created pointer we give it a small fraction $\texttt{a.data} \overset{q}{\mapsto} v$, which is sufficient to perform concurrent reads. When a pointer goes out of scope, its small fraction $\texttt{a.data} \overset{q}{\mapsto} v$ is recollected. Before the very end, we recollect all the small fractions into the full fraction $\texttt{a.data} \mapsto v = \texttt{a.data} \mapsto v$. Then we are ready for deallocating a.data.

The counter field, on the other hand, needs concurrent **FAA** accesses, so we will use a cancelable GPS protocol (Definition 17.4) to share it. The cancelable protocol is also used for recollecting the small fractions of the data field.

[1] Boyland, "Checking interference with fractional permissions" [Boy03].

[2] To allow concurrent atomic writes to the underlying object of type T with only some fractional ownership, the type needs to support interior mutability. This can be achieved if we can *e.g.*, also use a cancelable GPS protocol for T.

19.2.1 *The Abstract Specifications for Core Arc*

We want to prove the following simple specification for Core Arc:

$$\{\mathsf{True}\}\ \texttt{new}(v)\ \{\texttt{a}.\ \exists \gamma_i, \gamma.\ \mathsf{ARC}^{\gamma_i,\gamma}(\texttt{a}, v, \mathcal{I})\} \quad \text{(ARC-NEW-SPEC)}$$

$$\{\mathsf{ARC}^{\gamma_i,\gamma}(\texttt{a}, v, \mathcal{I})\}\ \texttt{clone}(\texttt{a})\ \{\mathsf{ARC}^{\gamma_i,\gamma}(\texttt{a}, v, \mathcal{I}) * \mathsf{ARC}^{\gamma_i,\gamma}(\texttt{a}, v, \mathcal{I})\} \quad \text{(ARC-CLONE-SPEC)}$$

$$\{\mathsf{ARC}^{\gamma_i,\gamma}(\texttt{a}, v, \mathcal{I})\}\ \texttt{drop}(\texttt{a})\ \{\mathsf{True}\} \quad \text{(ARC-DROP-SPEC)}$$

where the abstract predicate $\mathsf{ARC}^{\gamma_i,\gamma}(\mathtt{a}, v, \mathcal{I})$ represents the logical ownership of an Arc pointer to some object a whose data is v. The arguments γ_i, γ, and $\mathcal{I}$ are used for the cancelable GPS protocol.

ARC-NEW-SPEC says that new(v) allocates a new Arc pointer, and ARC-CLONE-SPEC says that clone can create two pointers from one. ARC-DROP-SPEC simply says that a drop is always safe with just a Arc pointer ownership.

19.2.2 *The GPS Protocol for Core* Arc

Definition 19.2 (Model of the ARC Assertion)**.** We define the abstract predicate $\mathsf{ARC}^{\gamma_i,\gamma}(\mathtt{a}, v, \mathcal{I})$ as follows:

$$\mathsf{ARC}^{\gamma_i,\gamma}(\mathtt{a}, v, \mathcal{I}) ::= \exists q.\, \mathtt{a.data} \overset{q}{\mapsto} v * \boxed{(\mathtt{a.counter}, \gamma_i, \_) : \_ \mid \mathcal{I}^{\gamma,v}}_{q} * \heartsuit^{\gamma_i}_{q} * \boxed{\mathsf{Count}(q)}^{\gamma}$$

Owning an Arc pointer $\mathsf{ARC}^{\gamma_i,\gamma}(\mathtt{a}, v, \mathcal{I})$ means that we own: (1) some small fraction q of the data field $\mathtt{a.data} \overset{q}{\mapsto} v$ at the value v, which allows us to safely read a.data for the value v;[3] (2) the CAS-only assertion $\boxed{(\mathtt{a.counter}, \gamma_i, \_) : \_ \mid \mathcal{I}^{\gamma,v}}_{q}$ that the counter field is governed by a protocol $\mathcal{I}$ protected by γ_i, as well as the cancelable invariant token $\heartsuit^{\gamma_i}_{q}$—with the *same* fraction q—to know that the protocol is still alive, which allows us to concurrently access a.counter; and finally (3) an *unsynchronized* ghost element $\boxed{\mathsf{Count}(q)}^{\gamma}$ that represent the 1 *single* count of this pointer in the *total* count (see below).

[3] In the general case, this would be fractional ownership of the underlying type τ's interpretation, which may allow more than read access.

Definition 19.3 (GPS Protocol for a.counter)**.** The protocol for the location a.counter is defined as follows:

$$\begin{aligned}
\mathcal{I}^{\gamma,v} &::= (\mathcal{I}_w^{\gamma,v}, \mathcal{I}_r, \mathcal{I}_m) \\
\mathcal{I}_r(\mathtt{a}, \gamma_\mathtt{a}, _, _, n) &::= n \geq 0 \\
\mathcal{I}_m(\mathtt{a}, \gamma_\mathtt{a}, _, _, n) &::= n = 0 \\
\mathcal{I}_w^{\gamma,v}(\mathtt{a}, \gamma_\mathtt{a}, _, _, n) &::= \begin{cases} \mathsf{False} & n < 0 \\ \boxed{\mathsf{TotalCount}(0,0)}^{\gamma} & n = 0 \\ \exists q_{\text{in}}, q_{\text{out}} \in (0,1).\, \mathtt{a.data} \overset{q_{\text{in}}}{\mapsto} v * \heartsuit^{\gamma_i}_{q_{\text{in}}} & \\ \quad * \mathcal{W}_{\text{shr}}(\mathtt{a}, \gamma_\mathtt{a}, _, _, n) * \mathcal{R}^{q_{\text{in}}}_{\text{shr}}(\mathtt{a}, \gamma_\mathtt{a}, _, _, n) & n > 0 \\ \quad * q_{\text{in}} + q_{\text{out}} = 1 * \boxed{\mathsf{TotalCount}(n, q_{\text{out}})}^{\gamma} & \end{cases}
\end{aligned}$$

(ARC-INV)

First of all, $\mathcal{I}$ requires that the value v of the counter field to be non-negative. When it is positive *i.e.*, when there is some Arc pointers, the number of pointers is n and the write interpretation $\mathcal{I}_w^{\gamma,v}$ owns the unsynchronized ghost element $\boxed{\mathsf{TotalCount}(v, q_{\text{out}})}^{\gamma}$. We use this ghost element to track the protocol state, and so have no need for the GPS protocol type, which we pick to be the trivial unit protocol $S_{()}$.

The element $\boxed{\mathsf{TotalCount}(v, q_{\text{out}})}^{\gamma}$ tracks the globally-consistent knowledge that there are currently n pointers and the *sum* of all fractional permissions owned by those pointers is q_{out}.[4] The write interpretation further requires that the remaining fraction $q_{\text{in}} = 1 - q_{\text{out}}$ must be owned

[4] Here, *out* means ownership of the fractions outside of $\mathcal{I}_w^{\gamma,v}$.

COUNTING-START
$$q \in (0,1] \vdash \dot{\Rightarrow}\exists\gamma.\, \boxed{\mathsf{TotalCount}(1,q)}^{\gamma} * \boxed{\mathsf{Count}(q)}^{\gamma}$$

COUNTING-NEW
$$q + q' \leq 1 \vdash \boxed{\mathsf{TotalCount}(n,q)}^{\gamma} \dot{\Rightarrow} \boxed{\mathsf{TotalCount}(n+1,q+q')}^{\gamma} * \boxed{\mathsf{Count}(q')}^{\gamma}$$

COUNTING-AGREE
$$\boxed{\mathsf{TotalCount}(n,q)}^{\gamma} * \boxed{\mathsf{Count}(q')}^{\gamma} \vdash n \geq 1 \land q' \leq q \leq 1$$

COUNTING-DROP
$$\boxed{\mathsf{TotalCount}(n+1,q+q')}^{\gamma} * \boxed{\mathsf{Count}(q')}^{\gamma} \dot{\Rightarrow} \boxed{\mathsf{TotalCount}(n,q)}^{\gamma} \land (n = 0 \Rightarrow q = 0)$$

FIGURE 19.2: Counting permissions for Core Arc

by the protocol. This includes the fractional ownership of a.data and the cancelable token $\heartsuit^{\gamma_i}_{q_{\text{in}}}$ of a.counter. The fraction q_{in} in fact includes the *used* fraction of pointers that have been dropped. Thus the protocol makes sure that any fractions of the a.data and γ_i are all accounted for.

Finally, when the counter reaches 0, the protocol is simply trivial with the ghost element also being 0. Note that the move interpretation $\mathcal{I}_m$ allows only for this case, which means that we can perform a single-writer write once the counter reaches 0. This is simply because there is no more sharing in this case. Such a single-writer write can be useful in resetting and reusing the reference counting, which will be needed in Full Arc (§19.3).

The ghost elements $\boxed{\mathsf{TotalCount}(n,q)}^{\gamma}$ and $\boxed{\mathsf{Count}(q)}^{\gamma}$ is an instance of *counting permissions*,[5] used here to track the outside fractions associated with each single count. They satisfy the axioms in Figure 19.2. COUNTING-START creates a ghost location γ for the first count and gives us the total count $\boxed{\mathsf{TotalCount}(1,q)}^{\gamma}$ as well as a single count $\boxed{\mathsf{Count}(q)}^{\gamma}$. With COUNTING-NEW we can increase the total count and produce more single counts. With COUNTING-DROP we can decrease the total count by consuming single counts. COUNTING-AGREE ensures that every single count is always included in the total count. How this ghost construction comes into play will be revealed next section.

[5] Bornat et al., "Permission accounting in separation logic" [Bor+05].

After this long setup, we are finally ready to explain the verification of Core Arc.

19.2.3 *Verifying* new

In the proof of ARC-NEW-SPEC, we elide the standard allocation and initialization of the data and counter fields. Our main obligation here is to transform the two full ownership $\mathtt{a.data} \mapsto v$ and $\mathtt{a.counter} \mapsto 1$ to the abstract permission $\mathsf{ARC}^{\gamma_i,\gamma}(\mathtt{a}, v, \mathcal{I})$ for some γ_i and γ. That is, turning our unique ownership into sharing mode.

To do so, we initialize an iRC11 cancelable single-writer protocol for a.counter, using the rule GPS-SW-INIT.[6] The rule gives us fresh ghost locations γ_i and $\gamma_{\mathtt{a}}$ and the full cancelable token $\heartsuit^{\gamma_i}_1$. What we need to provide are the points-to $\mathtt{a.counter} \mapsto 1$, which we have, and the write

[6] see Figure 17.4

interpretation $\mathcal{I}_w^{\gamma,v}(\mathtt{a}, \gamma_{\mathtt{a}}, \_, \_, 1)$ for some γ. Below, we write $\mathcal{I}_w^{\gamma,v}(1)$ as a short-hand notation for $\mathcal{I}_w^{\gamma,v}(\mathtt{a}, \gamma_{\mathtt{a}}, \_, \_, 1)$.

For γ, we use COUNTING-START to create the total count and the first single count with $q ::= 1/2$. That is, we get $\boxed{\mathsf{TotalCount}(1, 1/2)}^{\gamma}$ and $\boxed{\mathsf{Count}(1/2)}^{\gamma}$. We use $\boxed{\mathsf{TotalCount}(1, 1/2)}^{\gamma}$ for $\mathcal{I}_w$ and $\boxed{\mathsf{Count}(1/2)}^{\gamma}$ for ARC. Similarly, we split $\mathtt{a.data} \mapsto v$ into two halves $\mathtt{a.data} \overset{1/2}{\longmapsto} v$'s and use each for $\mathcal{I}$ and ARC.

With $\mathtt{a.data} \overset{1/2}{\longmapsto} v$ and $\boxed{\mathsf{TotalCount}(1, 1/2)}^{\gamma}$, to establish $\mathcal{I}^{\gamma,v}(1)$, we need $\heartsuit^{\gamma_i}_{1/2}$, and $\mathcal{W}_{\mathsf{shr}}(\mathtt{a}, \gamma_{\mathtt{a}}, \_, \_, 1)$, and $\mathcal{R}^{1/2}_{\mathsf{shr}}(\mathtt{a}, \gamma_{\mathtt{a}}, \_, \_, 1)$. Unfortunately, GPS-SW-INIT does not allow us to use some of the cancelable token nor the single-writer permission $\mathcal{W}(\ell, \gamma, t, s, v)$ to establish $\mathcal{I}_w$. Therefore, we need to use a more general initialization rule that has the flavors of both CINV-ALLOC-FRAC[7] and GPS-MID-SW-INIT-STRONG[8] combined. Concretely, we want keep half of cancelable token ($\heartsuit^{\gamma_i}_{1/2}$) locally, and use the other half for $\mathcal{I}_w^{\gamma,v}(1)$. Furthermore, we use the rule GPS-W-WSHR-RSHR[9] to turn the single-writer permission $\mathcal{W}(\mathtt{a}, \gamma_{\mathtt{a}}, \_, \_, 1)$ into the shared-writer permission $\mathcal{W}_{\mathsf{shr}}(\mathtt{a}, \gamma_{\mathtt{a}}, \_, \_, 1)$ and the full shared-reader permission $\mathcal{R}^{1}_{\mathsf{shr}}(\mathtt{a}, \gamma_{\mathtt{a}}, \_, \_, 1)$. We use the former and *half* $(1/2)$ of the latter to complete $\mathcal{I}_w^{\gamma,v}(1)$.

As a result, we get $\heartsuit^{\gamma_i}_{1/2} * \boxed{(\mathtt{a.counter}, \gamma_i, \gamma_{\mathtt{a}}) : \_ \mid \mathcal{I}^{\gamma,v}}_{1/2}$. We combine these with the remaining $\mathtt{a.data} \overset{1/2}{\longmapsto} v$ and $\boxed{\mathsf{Count}(1/2)}^{\gamma}$ to complete the first $\mathsf{ARC}^{\gamma_i,\gamma}(\mathtt{a}, v, \mathcal{I})$ permission. □

[7] see Figure 11.3

[8] see Figure 17.8

[9] see Figure 17.3

19.2.4 *Verifying* `clone`

In proving ARC-CLONE-SPEC, we need to split one $\mathsf{ARC}^{\gamma_i,\gamma}(\mathtt{a}, v, \mathcal{I})$ into two. Unfolding the definition of $\mathsf{ARC}^{\gamma_i,\gamma}(\mathtt{a}, v, \mathcal{I})$,[10] we see that the fractions $\mathtt{a.data} \overset{q}{\mapsto} v$, $\boxed{(\mathtt{a.counter}, \gamma_i, \_) : \_ \mid \mathcal{I}^{\gamma,v}}_{q}$, and $\heartsuit^{\gamma_i}_{q}$ (for some q) can be split into halves *i.e.*, $\mathtt{a.data} \overset{q/2}{\longmapsto} v$, $\boxed{(\mathtt{a.counter}, \gamma_i, \_) : \_ \mid \mathcal{I}^{\gamma,v}}_{q/2}$, and $\heartsuit^{\gamma_i}_{q/2}$, each for one new ARC. So we only need to transform one single count $\boxed{\mathsf{Count}(q)}^{\gamma}$ into two. To match the fraction $q/2$, we actually need two single counts of the form $\boxed{\mathsf{Count}(q/2)}^{\gamma}$. Unfortunately, $\boxed{\mathsf{Count}(q)}^{\gamma}$ is *not* splittable into two $\boxed{\mathsf{Count}(q/2)}^{\gamma}$'s. So we can only get two $\boxed{\mathsf{Count}(q/2)}^{\gamma}$'s with the help of the total count $\boxed{\mathsf{TotalCount}(\_, \_)}^{\gamma}$, which is inside the GPS protocol. To do so, at the relaxed **FAA** made by `clone` (Figure 19.1), we invoke the CAS rule for the single-writer protocol in the *CAS-only* mode. We did not present this rule, but it is very much the same as the rule GPS-ATBOR-WSHR-CAS-INT[11] for the Atomic-Borrows-based single-writer variant of GPS protocols, where we only need to replace the lifetime token fraction $[\kappa]_q$ with the cancelable token $\heartsuit^{\gamma_i}_{q}$—which we do have from $\mathsf{ARC}^{\gamma_i,\gamma}(\mathtt{a}, v, \mathcal{I})$.

Note that, in `clone`, we use a *relaxed* **FAA** which is a relaxed read and a relaxed write. Therefore, in the successful case of the **FAA**, we only get to access the write interpretation $\mathcal{I}_w^{\gamma,v}$ under the release modality Δ and we can only get some resource out under the acquire modality ∇.[12] But remember that, if our resource is, however, view-agnostic—for example, if they are unsynchronized ghost state—then the fence modalities can be bypassed. We exploit this in our invocation of the CAS rule for `clone`.

[10] see Definition 19.2

[11] see Figure 17.7

[12] see §8.3

In particular, as we have $\boxed{\mathsf{Count}(q)}^{\gamma}$, we use GHOST-RELMOD[13] to get $\Delta\boxed{\mathsf{Count}(q)}^{\gamma}$.

[13] see Figure 8.1

Then, we now have to show that

$\mathcal{I}_w^{\gamma,v}(n) * \boxed{\mathsf{Count}(q)}^{\gamma} \dot{\Rrightarrow} \mathcal{I}_w^{\gamma,v}(n+1) * \boxed{\mathsf{Count}(q/2)}^{\gamma} * \boxed{\mathsf{Count}(q/2)}^{\gamma}$,

where n is the value the **FAA** reads from `a.counter`.

First, by the definition of $\mathcal{I}_w^{\gamma,v}(n)$ (see ARC-INV), we know that $n \geq 0$. By owning $\boxed{\mathsf{Count}(q)}^{\gamma}$, we also know that n cannot be 0, because if $n = 0$, we can combine $\boxed{\mathsf{TotalCount}(0,0)}^{\gamma}$ with $\boxed{\mathsf{Count}(q)}^{\gamma}$ and use the rule COUNTING-AGREE to derive the contradiction that $0 \geq 1$. Thus $n > 0$.

Now, we are not going to change the fractions ($q_{\mathrm{in/out}}$) and the fractional ownerships: we will keep them the same (*i.e.*, *framing*) for $\mathcal{I}^{\gamma,v}(n+1)$. Therefore our job is simply transform $\boxed{\mathsf{TotalCount}(n, q_{\mathrm{out}})}^{\gamma} * \boxed{\mathsf{Count}(q)}^{\gamma}$ to $\boxed{\mathsf{TotalCount}(n+1, q_{\mathrm{out}})}^{\gamma} * \boxed{\mathsf{Count}(q/2)}^{\gamma} * \boxed{\mathsf{Count}(q/2)}^{\gamma}$. This is simple: we first use COUNTING-DROP to drop the single count $\boxed{\mathsf{Count}(q)}^{\gamma}$ associated with q and get $\boxed{\mathsf{TotalCount}(n-1, q_{\mathrm{out}} - q)}^{\gamma}$. We then call COUNTING-NEW twice on $\boxed{\mathsf{TotalCount}(n-1, q_{\mathrm{out}} - q)}^{\gamma}$, each time creating a new single count $\boxed{\mathsf{Count}(q/2)}^{\gamma}$ and in the end we get back $\boxed{\mathsf{TotalCount}(n+1, q_{\mathrm{out}})}^{\gamma}$. Note that we always satisfy the side condition of COUNTING-NEW because $q_{\mathrm{out}} \leq 1$.

Finally, after the access, we get back the access token $\heartsuit_q^{\gamma_i}$ and two single counts under the acquire modality:

$$\nabla\left(\boxed{\mathsf{Count}(q/2)}^{\gamma} * \boxed{\mathsf{Count}(q/2)}^{\gamma}\right)$$

Since the single counts are unsynchronized ghost state, we use ACQMOD-GHOST[14] to get $\boxed{\mathsf{Count}(q/2)}^{\gamma} * \boxed{\mathsf{Count}(q/2)}^{\gamma}$. Now we can split the token $\heartsuit_q^{\gamma_i}$ and the fraction ownership $\mathtt{a.data} \overset{q}{\mapsto} v$ into two halves and gain two $\mathsf{ARC}^{\gamma_i,\gamma}(\mathtt{a}, v, \mathcal{I})$'s. □

[14] see Figure 8.1

19.2.5 *Verifying* `drop`

The first intuition in the proof of `drop` is that, if the `drop` is not the last drop, we will return all the resources of the current pointer $\mathsf{ARC}^{\gamma_i,\gamma}(\mathtt{a}, v, \mathcal{I})$ to the protocol. This includes the fractional ownership $\mathtt{a.data} \overset{q}{\mapsto} v$, the shared-reader permission $\boxed{(\mathtt{a.counter}, \gamma_i, \_) : \_ \mid \mathcal{I}^{\gamma,v}}_q$, the cancelable token $\heartsuit_q^{\gamma_i}$ and the single count element $\boxed{\mathsf{Count}(q)}^{\gamma}$. The former three will be stored in the protocol and will be transferred to the last `drop` for deallocation. The single count element will be used to decrease the total count by 1.

The second intuition is that, in the case of the last `drop`, we know from the ARC permission and the protocol's write interpretation that the local fractions and the fractions stored in the protocol sum up to 1, so we can recollect the full fraction for deallocation.

In both cases, we need a stronger rule for *release* **FAA** that allow us to use the token $\heartsuit_q^{\gamma_i}$ to access the protocol and *simultaneously* use the token to establish the write interpretation. Consequently, we would not regain $\heartsuit_q^{\gamma_i}$ in the post-condition of such a rule. While we do not write down this rule concretely, it has the same flavor as the release single-writer rule GPS-SW-WRITE-REL.[15]

[15] see Figure 17.4

Now, at the release **FAA** of drop, using $\heartsuit_q^{\gamma_i}$, we have the following goal $\heartsuit_q^{\gamma_i} * P * \mathcal{I}^{\gamma,v}(n) \Rrightarrow \mathcal{I}^{\gamma,v}(n-1) * Q(n)$ where n is the old value of a.counter and

$$P ::= \texttt{a.data} \overset{q}{\mapsto} v * \boxed{(\texttt{a.counter}, \gamma_i, \_) : \_ \mid \mathcal{I}^{\gamma,v}}_{q} * \boxed{\mathsf{Count}(q)}^{\gamma}$$

$$Q(n) ::= \begin{cases} \mathsf{True} & n \neq 1 \\ \texttt{a.data} \mapsto v * \heartsuit_1^{\gamma_i} & n = 1 \\ \quad * \mathcal{W}_{\mathsf{shr}}(\texttt{a}, \gamma_{\texttt{a}}, \_, \_, 0) & \\ \quad * \boxed{(\texttt{a.counter}, \gamma_i, \_) : \_ \mid \mathcal{I}^{\gamma,v}}_{1} & \end{cases}$$

That is, P are the resources we have locally in our pre-condition, and Q are the resources we will have in our post-condition of the rule.

Similarly to the reasoning in clone, with $\boxed{\mathsf{Count}(q)}^{\gamma}$ from P, we know that $n > 0$ and the invariant has some fractional permissions $\heartsuit_{q_{\text{in}}}^{\gamma_i}$ and $\texttt{a.data} \overset{q_{\text{in}}}{\mapsto} v$ for some q_{in} (see ARC-INV).

Now, if this is not the last drop *i.e.*, $n - 1 > 0$, we need to re-establish $\mathcal{I}^{\gamma,v}(n-1)$ with some new fractions $q'_{\text{in/out}}$. We pick them as follows: $q'_{\text{in}} ::= q_{\text{in}} + q$ and $q'_{\text{out}} ::= q_{\text{out}} - q$. From $\heartsuit_q^{\gamma_i} * P * \mathcal{I}^{\gamma,v}(n)$, we can easily get $\heartsuit_{q'in}^{\gamma_i} = \heartsuit_{q_{\text{in}}}^{\gamma_i} * \heartsuit_q^{\gamma_i}$ and $\texttt{a.data} \overset{q'_{\text{in}}}{\mapsto} v = \texttt{a.data} \overset{q_{\text{in}}}{\mapsto} v * \texttt{a.data} \overset{q}{\mapsto} v$, as well as $\boxed{(\texttt{a.counter}, \gamma_i, \_) : \_ \mid \mathcal{I}^{\gamma,v}}_{q'_{\text{in}}}$, which are needed for $\mathcal{I}^{\gamma,v}(n-1)$. Our remaining work is to transform $\boxed{\mathsf{TotalCount}(n, q_{\text{out}})}^{\gamma} * \boxed{\mathsf{Count}(q)}^{\gamma}$ to $\boxed{\mathsf{TotalCount}(n-1, q'_{\text{out}})}^{\gamma}$. Fortunately, this is but a simple application of COUNTING-DROP. Then we are done because $n \neq 1$.

In the case where this is the last drop's **FAA**, we have $n = 1$ and we must prove $\heartsuit_q^{\gamma_i} * P * \mathcal{I}^{\gamma,v}(1) \Rrightarrow \mathcal{I}^{\gamma,v}(0) * Q(1)$. From $\mathcal{I}^{\gamma,v}(1)$ we have $\boxed{\mathsf{TotalCount}(1, q_{\text{out}})}^{\gamma}$ and from P we have $\boxed{\mathsf{Count}(q)}^{\gamma}$. By an application of COUNTING-DROP, we have $\boxed{\mathsf{TotalCount}(0, 0)}^{\gamma}$, which is exactly $\mathcal{I}^{\gamma,v}(0)$, and additionally the fact that $q_{\text{out}} = q$. From $\mathcal{I}^{\gamma,v}(1)$ we also know that $q_{\text{in}} + q_{\text{out}} = q_{\text{in}} + q = 1$. Thus combining what we have left from our assumption $\heartsuit_q^{\gamma_i} * P * \mathcal{I}^{\gamma,v}(1)$, we have exactly $Q(1)$. So we finish the last drop's **FAA** and gain $\nabla Q(1)$.

As the return value is $n = 1$, we perform an acquire fence (see the code of drop in Figure 19.1). Thanks to the acquire fence rule HOARE-ACQ-FENCE,[16] we remove the modality and regain $Q(1)$. We are almost done: we only need to get back the points-to ownership of a.counter. For this, we *cancel* the GPS single-writer protocol for a.counter using the cancellation rule GPS-SW-DEALLOC.[17] The rule requires the full token $\heartsuit_1^{\gamma_i}$, which we do have, to ensure that the cancellation happens after all accesses to the protocol. At long last, after the cancellation we now have the full ownership of both fields and can safely use NA-DEALLOC to free them. □

Note that in this proof, before applying GPS-SW-DEALLOC, we can first use the rule GPS-SW-W-REVERT[18] to regain the single-writer assertion $\boxed{(\texttt{a.counter}, \gamma_i, \_) : \_ \mid \mathcal{I}^{\gamma,v}}_{W}$. With that and a *stronger* variant of GPS-SW-DEALLOC,[19] we can regain exactly $\mathcal{I}_w^{\gamma,v}(\texttt{a}, \gamma_{\texttt{a}}, \_, \_, 0)$ which will allow us to reuse the counting permissions at the ghost location γ, if needed.

[16] see Figure 8.1

[17] see Figure 17.4

[18] see Figure 17.3

[19] see the discussion for GPS-SW-DEALLOC in §17.1.2

Arc		Weak	
new:	`fn(T) -> Arc<T>`	new:	`fn() -> Weak<T>`
deref:	`fn(&Arc<T>) -> &T`		
clone:	`fn(&Arc<T>) -> Arc<T>`	clone:	`fn(&Weak<T>) -> Weak<T>`
downgrade:	`fn(&Arc<T>) -> Weak<T>`	upgrade:	`fn(&Weak<T>) -> Option<Arc<T>>`
drop:	`fn(Arc<T>) -> ()`	drop:	`fn(Weak<T>) -> ()`
get_mut:	`fn(&mut Arc<T>) -> Option<&mut T>`		
make_mut:	`fn(&mut Arc<T>) -> &mut T`		

TABLE 19.1: An excerpt of Rust's `Arc<T>` and `Weak<T>` APIs

19.3 Verification of `Arc`'s Full APIs

We discuss the verification of an extended version of `Arc`, which is also the version we have verified in $\text{RB}_{\texttt{rlx}}$. Its most interesting APIs are given in Table 19.1. Here we need to tackle two extra sets of behaviors, presented as two following challenges.

`Arc<T>` HAS A SUBORDINATE TYPE `Weak<T>`. The first challenge involves a type called `Weak<T>`. `Weak` itself is a variant of `Arc`: it has a counter to count how many `Weak` pointers are in existence, and also has the similar `clone` and `drop` functions.[20] However, `Weak` does not guarantee access to the underlying object of type `T`: while owning an `Arc` guarantees that the object is still available, owning a `Weak` does not prevent the object to be reclaimed. In order to access the object with a `Weak` pointer, one first calls `Weak::upgrade` to obtain an `Arc` pointer. `upgrade` can fail when the object has already been reclaimed, that is when there is no `Arc` pointer left. A `Weak` pointer are typically created by calling `Arc::downgrade` on a shared reference of `Arc`.

[20] see Table 19.1

The challenge for verifying `Arc` and `Weak` in the relaxed memory setting is that they involve two tightly coupled atomic locations—one for each counter. As multi-location RMC invariants (as presented in Chapter 11) are consolidated at a much later time after the $\text{RB}_{\texttt{rlx}}$ work, we needed to use separate GPS protocols for each counter and at the same time maintain their relation. This was a known challenge, as had been observed earlier by GPS.[21] The general solution is to construct ghost state to encode the relation between the locations and prevent their protocols from breaking the relation. We were able to set up several unsynchronized ghost state constructions to encode the relation, but those, unfortunately, are not enough.

[21] Turon et al., "GPS: navigating weak memory with ghosts, protocols, and separation" [TVD14].

`Arc<T>` SUPPORTS TEMPORARY BORROWS OF THE UNDERLYING CONTENT. The second challenge involves the support to temporarily reclaim full ownership of the underlying content when the thread knows it owns the last unique `Arc` and `Weak` pointers. The functions `Arc::get_mut` and `Arc::make_mut` provide these capabilities: they return a mutable reference `&mut T` to the underlying content. The reclamation is temporary because when the reference goes out of scope (when the lifetime of the mutable reference ends), the content is returned and the original `Arc`

```
fn is_unique(&mut self) -> bool {
  // lock the weak pointer count if we appear to be the sole weak pointer holder.
  if self.inner().weak.compare_exchange(1, usize::MAX, Acquire, Relaxed).is_ok() {
    let unique = self.inner().strong.load(Relaxed) == 1;

    self.inner().weak.store(1, Release); // release the lock
    unique
  } else { false }
}
fn get_mut(this: &mut Self) -> Option<&mut T> {
  if this.is_unique() {
    unsafe { Some(&mut this.ptr.as_mut().data) }
  } else { None }
}
fn drop(&mut self) {
  if self.inner().strong.fetch_sub(1, Release) != 1 {
    return;
  } ...
}
```

TABLE 19.2: Rust's implementation (excerpt) of `Arc::get_mut` and `Arc::drop`

pointer can be used again.

The challenge here is to guarantee that if the temporary reclamation is successful, it is synchronized with *all* accesses to the content of type `T`. Again, note that those accesses can only happen between the construction and the destruction of an `Arc` pointer. How an `Arc` pointer can be constructed is now more complicated than that of Core `Arc`: an `Arc` pointer can now additionally be created by `upgrade`-ing from a `Weak` pointer. Therefore, to establish the synchronization guarantee, we now need to handle the intertwined life-cycles of `Arc` and `Weak` pointers.

To be more concrete, let us look at the implementation of `get_mut` in Table 19.2. To return temporary full ownership of the `data` field, the function checks that the thread owns the unique `Arc` and `Weak` pointers in two steps, using `is_unique`.

First, it acquires a "lock" on the `Weak` counter—`a.weak`—to make sure that there is no other `Weak` pointers. This is done by an acquire compare-and-swap (**`CAS`**) from 1 to -1. The function uses -1 as the "locked" value to resolve conflicts with other contentious `Arc::get_mut` or `Arc::downgrade` calls. If the **`CAS`** succeeds, the thread knows that there is no `Weak` pointers left, but there may exist still some `Arc` pointers. This comes from the agreed contract between the counters: the `Weak` counter *implicitly* counts 1 for *all* `Arc` pointers. So when the thread still owns an `Arc` pointer, and the value of the `Weak` counter is exactly 1, that 1 must be accountable for the remaining `Arc` pointers, and there is no `Weak` pointers left.

Second, it does an acquire read on the `Arc` counter—`a.strong`—and then checks if the value read is 1. If that value is 1, `is_unique` succeeds and `get_mut` concludes that thread owns the unique `Arc` pointer, and gives the thread temporary full access to the underlying content with

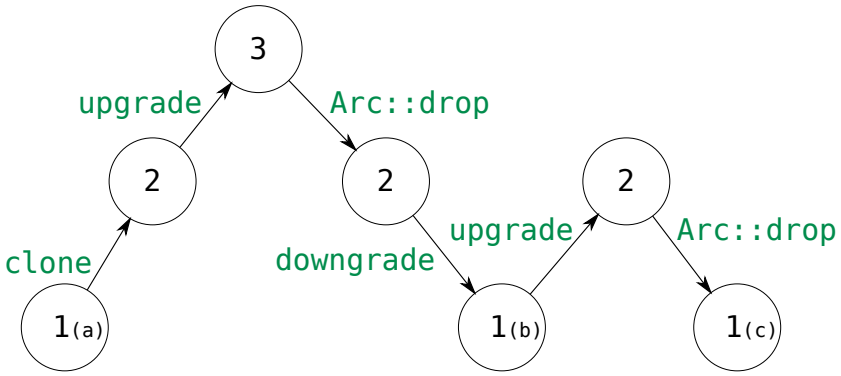


FIGURE 19.3: A truncated history of the Arc counter

type &mut T.[22] No matter if the second check fails or not, is_unique will release the lock on the Weak counter with a release write of value 1.

[22]The Arc::make_mut function also follows the similar pattern, but the targets are reversed: it first acquires a "lock" on the Arc counter and then reads the Weak counter.

CORRECTNESS OF is_unique. We wish to verify is_unique against the following abstract specification.

$$\left\{\mathsf{ARC}_q^{\gamma_i,\gamma}(\mathtt{a},\mathcal{I}) * \Phi(q)\right\}$$
$$\quad \texttt{is\_unique}(\mathtt{a})$$
$$\left\{\begin{array}{l} b.\ \textbf{if}\ b\ \textbf{then}\ \mathtt{a.strong} \mapsto 1 * \mathtt{a.weak} \mapsto 1 * \Phi(1) \\ \quad \textbf{else}\ \mathsf{ARC}_q^{\gamma_i,\gamma}(\mathtt{a},\mathcal{I}) * \Phi(q) \end{array}\right\} \quad \text{(ARC-IsUnique-spec)}$$

where Φ is a predicate on fractions that may be instantiated with the interpretation of the underlying type T. The specification says that if the function returns **true**, then we should have access to the full fraction of $\Phi(1)$. Therefore, if we have instantiated Φ as the ownership of T, we should have full access to T (*e.g.*, &mut T) in that case.

The key idea to the correctness of is_unique is that its two checks ensure the synchronization guarantee for temporary reclamation. The second check ensures that the thread is synchronized with all *other* Arc::drop calls. This means that it is synchronized with all accesses to the content made by all other Arc pointers. The thread, of course, must have synchronized with all accesses made by the current Arc pointer that it owns. Consequently, the thread must have synchronized with all accesses to the underlying content.

The problem, however, is that the second check uses an acquire *read*, instead of a **CAS**. If it were a **CAS**, then we would be guaranteed to read the latest value of the Arc counter, and thus synchronizing with all other Arc::drop's. However, an acquire read does not guarantee reading the latest value: it can read a stale one. Consider a truncated history of the Arc counter in Figure 19.3, where our call to get_mut was initiated somewhere before the latest write 1(c) to the counter. Since we do not know exactly when get_mut was initiated, the second check by is_unique may read 1 from any events 1(a), 1(b) or 1(c). Had it read from 1(a), we would not have synchronized with the Arc::drop's or downgrade's after that. Our obligation here is to show that if the second check read 1, it must have read from 1(c).

By contradiction, we show that it is impossible to read 1 from 1(a), 1(b) or any stale 1 values. Put it another way, we show that the thread

has observed all updates to the Arc counter from a stale 1 to 2, denoted as stale$(1 \rightsquigarrow 2)$, and therefore cannot read those stale 1's again. This is where the first check comes into play: it gives us the guarantee that the thread has observed all stale$(1 \rightsquigarrow 2)$ updates. Note that these updates come either from an Arc::clone or from a Weak::upgrade. If the update is from an Arc::clone, like in 1(a), the thread must have observed it because that update must have been performed by some Arc pointer—unique at that time—of which the current Arc pointer (which this thread owns) is a *descendant*.

The remaining case is when the update is from a Weak::upgrade, like in 1(b). By the first check the thread is synchronized with all Weak::drop's by *all* Weak pointers. Note that Weak::drop, similar to Core Arc::drop,[23] does a release **FAA** to decrement the Weak counter. However, unlike in Core Arc, the last Weak::drop decrements the counter to 1 (instead of 0). Therefore, when the first check did a successful acquire **CAS** for value 1 on the Weak counter, it knows that there is no Weak pointers left and it is synchronized with all Weak::drop's.

[23] see Figure 19.1

If an update stale$(1 \rightsquigarrow 2)$ is from an Weak::upgrade, it must happen-before the Weak::drop of the same Weak pointer. Thus, by synchronizing with all Weak::drop's, the thread is guaranteed to synchronize with all stale$(1 \rightsquigarrow 2)$ updates from Weak::upgrade's. It follows that the thread must have read the latest write to the Arc counter.

ANOTHER INSTANCE OF SYNCHRONIZED GHOST STATE. Thus, our challenge here pins down to formalizing the observations of stale$(1 \rightsquigarrow 2)$ and the two sources of those observations. Furthermore, the observations are tied to the ownership of some Arc or Weak pointer, and when such ownership is transferred the observations must also be transferred in a *synchronized* way.

For this purpose, we use an instance of synchronized ghost state for those observations. Similar to the ghost state for RMC cancelable invariants,[24] we use the ghost state of form $\boxed{\circ (q, O)}^{\gamma}$ where O is a set of observations. In the particular case of Arc, an observation is simply a unique identifier id for each stale$(1 \rightsquigarrow 2)$ update event on the Arc counter. We therefore rely on the iRC11 logic to provide unique identifiers for update events. Fortunately, we can simply use the *timestamp* of an update/write event as the identifier. This is indeed a motivation to expose timestamps in the logic, and subsequently in GPS protocol assertions. We thus can tie the logical ghost state to the write events, making the observations actually *physical* and so can only be transferred with physical synchronization.

[24] see §11.2.2

In the verification of Arc, we use two different constructions: one, $\boxed{\circ (q, O_u)}^{\mu}$, to track the observations coming from Weak::upgrade, and another, $\boxed{\circ (q, O_c)}^{\gamma}$, to track those coming from Arc::clone. The former construction $\boxed{\circ (q, O_u)}^{\mu}$ enjoys similar properties to that of RMC cancelable invariants. That is, the observations can be joined (using set union), and if we own the full fraction $\boxed{\circ (1, O_u)}^{\mu}$, then we are guaranteed that O_u contains all possible Weak::upgrade's stale$(1 \rightsquigarrow 2)$ events and we have *physically* seen them all. Additionally, each owner of each fraction q

can concurrently add observations to its local set O. This is to reflect the fact that any `Weak` pointer can always perform a stale$(1 \rightsquigarrow 2)$ event.

The latter construction $\boxed{\circ(q, O_c)}^{\gamma}$ is a bit different. Even if we only own a fraction $\boxed{\circ(q, O_c)}^{\gamma}$, we need to know that O_c contains all possible `Arc::clone`'s stale$(1 \rightsquigarrow 2)$ events and we have *physically* seen all of them. Furthermore, we can only add observations to O_c if we have the full fraction $\boxed{\circ(1, O_c)}^{\gamma}$. This reflects the fact that any `Arc` pointer must have seen all `Arc::clone`'s $1 \rightsquigarrow 2$ updates, and that any `Arc::clone`'s $1 \rightsquigarrow 2$ update can only be done by the one `Arc` pointer that was unique and should own the full fraction at the time of the update.

We then set up that the abstract predicate ARC for ownership of `Arc` pointers also contains a fraction $\boxed{\circ(q, O_c)}^{\gamma}$ for some q[25] and O_c (because only `Arc` pointers can do `Arc::clone`), and that the abstract predicate WEAK for ownership of `Weak` pointers contains a fraction $\boxed{\circ(q, O_u)}^{\mu}$ for some q and O_u (because only `Weak` pointers can do `Weak::upgrade`). We further require that `Arc::drop` also releases the fraction $\boxed{\circ(q, O_c)}^{\gamma}$ like releasing the other fractions, and similarly that `Weak::drop` releases $\boxed{\circ(q, O_u)}^{\mu}$.

[25] the same q in *e.g.*, $\heartsuit_q^{\gamma_i}$ and `a.data` $\overset{q}{\mapsto} v$—see Definition 19.2

With that setup, we are ready to show that when the two checks of `is_unique` succeed, the thread must have observed all stale$(1 \rightsquigarrow 2)$ updates. First, when acquiring the "lock" on the `Weak` counter, the thread also acquire the full fraction $\boxed{\circ(1, O_1)}^{\mu}$ from the `Weak` counter protocol. The full fraction is available in the protocol because all `Weak` pointers have been `drop`ped. With $\boxed{\circ(1, O_1)}^{\mu}$, the thread is guaranteed to have seen all `Weak::upgrade`'s stale$(1 \rightsquigarrow 2)$ updates. Second, since the thread owns an `Arc` pointer, it owns a fraction $\boxed{\circ(q, O_2)}^{\gamma}$, which guarantees that the thread has seen all `Arc::clone`'s stale$(1 \rightsquigarrow 2)$ updates. Consequently, the thread must have read 1 from the latest write to the `Arc` counter, and thus is synchronized with all previous accesses to the underlying content `T`. □

19.4 Insufficient Synchronization in `get_mut`

Unfortunately, our setup was not strong enough to verify `Arc` and `Weak` without change. The two reads of the counters in the second check of `get_mut` and `make_mut` were **`rlx`** in the original code (line 4, Table 19.2), and we had to strengthen them both to **`acq`** in order to make the verification go through. The reason is that, while we managed to temporarily get the full resources out by a read, the **`rlx`** reads do not give us those resources in the current view: they are under an acquire modality ∇, and we could not move them out of the modality, and got stuck in the proof.

While we conjecture that a **`rlx`** read in `make_mut` is in fact sufficient, a **`rlx`** read in `get_mut` turned out to be insufficient and we have reported the bug and the fix has been merged into Rust codebase. The following example invokes a data race when using `get_mut`:

```
1  let mut arc1 = Arc::new(0);
2  let     arc2 = Arc::clone(&arc1);
3  thread::spawn(move || {
```

```
4     let _ : u32 = *arc2; /* drop(arc2); */
5   });
6   loop {
7     match Arc::get_mut(&mut arc1) {
8       None => {}
9       Some(m) => { *m = 1u32; return; }
10    }
11  }
```

In this example there are two non-atomic operations: the read of the underlying integer in line 4 (child thread) and the write to the same integer in line 9 (parent thread). The read should be safe because the child thread owns `arc2`, thus the underlying integer is shared and *immutable*. The write should be safe because `get_mut` guarantees that the parent thread owns the unique `Arc` pointer (`arc1`) and should temporarily gain full access to the non-atomic integer. This can only happens after the child thread finishes and `arc2` has been dropped. However, the two non-atomic operations constitute a data race by C11 standard, because neither one happens-before the other. More specifically, in line 4 of the child thread, when `arc2` goes out of scope, it will be destructed by `Arc::drop`, which uses a release (**`rel`**) RMW (see the code at line 16, Table 19.2). This release RMW will be read by `get_mut` (line 4, Table 19.2) in the parent thread (line 7). If this read had been **`acq`**, then there would have been a release-acquire synchronization between the release RMW of `drop` and the acquire read of `get_mut`, and the non-atomic read of the child thread would have been guaranteed to happen-before the non-atomic write of the parent thread. However, the read was **`rlx`**, thus no happen-before relationship can be established between the two non-atomic operations.

CHAPTER SUMMARY. In this chapter, we sketched the verification of $\text{RB}_{\texttt{rlx}}$ most substantial example: `Arc<T>` and `Weak<T>` libraries from Rust. We followed the original RustBelt work to verified the libraries against a sufficiently strong abstract specification, which is then used to perform the semantic type checking of those libraries. The semantic type checking proofs are reused as-is from the original RustBelt work, and we have only discussed the RMC verification of the libraries against the abstract specification. We reported a synchronization bug in `Arc::get_mut`.

20

Related Work

Doko and Vafeiadis[1] verify a subset of the `Arc` library with FSL++. We improve on their results by (1) enlarging the scope of the verification to include important parts of the API such as the `make_mut` and `get_mut` functions (the latter of which we found to contain a data race) as well as the `Weak` reference type, (2) allowing full resource reclamation of both the contents and the reference-count fields of the `Arc` data structure, and (3) embedding our verification effort in the RustBelt framework, so that we can establish the soundness of `Arc` when linked with unknown well-typed λ_{Rust} code.

[1] Doko and Vafeiadis, "Tackling Real-Life Relaxed Concurrency with FSL++" [DV17].

Cancelable single-writer protocols given in §17.1 are more general and support stronger reclamation schemes than any variants from previous RMC logics, such as FSL/FSL++, GPS, and iGPS. For example, previous logics would only be able to verify `RwLock`'s writer-lock release function where the lock is released with a **`CAS`**—but not with a *release write* as in the Rust's implementation. More importantly, previous variants were not designed to reclaim resources in the fashion of Rust's lifetimes.

Part IV

COMPASS

Part IV presents the Compass specification framework. Chapter 21 starts by reviewing specifications with logical atomicity in both SC and RMC settings. Chapter 22 discusses how to encode Yacovet specs in iRC11 with logical atomicity. Chapter 23 present the proofs of RMC stacks and queues against the Compass specs, relying on general *multi-location invariants* (Chapter 11) and atomic points-to (Chapter 10). Chapter 24 discusses the composition of the stack spec and the exchanger spec to verify the elimination stack. In doing so, it discusses the role of *helping* with logical atomicity in the specs of exchangers. The top of Figure 1.1 presents the dependency between these chapters and with previous chapters.

21

Background: Strong Specifications with Logical Atomicity

Strong memory models provide strong guarantees about the ordering of memory operations, making it easier to write clearly correct library implementations. More relaxed memory consistency models offer more opportunities for more efficient implementations, which, on the other hand, may provide weaker guarantees to clients. In this chapter, using the Queue data structure as an example, we review existing *logically atomic* specifications (from now on, *specs* for short) in stronger memory models (Figure 21.1), and in Chapter 22 we will present several of our logically atomic specs in the RC11 model. We review, in §21.1, the traditional Hoare-triple-based specs for sequential queues; in §21.2, *logical atomicity*[1] and its uses to give strong specs for concurrent SC queues; and in §21.3, how Cosmo[2] extends those specs for RMC with *thread views*.

[1] Rocha Pinto et al., "TaDA: A Logic for Time and Data Abstraction" [RPDG14]; Svendsen and Birkedal, "Impredicative Concurrent Abstract Predicates" [SB14]; Jung et al., "The future is ours: prophecy variables in separation logic" [Jun+20].

[2] Mével and Jourdan, "Formal verification of a concurrent bounded queue in a weak memory model" [MJ21].

21.1 Sequential Specifications for Queues

The separation logic sequential specs for queues are given as SEQ-ENQ and SEQ-DEQ (Figure 21.1). SEQ-ENQ specifies that an enqueue function call **enq**$([q, v])$ can run safely as long as it has Queue(q, vs), an abstract separation logic assertion that represents full ownership of the queue object q (an instance of the data structure). An implementation can define Queue(q, vs) as arbitrary resources that it specifically needs. But from the perspective of clients, Queue(q, vs) is abstract because it asserts that q's current state can be seen *abstractly* as a list of values vs—that is, the queue's elements are currently vs, ordered by the list order.

SEQ-ENQ then says that **enq**$([q, v])$ requires and consumes q's ownership at the beginning of the call, and at the end of the call it returns the ownership with the updated abstract state $vs +\!\!+ [v]$, reflecting the operation's effects: v has been enqueued to the end of q. Conversely, by SEQ-DEQ, a dequeue **deq**$([q])$ also consumes q's ownership and, if the queue is not empty, returns the head value v of vs and gives back the ownership with only its tail vs'. (The notation $\{v.\, Q\}$ denotes the post-condition with a returned value v.) Otherwise, if q is empty, **deq**$([q])$ returns empty (ϵ) and the fact that the abstract state is also empty ($[]$).

That an operation is allowed to consume the queue ownership for

the *whole* duration of its execution is what makes the specs *sequential*: a group of threads cannot access the ownership $\mathsf{Queue}(q, vs)$ concurrently in order to perform concurrent enqueues and/or dequeues. To have strong specs for such *fine-grained concurrency*, we need logical atomicity.

21.2 SC Specifications with Logical Atomicity

In fine-grained concurrency, a concurrent object's ownership is shared for concurrent accesses, and contention is most commonly resolved by *atomic* read-modify-write (RMW) instructions, such as compare-and-swap (**CAS**). In this case, even if a concurrent object's operation may not be atomic (because it is implemented with multiple instructions), its effects are published by a single atomic instruction. This is the intuition of *logical atomicity*: from the perspective of clients, the operation *appears* to be atomically updating the object exactly around a single atomic instruction—often called the *commit* or *linearization* point of the operation.

As such, a client only needs to provide ownership of the concurrent object at the operation's commit point, and can expect the update to happen right after the commit point. This idea is encoded in *logically atomic triples* (LATs),[3] of the form $\langle P \rangle\, e\, \langle Q \rangle$, with angle brackets $\langle\,\rangle$ instead of curly braces. The intuitive interpretation is also a bit more subtle than normal Hoare triples: $\langle P \rangle\, e\, \langle Q \rangle$ means that there exists a commit point (instruction) c by which e *atomically* consumes P, transforms it, and returns Q.

[3] Rocha Pinto et al., "TaDA: A Logic for Time and Data Abstraction" [RPDG14]; Svendsen and Birkedal, "Impredicative Concurrent Abstract Predicates" [SB14]; Jung et al., "Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning" [Jun+15]; Jung et al., "The future is ours: prophecy variables in separation logic" [Jun+20].

Using LATs, we can give strong specs like SC-ENQ and SC-DEQ (Figure 21.1) to fine-grained concurrent SC queues. Here we use red font-face to denote the gradual changes in the specs. One obvious change is the aforementioned angle brackets $\langle\,\rangle$. Less obvious is the quantification of vs in the precondition $\langle vs.\, \mathsf{Queue}(q, vs)\rangle$: this is a special form of universal quantification that signifies the possibility that the queue may be modified concurrently. Specifically, it signifies that during the specified enqueue/dequeue operation, other threads may be changing the state vs of the queue *arbitrarily*, up until the commit point of the operation, when it atomically updates the state to what is described in the post-condition. For example, SC-ENQ says that **enq**$([q, v])$ can withstand arbitrary concurrent updates to the state vs of q, up until the commit point when it atomically transforms $\mathsf{Queue}(q, vs)$ (where vs is the state at that instant) to the new state $\mathsf{Queue}(q, vs +\!\!+ [v])$. In contrast, the sequential spec SEQ-ENQ implicitly quantifies over vs with a normal universal quantifier $(\forall vs)$ at the outside: this allows the implementation to assume exclusive ownership of $\mathsf{Queue}(q, vs)$ for an arbitrary but *unchanging* vs, thereby prohibiting concurrent interference.

Last but not least, we add a *local* precondition $\mathsf{isQueue}(q)$, another abstract assertion that encodes persistent separation logic facts about the queue, *e.g.*, facts about its head and tail pointers. Thus, they are freely duplicable, and they are local in the sense that they are to be provided at the beginning of a call, so that operations can use them for the whole execution, more conveniently than $\mathsf{Queue}(q, vs)$ which is

SEQ-ENQ
$\{\mathsf{Queue}(q, vs)\}\ \mathbf{enq}([q, v])\ \{().\ \mathsf{Queue}(q, vs +\!\!+ [v])\}$

SEQ-DEQ
$\{\mathsf{Queue}(q, vs)\}\ \mathbf{deq}([q])\ \{v.\ (vs = [] * v = \epsilon * \mathsf{Queue}(q, [])) \lor (\exists vs'.\ vs = v :: vs' * \mathsf{Queue}(q, vs'))\}$

SC-ENQ
$\mathsf{isQueue}(q) \vdash \langle vs.\ \mathsf{Queue}(q, vs)\rangle\ \mathbf{enq}([q, v])\ \langle().\ \mathsf{Queue}(q, vs +\!\!+ [v])\rangle$

SC-DEQ
$\mathsf{isQueue}(q) \vdash \langle vs.\ \mathsf{Queue}(q, vs)\rangle\ \mathbf{deq}([q])\ \langle v.\ (vs = [] * v = \epsilon * \mathsf{Queue}(q, [])) \lor (\exists vs'.\ vs = v :: vs' * \mathsf{Queue}(q, vs'))\rangle$

ABS-SO-ENQ
$\mathsf{isQueue}(q) * \sqsupseteq V \vdash \langle vs.\ \mathsf{Queue}(q, vs)\rangle\ \mathbf{enq}([q, v])\ \langle().\ \mathsf{Queue}(q, vs +\!\!+ [(v, V)])\rangle$

ABS-SO-DEQ
$\mathsf{isQueue}(q) * \sqsupseteq V \vdash$
$\langle vs.\ \mathsf{Queue}(q, vs)\rangle\ \mathbf{deq}([q])\ \langle v.\ (v = \epsilon * \mathsf{Queue}(q, vs)) \lor (\exists vs', V'.\ vs = (v, V') :: vs' * \mathsf{Queue}(q, vs') * \sqsupseteq V')\rangle$

FIGURE 21.1: Specifications of Queue operations, from sequential, to SC concurrency and strong RMC

neither duplicable nor local.

Intuitively, it should be clear that $\langle P\rangle\, e\, \langle Q\rangle_{\mathcal{E}}$ is a stronger spec than $\{P\}\, e\, \{Q\}_{\mathcal{E}}$, seeing as the former permits concurrent interference whereas the latter does not. Intuitively, if e only needs P and Q around its commit point c, then it can also work with having P and Q around the whole execution, which includes c. But how does a client actually make use of these LATs to arbitrate concurrent accesses to a shared resource like $\mathsf{Queue}(q, vs)$? To that end, we need to see the interaction between LATs and *invariants*, which can formally explain how LATs are strictly stronger than normal Hoare triples.

LOGICAL ATOMICITY AND INVARIANTS. We recall the standard access rule for invariants HOARE-INV given in Figure 6.6: a physically atomic instruction e can access and rely on I, in addition to P, for its execution, as long as it gives I back right afterwards. The LAT invariant access rule LAT-INV *strengthens* HOARE-INV, as it relaxes the restriction of "accessing around atomic instructions" (atomic(e)) to "accessing around *logically* atomic expressions".

LAT-INV
$$\frac{\langle \triangleright I * P\rangle\, e\, \langle \triangleright I * Q\rangle_{\mathcal{E}\setminus\mathcal{N}} \qquad \mathcal{N} \subseteq \mathcal{E}}{\boxed{I}^{\mathcal{N}} \vdash \langle P\rangle\, e\, \langle Q\rangle_{\mathcal{E}}}$$

With this rule, clients can build protocols to use and combine libraries with LAT specs. For example, we can allocate an invariant

$$\boxed{\exists vs_1, vs_2.\ \mathsf{Queue}(q_1, vs_1) * \mathsf{Queue}(q_2, vs_2) * R(vs_1, vs_2)}^{\mathcal{N}}$$

that ties together two queues by a relation R, and then use LAT-INV with SC-ENQ and SC-DEQ to verify clients that use the two queues and adhere to the "protocol" R. For example, R may require that vs_1 and vs_2 are disjoint, or even more specifically, that one queue only contains odd numbers and the other only contains even numbers.

In summary, with logical atomicity and invariants, one can give stronger modular specs for fine-grained concurrent libraries. Furthermore, LAT specs can be seen as giving *abstract operational semantics* to a library's operations. As such, the library should be linearizable, *i.e.*, there is a total order of its operations according to which the concurrent object appears to behave sequentially. In fact, Birkedal et al.[4] recently showed formally that, in SC, logical atomicity implies linearizability. It is therefore an important tool to achieve full functional correctness *and* modular client reasoning.

[4] Birkedal et al., "Theorems for free from separation logic specifications" [Bir+21].

21.3 Logically Atomic Specifications in RMC with Views

However, linearizability and logical atomicity do not directly extend to relaxed memory. In RMC, a total order of operations (the linearization) might not exist, or if it does exist, it may not be very useful. In contrast to the SC model where every atomic instruction is *synchronized with* each other instruction, in RMC an atomic instruction may only be synchronized with *some* other instructions. It is the partially-ordered synchronizations—formally defined as the *happens-before* (hb) relation—between operations that really matter for their correctness, not the total order. In the terms of logical atomicity, this means that an update to the state by the commit of an operation o may only be meaningful to operations that are synchronized with o. Consequently, LAT specs for RMC libraries have to additionally account for hb.

21.3.1 *Cosmo Specs for Queues*

Abs-So-Enq and Abs-So-Deq (in Figure 21.1) are a simplified version of Cosmo specs for multi-producer multi-consumer queues. They differ from the SC specs in the extra tracking of views (in red in Figure 21.1): (1) the specs take the "seen view" assertion $\sqsupseteq V$ as a *local precondition* (that is, outside of the LAT precondition and needed at the beginning of the call); and (2) the abstract state is no longer just a list of values, but a list of value-view pairs, where the view component of a pair is the view of the enqueue operation (after its commit point). Similar to the release-acquire rules, the views in the abstract state support view transfers between matching enqueue-dequeue pairs: by Abs-So-Enq, an enqueue releases its local view V at its commit point, and by Abs-So-Deq, the matching dequeue acquires V into its local view, also at its respective commit point. Effectively, they expose the so relation between matching enqueue-dequeue pairs via views in the abstract state. This is why we call them $\texttt{LAT}^{\text{abs}}_{\text{so}}$ style. (The complete Cosmo specs also track so among enqueues and among dequeues.)

21.3.2 *Abstract State and Read-Only Operations*

However, by using just the abstract state, the specs do not specify behaviors of *read-only* operations that do not modify the abstract state. For example, in Abs-So-Deq, a failing empty dequeue is a read-only operation, and the $\texttt{LAT}^{\text{abs}}_{\text{so}}$ specs do not give us any new facts about vs. This is

```
enq([q, 41]);  ||          ||  repeat (*acq flag != 0);
enq([q, 42]);  ||  deq([q]) ||  deq([q])
flag :=rel 1   ||          ||  // return 41 or 42, not empty
```

FIGURE 21.2: A Message-Passing (MP) client with Queues

weaker than in the SC model, where SC-DEQ says that dequeues fail with ϵ only if the state vs is truly empty (at the commit point).

Realistically, an RMC spec cannot be quite as strong as the SC spec: recall that in RMC effects can appear to threads differently, so it may be that the thread π sees the queue as empty and returns ϵ, but the queue is in fact not empty, because a fresh enqueue by another thread ρ has not become visible to π yet. But we can do better than the empty case of ABS-SO-DEQ, which gives the client no useful information.

More concretely, the Cosmo spec's ABS-SO-DEQ cannot be used to verify the Message-Passing client with queues, with the expected behavior given in Figure 21.2. Here, the queue is accessed concurrently by 3 threads: the left-most thread performs 2 enqueues (**enq**), the middle one performs a dequeue (**deq**), and the right-most thread waits for the signal by the left-most thread through flag and then performs a dequeue. A weak implementation of dequeue can return *empty* even though the queue is not empty, due to contention. However, in this example, the right-most thread *cannot* get an empty dequeue result, because (1) at most one enqueue could have been consumed concurrently by the middle thread, and (2) due to the release-acquire synchronization through flag, the thread has synchronized with the two enqueues.

Unfortunately, the Cosmo spec only exposes *internal* (to the implementation) synchronizations among operations, without taking into account how additional *external* synchronizations created by the client (such as the synchronization through flag) can affect the behaviors of dequeues. It therefore cannot exclude the possibility that the right-most thread's dequeue returns empty.

In the next Chapter 22, we present specs that expose more of the hb relation, enough to cover read-only operations such as failing dequeues. Using those specs, we can verify the MP client in Figure 21.2: by combining the queue's richer hb relation with the client's *external* hb relation, we prove that the right-most thread's dequeue cannot return empty.

22

Strong Compass Specifications with Richer Partial Orders

In this chapter, we present several of our logically atomic specs that, by exposing richer partial orders that can be combined with *external synchronizations*, can stay reasonably strong and yet still satisfiable by more relaxed implementations in the weaker ORC11 memory model. In §22.1 we present the $\texttt{LAT}^{\text{abs}}_{\text{hb}}$ style which generalizes the $\texttt{LAT}^{\text{abs}}_{\text{so}}$ style, and its instance for queues, which suffices to verify the MP client in Figure 21.2. In §22.2, we present the $\texttt{LAT}_{\text{hb}}$ spec style, a weakening of $\texttt{LAT}^{\text{abs}}_{\text{hb}}$. In §22.3, we show how to encode these logically atomic specs in iRC11.

22.1 Graph-Based Specs to Encode Partial Orders

The $\texttt{LAT}^{\text{abs}}_{\text{hb}}$ style extends the $\texttt{LAT}^{\text{abs}}_{\text{so}}$ style by exposing a greater part of hb. An instance for queues is given in ABS-HB-ENQ, ABS-HB-DEQ, and ABS-HB-QUEUE-CONSISTENCY (Figure 22.1). That these specs are stronger than those of Cosmo can be seen easily by ignoring the added red parts. The main improvement of this instance is in ABS-HB-DEQ's failure case, where the caller sees the queue as empty. Here, the spec provides more information about how the resulting read-only *empty dequeue* operation is ordered with other operations in hb.

As read-only operations have no effects on the abstract state, we need a new component G to identify and relate them to other operations. The component $G \in \mathit{Graph}$ is a general construction inspired by the declarative specs of Yacovet.[1] Yacovet works on *whole-program execution graphs*, and abstracts them into per-library *event graphs* of operations, where every operation is uniquely identified by an event. A Yacovet spec for a library encodes the ordering between events in a graph as partial orders that must satisfy some *library-specific consistency conditions*. Here, we encode Yacovet specs with the event graph component G. The main differences with Yacovet are that (1) G records only the library events that have happened so far, not complete executions; and (2) our specs are stated as separation logic LATs, so each operation can access the current, up-to-date event graph G and only needs to extend G with the operation's event and to maintain the graph's consistency.

The (simplified) types of event graphs are given at the top of Fig-

[1] Raad et al., "On library correctness under weak memory consistency: specifying and verifying concurrent libraries under declarative consistency models" [Raa+19].

$$
\begin{aligned}
V \in \mathit{View} &::= \mathit{Loc} \to \mathit{Time} \\
e \in \mathit{EventId} &::= \mathbb{N} \\
\mathit{QueueEvent} &::= \mathsf{Enq}(v) \mid \mathsf{Deq}(v) \mid \mathsf{Deq}(\epsilon) \\
M \in \mathit{LogView} &::= \wp(\mathit{EventId}) \\
\mathit{Event} &::= \mathit{QueueEvent} \times \mathit{View} \times \mathit{LogView} \\
G \in \mathit{Graph} &::= (\mathit{EventId} \to \mathit{Event}, \wp(\mathit{EventId} \times \mathit{EventId}))
\end{aligned}
$$

$$
\mathsf{QueueConsistent}(vs, G) := \wedge \begin{cases}
\forall (e,d) \in G.\mathsf{so}.\, \exists v.\, G(e).\mathsf{type} = \mathsf{Enq}(v) \wedge G(d).\mathsf{type} = \mathsf{Deq}(v) \wedge \ldots & \text{(Queue-Matches)} \\
\forall (e,d) \in G.\mathsf{so}, e'.\, G(e').\mathsf{type} = \mathsf{Enq}(\_) \to (e',e) \in G.\mathsf{lhb} \to & \\
\quad \exists d'.\, (e',d') \in G.\mathsf{so} \wedge (d,d') \notin G.\mathsf{lhb} & \text{(Queue-FIFO)} \\
\forall d, e.\, G(d).\mathsf{type} = \mathsf{Deq}(\epsilon) \to G(e).\mathsf{type} = \mathsf{Enq}(\_) \to & \\
\quad (e,\_) \notin G.\mathsf{so} \to (e,d) \notin G.\mathsf{lhb} & \text{(Queue-EmpDeq)} \\
\ldots &
\end{cases}
$$

Abs-Hb-Queue-Consistency
$$\mathsf{Queue}(q, vs, G) \vdash \mathsf{QueueConsistent}(vs, G)$$

Abs-Hb-Enq
$$
\begin{array}{l}
\mathsf{SeenQueue}(q, G_0, M_0) * \sqsupseteq V \vdash \\
\langle G, vs.\, \mathsf{Queue}(q, vs, G) \rangle \\
\quad \mathbf{enq}([q, v]) \\
\left\langle (). \exists G' \sqsupseteq G, M' \supseteq M_0, V' \sqsupseteq V.\, \mathsf{Queue}(q, vs +\!\!+ [(v, V')], G') \right. \\
\qquad \left. * \mathsf{SeenQueue}(q, G', M') * \sqsupseteq V' * \exists e \notin G.\, e \in M' \wedge G' = G[e \mapsto (\mathsf{Enq}(v), V', M')] \right\rangle
\end{array}
$$

Abs-Hb-Deq
$$
\begin{array}{l}
\mathsf{SeenQueue}(q, G_0, M_0) * \sqsupseteq V \vdash \\
\langle G, vs.\, \mathsf{Queue}(q, vs, G) \rangle \\
\quad \mathbf{deq}([q]) \\
\left\langle v. \exists vs', G' \sqsupseteq G, M' \supseteq M_0, V' \sqsupseteq V.\, \mathsf{Queue}(q, vs', G') * \mathsf{SeenQueue}(q, G', M') * \sqsupseteq V' \right. \\
\qquad * \vee \begin{cases}
(v = \epsilon \wedge vs' = vs \wedge \exists d \notin G.\, d \in M' \wedge G' = G[d \mapsto (\mathsf{Deq}(\epsilon), V', M')]) \\
(\exists V_e.\, vs = (v, V_e) :: vs' \wedge \exists e, M_e, d \notin G.\, G(e) = (\mathsf{Enq}(v), V_e, M_e) \wedge (e,\_) \notin G.\mathsf{so} \wedge V_e \sqsubseteq V' \\
\qquad \wedge M_e \cup \{e, d\} \subseteq M' \wedge G' = G[d \mapsto (\mathsf{Deq}(v), V', M')] \wedge G'.\mathsf{so} = \{(e,d)\} \cup G.\mathsf{so})
\end{cases} \\
\left. \right\rangle
\end{array}
$$

Figure 22.1: Compass Specs for Queues

ure 22.1. A graph G is a pair of (1) a function that maps each event id $e \in \mathit{EventId}$ to event data of type *Event*, and (2) a set of event id pairs that encodes the so relation. We use $G(e)$ to denote the event data for e in G, and $G.\mathsf{so}$ to denote the so relation of G.

The type *Event* is a tuple of (1) an event type (`type`), (2) a *physical* view (`view`), and (3) a *logical* view (`logview`). In Figure 22.1 we give an instance of the event type for queues: the events can be an *enqueue* event of v ($\mathsf{Enq}(v)$), a *successful dequeue* event of v ($\mathsf{Deq}(v)$), or a *failing (empty) dequeue* event ($\mathsf{Deq}(\epsilon)$). An event's physical view is the view at the commit point of the operation that the event represents, and is needed in the logic to interact with other memory instructions. The event's logical view is also recorded at the commit point of its operation, and is a set of events for all library operations that *happen-before* the operation in question. If an event e is in the logical view of another event d, *i.e.*, $e \in G(d).\mathtt{logview}$, we say that e happens before d. Technically, it is the commit instruction of e's operation that happens before the commit instruction of d's operation.

Intuitively, we use the logical view construction as an approximation of the hb relation between library operations, just as the physical view construction is an approximation of hb between memory instructions. The difference is that while physical views approximate hb *globally* between memory instructions, logical views only approximate hb *locally* for the library in question. As such, our logical views correspond to the *local happens-before* lhb relation of a library object introduced by Yacovet. Henceforth we use $e \in G(d).\mathtt{logview}$ and $(e, d) \in G.\mathsf{lhb}$ interchangeably.

The $\mathtt{LAT}^{\mathsf{abs}}_{\mathsf{hb}}$ style extends $\mathtt{LAT}^{\mathsf{abs}}_{\mathsf{so}}$ following a simple pattern: (1) the abstract state is accompanied by the graph that tracks all operations committed so far, and (2) at each operation's commit point, in addition to a potential update of the abstract state, a fresh event e representing the operation is added to the graph. For example, in Abs-Hb-Enq, when an enqueue of v commits, the current graph G of q is extended atomically with a fresh event e whose type is $\mathsf{Enq}(v)$, into G': $G \sqsubseteq G'$.

Local assertions for logical views. The partial orders are also extended at e's commit point to relate it to other operations. In Abs-Hb-Enq, $G'.\mathsf{lhb}$ extends $G.\mathsf{lhb}$ by setting $G'(e).\mathtt{logview} = M'$, the set containing all operations that happen before e. M' includes M_0—the *local logical view* of the calling thread, which tracks the operations that happen-before the **enq** call. This tracking of thread-local logical views is done by a new *persistent* assertion $\mathsf{SeenQueue}(q, G_0, M_0)$, where G_0 is a *snapshot* of the current G ($G_0 \sqsubseteq G$), and together with M_0 they accumulate (a lower bound on) the information about operations that the thread has synchronized with. For instance, after the call, the thread receives $\mathsf{SeenQueue}(q, G', M')$ with the latest snapshot G' and a new logical view M', reflecting that the thread has synchronized with more operations ($M_0 \sqsubseteq M'$), including the operation e that it has just executed ($e \in M'$). By taking SeenQueue as a local pre-condition, the specs can specify that the operation's behavior can depend on what has happened before it—we will shortly see how that allows us to use Abs-Hb-Deq to verify the MP

client in Figure 21.2.

Compared to the $\texttt{LAT}^{\text{abs}}_{\text{so}}$ style, in $\texttt{LAT}^{\text{abs}}_{\text{hb}}$ each library type has a local logical view assertion like SeenQueue that plays a double role: (1) to track the thread-local logical view (as explained above) and also (2) to track persistent facts about the object like the isQueue(q) assertion in ABS-SO-ENQ. The logical view assertion plays the same role for logical views as the "seen view" assertion $\sqsupseteq V$ does for physical views: the tracked current local view can be published into the "public domain" (*i.e.*, the shared graph for logical views, the shared location history or abstract state for physical views) so that it can be consumed by other threads.

CONSISTENCY CONDITIONS. The $\texttt{LAT}^{\text{abs}}_{\text{hb}}$ style specifies properties of the abstract state and the partial orders through the library's *consistency conditions*. The consistency conditions are invariant, *i.e.*, should be maintained by all operations, and are specific to each library type.

For example, an excerpt of QueueConsistent, the consistency conditions for the queue library type, is given at the bottom right of Figure 22.1. It requires, among other things, that enqueues and dequeues must follow the first-in-first-out principle (FIFO, QUEUE-FIFO), stated in a fashion that is not too strong for RMC (more about that below). The fact that QueueConsistent is maintained by all operations is encoded in ABS-HB-QUEUE-CONSISTENCY: the queue ownership assertion Queue(q, vs, G), which is consumed and reproduced around the commit point, always implies consistency. So when ABS-HB-ENQ and ABS-HB-DEQ extend (vs, G) to new state (vs', G'), the operations can assume QueueConsistent(vs, G) and must then re-establish QueueConsistent(vs', G').

More specifically, if **deq** succeeds with a value v, ABS-HB-DEQ tells the client that $G'.\mathsf{so}$ extends $G.\mathsf{so}$ with a new pair (e, d) where d is the new successful event added by the dequeue operation and e is an existing enqueue event that d dequeues from. Therefore, through ABS-HB-QUEUE-CONSISTENCY, the spec additionally says that (e, d) satisfies, among other things,[2] (1) QUEUE-MATCHES: the return value v of the dequeue d must match the value enqueued by e; and (2) QUEUE-FIFO: if there is another enqueue event e' that happens before e, then e' must already have been dequeued by some d' ($(e', d') \in G.\mathsf{so}$), and our d cannot happen before d' ($(d, d') \notin G.\mathsf{lhb}$). (The consistency conditions on enqueue events are elided, so we will not discuss them.)

[2] For example, an element can only be dequeued once.

WEAKER BUT FLEXIBLE. The QUEUE-FIFO condition appears weaker than what one might expect, *i.e.*, $(d', d) \in G.\mathsf{lhb}$, but such a condition only works for strongly synchronized (*e.g.*, SC) implementations. As stated, QUEUE-FIFO is also satisfiable by implementations that have little synchronization between dequeues. In fact, we have verified that QUEUE-FIFO is satisfiable by a fairly relaxed implementation (similar to the weak version in [Raa+19]) of the Herlihy-Wing queue.[3] The implementation only ensures lhb between matching enqueue-dequeue pairs, but not among enqueues or among dequeues. (As one might guess, enqueues only use release operations, and dequeues only use acquire ones.)

[3] Herlihy and Wing, "Linearizability: A Correctness Condition for Concurrent Objects" [HW90].

Nonetheless, QUEUE-FIFO is still flexible enough that, for example,

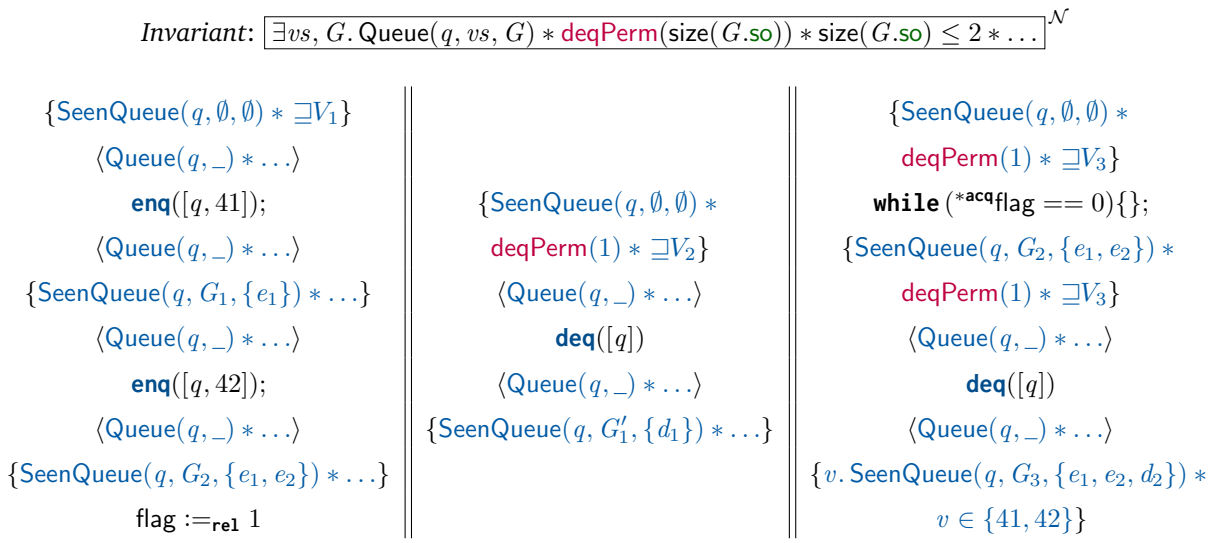


FIGURE 22.2: A proof sketch of Message Passing with queues

if a client decides to use the queue in an SC fashion by adding sufficient external synchronization, the client can know that lhb is total, *i.e.*, $(d', d) \in G.\mathsf{lhb} \vee (d, d') \in G.\mathsf{lhb}$, and can thus exclude the right-hand side of the disjunction and regain the stronger FIFO condition with $(d', d) \in G.\mathsf{lhb}$. This demonstrates the benefits of more detailed partial orders: by specifying ordering between operations with more complex but seemingly weaker conditions, we can (1) require only minimal ordering from implementations, and at the same time (2) allow clients the flexibility to strengthen the specs by combining the library's exposed internal ordering with the client-generated external ordering.

MESSAGE-PASSING CLIENT VERIFICATION When a call to **deq** returns empty (ϵ), consistency demands that the added empty dequeue event d satisfies QUEUE-EMPDEQ, which is sufficient to verify the MP client (Figure 21.2). Intuitively, QUEUE-EMPDEQ says that there cannot be another enqueue e which happens before d but has not been dequeued in G—if there were, then the dequeue would have successfully returned some element from the queue. The verification of MP depends on the fact that both enqueue events e_1 and e_2 done by the left-most thread, of which at most one can be consumed by the middle thread, happen before the dequeue of the right-most thread. By QUEUE-EMPDEQ the dequeue cannot be an empty one and must dequeue from e_1 or e_2 and return either 41 or 42.

The proof sketch of this example in Compass is given in Figure 22.2. Following the pattern mentioned at the end of §21.2, we put the ownership $\mathsf{Queue}(q, \_)$ in an invariant to enforce a concurrent protocol on the queue, using a *dequeue permission* called deqPerm that can be defined with Iris ghost state. One dequeue permission $\mathsf{deqPerm}(1)$ is needed to perform one successful dequeue. This requirement can be seen in the invariant: $\mathsf{deqPerm}(\mathsf{size}(G.\mathsf{so}))$ counts the number of successful dequeues, and a successful dequeue will extend $G.\mathsf{so}$ by 1, so anyone who successfully dequeues needs to put in a $\mathsf{deqPerm}(1)$ to re-establish the

invariant. For our particular example, we also implement deqPerm such that there are only *two* deqPerm(1)'s (*i.e.*, deqPerm(2)) in the whole system. We then give one permission to each consumer thread before they run. Initially the queue is set to be empty, and all threads are given a persistent observation SeenQueue$(q, \emptyset, \emptyset)$ of the initial empty state.

The verification of the left-most thread is straightforward: for each enqueue, we use LAT-INV to open the invariant and then use ABS-HB-ENQ. Afterwards the thread has two enqueue events $\{e_1, e_2\}$ in its logical view, and the write to flag releases SeenQueue$(q, G_1, \{e_1, e_2\})$ to the right-most thread. The verification of the middle thread uses LAT-INV and ABS-HB-DEQ, and if the dequeue succeeds, deqPerm(1) can be given up to re-establish the client invariant.

Finally, in the verification of the right-most thread, the acquire read of 1 from flag receives SeenQueue$(q, G_1, \{e_1, e_2\})$ from the left-most thread. We then use LAT-INV and ABS-HB-DEQ to perform the dequeue, with $M_0 := \{e_1, e_2\}$. Before re-establishing the invariant, we inspect the resulting dequeue d_3. If it is a successful dequeue, we can put deqPerm(1) in the invariant and finish. If d_3 is an empty dequeue, we derive a contradiction. As there are only two deqPerm(1) permissions in the whole system, of which one is owned by the current (right-most) thread, when we open the invariant we know that the most up-to-date (right before d_3) graph G can have at most one dequeue: size$(G.\mathsf{so}) \leq 1$. Furthermore, the thread has observed two enqueues, so in G there must be at least one enqueue that is not dequeued yet, which must be in $\{e_1, e_2\}$. Due to SeenQueue$(q, G_1, \{e_1, e_2\})$, both e_1 and e_2 happen before d_3. By QUEUE-EMPDEQ, we have our contradiction. □

22.2 Weaker Specs by Abandoning Abstract States

The $\texttt{LAT}^{\text{abs}}_{\mathsf{hb}}$ specs are particularly strong and only satisfiable by strong implementations, because one must be able to construct the abstract state at commit points. For example, we have verified that a purely release-acquire implementation of the Michael-Scott queue[4] satisfies the $\texttt{LAT}^{\text{abs}}_{\mathsf{hb}}$ specs for queues (and therefore transitively the $\texttt{LAT}^{\text{abs}}_{\mathsf{so}}$ specs). The release-acquire memory model, though not as strong as the SC or Multicore OCaml model, still provides sufficient synchronization to construct the list of values vs in the queue.

[4] Michael and Scott, "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms" [MS96].

However, it is extremely difficult to construct the abstract state for the relaxed Herlihy-Wing queue implementation mentioned above: it would require delicate reordering of commit points on the fly, and sometimes require *future-dependent* knowledge about dequeue operations. In fact, the verification of the LAT specs in the SC memory model for Herlihy-Wing queue relied on *prophecy variables*,[5] whose application in RMC is still an open research problem. In this work we instead verify the relaxed Herlihy-Wing implementation against $\texttt{LAT}_{\mathsf{hb}}$ specs, a weakening of the $\texttt{LAT}^{\text{abs}}_{\mathsf{hb}}$ specs where the abstract state is abandoned. In particular, our instance of the $\texttt{LAT}_{\mathsf{hb}}$ specs for queues is exactly the specs ABS-HB-ENQ and ABS-HB-DEQ (Figure 22.1) *without* vs.

[5] Jung et al., "The future is ours: prophecy variables in separation logic" [Jun+20].

$\texttt{LAT}_{\texttt{hb}}$ specs may appear weak, but they can still take advantage of external synchronization information, *i.e.*, the argument in §22.1 about flexibility of the partial orders still applies. Practically, they are sufficient to verify the MP client in Figure 21.2.

We can also use them to verify the following *single-producer single-consumer* (SPSC) client of a queue:

$$\begin{array}{c} \left\{\begin{array}{l}\mathsf{SeenQueue}(q,\_,\_)*\\ a_p \mapsto [a_0,\ldots,a_{n-1}]*\ldots\end{array}\right\} \\ \textbf{produce}(q, a_p, 0, n) \\ \{a_p \mapsto [a_0,\ldots,a_{n-1}]*\ldots\} \end{array} \Bigg\| \begin{array}{c} \left\{\begin{array}{l}\mathsf{SeenQueue}(q,\_,\_)*\\ a_c \mapsto [0,\ldots,0]*\ldots\end{array}\right\} \\ \textbf{consume}(q, a_c, 0, n) \\ \{a_c \mapsto [a_0,\ldots,a_{n-1}]*\ldots\} \end{array}$$

Here, there is only one thread performing enqueues—the **produce**r—and only one thread performing dequeues—the **consume**r. The **produce**r reads the array a_p for elements with the indices in $[0, n)$ and enqueues them in that order, while the **consume**r keeps dequeueing for n elements and writes them in the indices $[0, n)$ of the array a_c in the dequeueing order. The expected behavior is FIFO: in the end the array a_c should have the same elements as a_p.

To verify this example, we use the $\texttt{LAT}_{\texttt{hb}}$ specs for queues (*i.e.*, Abs-Hb-Enq and Abs-Hb-Deq *without* abstract states) to derive *stronger* $\texttt{LAT}_{\texttt{hb}}$-style specs for SPSC queues, simply by building a concurrent SPSC client protocol. In this derivation, thanks to logical atomicity, at every commit point of a successful dequeue we can easily match it up with the right enqueue and thus prove FIFO. With the SPSC $\texttt{LAT}_{\texttt{hb}}$ specs, the example's verification is straightforward.

22.3 Implementing Compass Specs in iRC11

Surprisingly, it is rather straightforward to implement logical atomicity specs in iRC11: we reuse the general definition of *atomic update* that Iris provides (see Definition 22.1), and RMC effects can be fully accounted for using the various view modalities introduced in §8.5.

Definition 22.1 (General Iris Atomic Update)**.** $\boxed{\langle x.\ P \mid y.\ Q \Rrightarrow \Phi\rangle^{\mathcal{E}}_{\mathcal{E}'}}$

iRC11 reuses the general logical atomicity setup provided by Iris for all bunched-implication logics. The key concept in this setup is the atomic update which encodes the obligations to update some state around an atomic commit point.

$$\langle x.\ P \mid y.\ Q \Rrightarrow \Phi\rangle^{\mathcal{E}}_{\mathcal{E}'} ::= {}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}'} \exists x.\ P * \wedge \begin{cases} P \ {}^{\mathcal{E}'}\!\Rrightarrow^{\mathcal{E}} \langle x.\ P \mid y.\ Q \Rrightarrow \Phi\rangle^{\mathcal{E}}_{\mathcal{E}'} \\ \forall y.\ Q \ {}^{\mathcal{E}'}\!\Rrightarrow^{\mathcal{E}} \Phi \end{cases}$$

The atomic update $\langle x.\ P \mid y.\ Q \Rrightarrow \Phi\rangle^{\mathcal{E}}_{\mathcal{E}'}$ encodes the obligation that one has to prove—potentially using some invariants in $\mathcal{E} \setminus \mathcal{E}'$ thanks to the mask-changing update ${}^{\mathcal{E}}\!\Rrightarrow^{\mathcal{E}'}$—an *atomic pre-condition* P that may depend on some existentially quantified value x. One then will have a choice—thanks to the classical conjunction $\wedge$—either to *abort* the update by returning P and get back the atomic update, or to *commit* the update

by assuming the *atomic post-condition* Q (which can depend on some new value y) to re-establish the invariants in $\mathcal{E} \setminus \mathcal{E}'$ (due to ${}^{\mathcal{E}'}\!\Rrightarrow^{\mathcal{E}}$) as well as the *private post-condition* Φ. Note that both Q and Φ may depend on both x and y.

Intuitively, the client of a logically atomic triple will prove this atomic update *i.e.*, they will have to produce x and the atomic pre-condition P, as well as proving the transformation of Q to Φ. Meanwhile, the prover of that logically atomic triple will consume this atomic update *i.e.*, they will consume P and produce y and the atomic post condition Q.

Definition 22.2 (Logically Atomic Triples)**.** A logically atomic triple in iRC11 is encoded using the atomic update instantiation for vProp (*i.e.*, P, Q, Φ are predicates on vProp) and the vProp weakest pre-condition.[6]

[6] see Definition 8.4

$$\langle x.\, P\rangle\, e \,\mathsf{in}\, \pi\, \langle y.\, Q\rangle_{\mathcal{E}}\{R\} ::= \\ \forall \Phi.\, \left(\langle x.\, P \mid y.\, Q \Rrightarrow (R \mathrel{-\!\!*} \Phi)\rangle^{\top\setminus\mathcal{E}}_{\mathcal{N}_{\text{HIST}}}\right) \mathrel{-\!\!*} \mathsf{wp}_{\top}\, e \,\mathsf{in}\, \pi\, \{\Phi\}$$

To say that e satisfies an atomic triple is to say that an atomic update for some post-condition $(R \mathrel{-\!\!*} \Phi)$ is at least the weakest pre-condition needed for e to result in a post-condition Φ.

$\mathcal{E}$ is the mask used by the prover of the triple to host *internal* invariants that are needed to verify the implementation. The client of the triple therefore can use any invariants outside of $\mathcal{E}$ *i.e.*, any invariants in $(\top \setminus \mathcal{E}) \setminus \mathcal{N}_{\text{HIST}}$. Recall that $\mathcal{N}_{\text{HIST}}$ is the mask needed by the model of the base logic of iRC11, so it cannot be used by the client either.[7]

[7] see more in Definition 7.18

Furthermore, the atomic triple also supports the *private post-condition* R which is produced by the prover of the triple and is only returned to the client *after* the invariants are established. This can be seen by unfolding the atomic update definition:[8] R appears after the mask-changing update ${}^{\mathcal{N}_{\text{HIST}}}\!\Rrightarrow^{\top\setminus\mathcal{E}}$. The private post-condition support will be needed in the specs of exchangers (see Chapter 24).

[8] see Definition 22.1

Recall the $\texttt{LAT}^{\text{abs}}_{\text{hb}}$ spec ABS-HB-ENQ for queues in Figure 22.1, we have

$$x ::= G, vs \\ y ::= (), G', M', V' \\ P ::= \mathsf{Queue}(q, vs, G) \\ Q ::= \mathsf{Queue}(q, vs +\!\!+ [(v, V')], G') * \ldots \\ R ::= \mathsf{True}$$

Therefore, ABS-HB-ENQ intuitively says that, given an atomic update that can atomically provide the queue state $\mathsf{Queue}(q, vs, G)$ and in return receive the updated state $\mathsf{Queue}(q, vs +\!\!+ [(v, V')], G')$ at the commit point, the implementation of **enq**$([q, v])$ can be verified.

Note that we always want to share a library's state ownership, *e.g.*, $\mathsf{Queue}(q, vs, G)$, so our specs always require it to be objective.[9] Recall that we can always make some resource objective by using the view-at modality.[10] We will then have to maintain that the local assertions, *e.g.*, $\mathsf{SeenQueue}(q, G, M)$ are tied to the views of the view-at modality used in the state ownership, *e.g.*, $\mathsf{Queue}(q, vs, G)$.

[9] see Definition 8.9

[10] see Definition 8.13

Last but not least, unsurprisingly an iRC11 logically atomic triple does imply a normal iRC11 Hoare triple,[11] that is

$$\langle x.\, P\rangle\ e \ \mathsf{in}\ \pi\ \langle y.\, Q\rangle_{\mathcal{E}}\{R\} \vdash \forall x.\, \{P\}\ e \ \mathsf{in}\ \pi\ \{y.\, Q * R\}_{\mathcal{E}}$$

This allows use to use iRC11 adequacy[12] to extract adequacy for a logically atomic triple.

[11] see Definition 8.5

[12] see Theorem 8.7

CHAPTER SUMMARY. In this chapter we have presented a set of concrete Compass specs for queues, and we have showed how to encode them with atomic updates in Iris. In the next chapter we will see how to perform verifications against these specs.

23

Verifications of Stacks and Queues

In this chapter, we demonstrate in more details the verifications of stack and queue implementations against Compass strong specs. In §23.1 we present a more complete version of the $\texttt{LAT}^{\text{abs}}_{\texttt{hb}}$ specs for queues given in Figure 22.1, and sketch the verification of a release-acquire implementation of the Michael-Scott queue.[1] We will also show how the specs imply more strongly consistent specs. In §23.2, we present a complete set of $\texttt{LAT}_{\texttt{hb}}$ specs for stacks, and sketch the verification of the release-acquire implementation of the Treiber stack[2] given in Figure 12.8.

[1] Michael and Scott, "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms" [MS96].

[2] Treiber, *Systems Programming: Coping with Parallelism* [Tre86].

23.1 Queue Specs and Verification of the Michael-Scott Queue

23.1.1 *Queue Specifications*

We present the remaining parts of our $\texttt{LAT}^{\text{abs}}_{\texttt{hb}}$ specs for queues in Figure 23.1, which complete those in Figure 22.1. We add the specs for the try versions who can fail due to contention. We also present the components of QueueConsistent. Again, the $\texttt{LAT}_{\texttt{hb}}$ specs for queues are simply the $\texttt{LAT}^{\text{abs}}_{\texttt{hb}}$ specs without the abstract state vs. ABS-HB-QUEUE-OBJ says that the queue's state assertion is objective, so it can be easily accessed from an objective invariant. We make explicit the namespace $\mathcal{N}_{\text{QUEUE}}$ which is needed to house an implementation's internal invariants, amd which can be picked by clients.

23.1.2 *RMC Implementations of Queues*

As mentioned in §22.2, we have verified two RMC queue implementations against Compass specs. We have verified a release-acquire implementation of the Michael-Scott queue[3] against the $\texttt{LAT}^{\text{abs}}_{\texttt{hb}}$ spec (see Figure 23.1), and therefore transitively against the $\texttt{LAT}_{\texttt{hb}}$ spec. We have verified an RMC implementation of the Herlihy-Wing queue[4] against the $\texttt{LAT}_{\texttt{hb}}$ spec.

[3] Michael and Scott, "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms" [MS96].

[4] Herlihy and Wing, "Linearizability: A Correctness Condition for Concurrent Objects" [HW90].

THE RELEASE-ACQUIRE IMPLEMENTATION OF THE MICHAEL-SCOTT QUEUE is given in Figure 23.2. This implementation follows the implementation given in the GPS logic.[5] The queue is implemented with a singly-linked list whose node contains a `next` field and a `data` field. Elements are enqueued at the `tail` and are dequeued from the `head` of the list.

[5] Turon et al., "GPS: navigating weak memory with ghosts, protocols, and separation" [TVD14].

`new_queue` allocates a new queue with a sentinel node s without a `data` field. Both the `head` and the `tail` of the list initially points to the

ABS-HB-NEW-QUEUE
$\{\mathsf{True}\}\ \texttt{new\_queue}()\ \{q.\ \mathsf{SeenQueue}^{\mathcal{N}_{\text{QUEUE}}}(q, \emptyset, \emptyset) * \mathsf{Queue}(q, [], \emptyset)\}_{\mathcal{N}_{\text{QUEUE}}}$

ABS-HB-TRY-ENQ
$\mathsf{SeenQueue}(q, G_0, M_0) * \sqsupseteq V \vdash$
$\langle G, vs.\ \mathsf{Queue}(q, vs, G)\rangle$

$\texttt{try\_enq}([q, v])$

$$\left\langle b.\ \exists vs', G' \sqsupseteq G, M' \sqsupseteq M_0, V' \sqsupseteq V.\ \mathsf{Queue}(q, vs', G') * \mathsf{SeenQueue}(q, G', M') * \sqsupseteq V' \right.$$
$$\left. * \vee \begin{cases} b = \texttt{false} & \wedge\ vs' = vs \wedge G' = G \\ b = \texttt{true} & \wedge\ vs' = vs +\!\!+ [(v, V')] \wedge \exists e \notin G.\ e \in M' \wedge G' = G[e \mapsto (\mathsf{Enq}(v), V', M')] \end{cases} \right\rangle_{\mathcal{N}_{\text{QUEUE}}}$$

ABS-HB-TRY-DEQ
$\mathsf{SeenQueue}(q, G_0, M_0) * \sqsupseteq V \vdash$
$\langle G, vs.\ \mathsf{Queue}(q, vs, G)\rangle$

$\texttt{try\_deq}([q])$

$$\left\langle v.\ \exists vs', G' \sqsupseteq G, M' \sqsupseteq M_0, V' \sqsupseteq V.\ \mathsf{Queue}(q, vs', G') * \mathsf{SeenQueue}(q, G', M') * \sqsupseteq V' \right.$$
$$\left. * \vee \begin{cases} v = \bot & \wedge\ vs' = vs \wedge G' = G \\ v = \epsilon & \wedge\ vs' = vs \wedge \exists d \notin G.\ d \in M' \wedge G' = G[d \mapsto (\mathsf{Deq}(\epsilon), V', M')] \\ v \notin \{\bot, \epsilon\} & \wedge\ \exists e, M_e, d \notin G, V_e.\ vs = (v, V_e) :: vs' \wedge G(e) = (\mathsf{Enq}(v), V_e, M_e) \\ & \quad \wedge\ (e, \_) \notin G.\mathsf{so} \wedge V_e \sqsubseteq V' \wedge M_e \cup \{e, d\} \sqsubseteq M' \\ & \quad \wedge\ G' = G[d \mapsto (\mathsf{Deq}(v), V', M')] \wedge G'.\mathsf{so} = \{(e, d)\} \cup G.\mathsf{so} \end{cases} \right\rangle_{\mathcal{N}_{\text{QUEUE}}}$$

ABS-HB-QUEUE-OBJ
$\mathsf{objective}(\mathsf{Queue}(q, vs, G))$

$\mathsf{QueueConsistent}(vs, G) :=$

$$\wedge \begin{cases} \forall (e, d) \in G.\mathsf{so}.\ \exists v.\ G(e).\mathsf{type} = \mathsf{Enq}(v) \wedge G(d).\mathsf{type} = \mathsf{Deq}(v) & \\ \quad \wedge\ G(e).\mathtt{view} \sqsubseteq G(d).\mathtt{view} \wedge G(e).\mathtt{logview} \sqsubseteq G(d).\mathtt{logview} & \text{(QUEUE-MATCHES-FULL)} \\ G.\mathsf{so} \text{ and } G.\mathsf{so}^{-1} \text{ are functional.} & \text{(QUEUE-SO-FUNCTIONAL)} \\ \forall d.\ G(d).\mathsf{type} = \mathsf{Deq}(\_) \to (\_, d) \notin G.\mathsf{so} \to G(d).\mathsf{type} = \mathsf{Deq}(\epsilon) & \text{(QUEUE-UNMATCHED-EMPDEQ)} \\ \forall (e, d) \in G.\mathsf{so}, e'.\ G(e').\mathsf{type} = \mathsf{Enq}(\_) \to (e', e) \in G.\mathsf{lhb} & \\ \quad \to \exists d'.\ (e', d') \in G.\mathsf{so} \wedge (e' \neq e \to (d, d') \notin G.\mathsf{lhb}) & \text{(QUEUE-FIFO-FULL)} \\ \forall d, e.\ G(d).\mathsf{type} = \mathsf{Deq}(\epsilon) \to G(e).\mathsf{type} = \mathsf{Enq}(\_) \to (e, d) \in G.\mathsf{lhb} & \\ \quad \to \exists d'.\ (e, d') \in G.\mathsf{so} \wedge (d', d) \in G.\mathsf{lhb} & \text{(QUEUE-EMPDEQ-FULL)} \end{cases}$$

FIGURE 23.1: Full $\mathtt{LAT}^{\text{abs}}_{\text{hb}}$ specs for queue

```
next := 0   data := 1
head := 0   tail := 1
new_queue ::=
  λ[].
  1: let s := alloc(1) in
  2: s + next :=na 0;
  3: let q := alloc(2) in
  4: q + head :=na s;
  5: q + tail :=na s;
  6: q

find_tail ::=
  λ[q].
  1: let n := *acq(q + tail) in
  2: let n' := *acq(n + next) in
  3: if n' = 0
  4: then n
  5: else
  6:   q + next :=rel n';
  7:   false

try_enq ::=
  λ[q, v].
  1: let n := alloc(2) in
  2: n + next :=na 0;
  3: n + data :=na v;
  4: let t :=
  5:   repeat (find_tail([q])) in
  6: if CAS^relacq(t + next, 0, n)
  7: then
  8:   q + tail :=rel n; true
  9: else free(n, 2); false

enq ::=
  rec try([q, v]) :=
  1: if try_enq([q, v])
  2: then ☣
  3: else try([q, v])

try_deq ::=
  λ[q].
  1: let h := *acq(q + head) in
  2: let n := *acq(h + next) in
  3: if n == 0
  4: then 0 // EMPTY
  5: else
  6:   if CAS^relacq(q + head, h, n)
  7:   then *na(n + 1)
  8:   else −1 // FAIL

deq ::=
  rec try([q]) :=
  1: let v := try_deq([q]) in
  2: if 0 ≤ v
  3: then v else try([q])
```

FIGURE 23.2: A release-acquire Michael-Scott queue

sentinel node.

`try_enq`$([q, v])$ first allocates a new node n with the `data` field storing the to-be-enqueued value v, and with the `next` field null (0). It then uses `find_tail`$([q])$ to try to find the "real" tail of the list: in the presence of concurrent enqueues, the value t one reads from the `tail` field is not guaranteed to be the most up-to-date tail. `try_enq` therefore uses a release-acquire **CAS** on $t +$ `next` to make sure that t is the real tail—the invariant is that the real tail's `next` field is always null. If the **CAS** succeeds, n is the new tail. Otherwise, n is deallocated. `enq`$([q, v])$ simply calls `try_enq`$([q, v])$ until it succeeds. One can consider optimizing `enq`$([q, v])$ by inlining `try_enq` to avoid allocating and deallocating n multiple times.

Reversely, `try_deq`$([q])$ reads the value n from the `next` field of q's `head`, and uses a release-acquire **CAS** on $q +$ `head` to make sure that n is the first node at the head. If the **CAS** succeeds, n is dequeued and the function reads n's `data` field non-atomically and returns the read value. n then becomes the new sentinel node. If the **CAS** fails, the function returns -1. Note that when n is null ($n == 0$), the function *subjectively* sees that the queue is empty and returns 0 (empty) immediately. `deq`$([q])$ simply calls `try_deq`$([q])$ until it succeeds or returns empty. (Technically, this means that we can only put positive numbers into the queue.)

THE RMC IMPLEMENTATION OF THE HERLIHY-WING QUEUE is given in Figure 23.3. This implementation follows the "weak" implementation from Raad et al.[6]. The queue has a backing store as an array of size N. When the size N is reached, *i.e.*, the queue is full, any enqueue will simply diverge. In practice, we may use a circular buffer as the backing store.

[6] Raad et al., "On library correctness under weak memory consistency: specifying and verifying concurrent libraries under declarative consistency models" [Raa+19].

new_queue allocates the array with size $N + 1$, and uses the first element of the array as the back counter to indicate the array slot that will be used for the next-to-be-enqueued element. back is initially set to 0. The implementation stores the elements from the index buff of the array. **new_queue** initialize the element array to all 0's (conveniently using back as a loop counter). We use 0 as the sentinel value to indicate that the array slot is not in used. Therefore the implementation only supports non-zero queue elements.

enq$([q, v])$ uses a release **FAA** to increase the back counter by 1 and to reserve the slot i (the returned value of the **FAA**) for its enqueue (line 1). If the previous value i is still in bound (of N), the reservation succeeded, and the function puts the value v in the slot i using a release write (line 3). Otherwise, the queue is full and the function diverges.

deq$([q])$ traverses the array from the index 0 to the minimum of N and the value b read from back. That is, in line 4 of **deq_loop**, j starts from $q +$ buff and ends with $q +$ buff $+ r$. The function then uses an acquire exchange to swap the element at j with 0. If the swapped value x from j was 0, it means that the slot j has been either reserved *or* dequeued by someone else, and the function continues with the next slot in the array. If x is non-zero, the dequeue succeeds, and x is returned. If the function has traversed the whole range r—that is when i is 0 in line 1 of **deq_loop**, the function uses the recursion dequeue to restart the traversal from the index 0.

As we can see, **deq** commits its effect in line 5 (of **deq_loop**) with a successful *acquire* exchange, and **enq** in line 3 with a *release* write. These are very relaxed operations: while they guarantee that matching enqueue-dequeue pairs are synchronized through the release-acquire idiom, they provide no guaranteed synchronization among enqueues nor among dequeues. Regardless, the queue still enjoys the FIFO property (QUEUE-FIFO-FULL), thanks to the combination of (1) the reservation order through back by enqueues *and* (2) the traversal order from index 0 by dequeues.

23.1.3 *The Verification of the Michael-Scott Queue*

We sketch the proof of the Michael-Scott queue implementation in Figure 23.2 against the $\texttt{LAT}^{\text{abs}}_{\text{hb}}$ spec in Figure 23.1.[7] The gist of the proofs is the definitions of the queue's state ownership Queue(q, vs, G) (Definition 23.7) and the queue's local observation SeenQueue(q, G, M) (Definition 23.9).

[7] The proof of the Herlihy-Wing will be elided, but is available in the Coq development.

Definition 23.1 (Extra ghost state for Michael-Scott queue)**.** We use the authoritative RA to create the master-snapshot ghost state for 4 types of data:

```
back := 0   buff := 1
new_queue ::=
  λ[].
  1: let q := alloc(N + 1) in
  2: q + back :=_na 0;
  3: repeat (
  4:   let c := *(q + back) in
  5:   q + buff + c := 0;
  6:   q + back := c + 1;
  7:   c + 1 == N
  8: );
  9: q + back := 0; q

enq ::=
  λ[q, v].
  1: let i := FAA^rel(q + back, 1) in
  2: if i < N
  3: then q + buff + i :=_rel v
  4: else diverge([☣])
```

```
deq_loop ::=
  λ[dequeue, q, r].
  rec loop([i]) :=
  1: if i == 0
  2: then dequeue([q])
  3: else
  4:   let j := q + buff + r − i in
  5:   let x := xchg^acq(j, 0) in
  6:   if x == 0
  7:   then loop([i − 1])
  8:   else x

deq ::=
  rec dequeue([q]) :=
  1: let b := *^acq(q + back) in
  2: let r := min(b, N) in
  3: deq_loop([dequeue, q, r])([r])
```

FIGURE 23.3: An RMC Herlihy-Wing Queue

- We need $\mathsf{master}^\gamma(G)$ to record the current graph G for some ghost location γ, and $\mathsf{snap}^\gamma(G')$ for some snapshot G' of G ($G' \sqsubseteq G$). This type of ghost state is needed generally in the graph-based specs of Compass.

- We need $\mathsf{master}^\gamma(\mathcal{T})$ to store the persistent mapping from a queue's node identified by some ghost location η to some enqueue event e. That is, $\mathcal{T} \in \mathit{GName} \xrightarrow{\mathsf{fin}} \mathit{EventId}$. The ghost state $\mathsf{snap}^\gamma(\mathcal{T}')$ records that $\mathcal{T}'$ is a snapshot of $\mathcal{T}$ ($\mathcal{T}' \sqsubseteq \mathcal{T}$).

- We need $\mathsf{master}^\gamma(\mathcal{L})$ to store the current list of enqueued nodes, where $\mathcal{L} \in (\mathit{GName} \times \mathit{Loc})^*$ is a list of pairs of a node identifier and the node's physical location (η, ℓ). The ghost state $\mathsf{snap}^\gamma(\mathcal{L}')$ records that $\mathcal{L}'$ is a snapshot of $\mathcal{L}$ ($\mathcal{L}' \sqsubseteq \mathcal{L}$).

- We need $\mathsf{master}^\gamma(\mathcal{D})$ to store the current list of dequeued nodes, where $\mathcal{D} \in (\mathit{GName} \times \mathit{Loc})^*$. The ghost state $\mathsf{snap}^\gamma(\mathcal{D}')$ records that $\mathcal{D}'$ is a snapshot of $\mathcal{D}$ ($\mathcal{D}' \sqsubseteq \mathcal{D}$). Note that we maintain that $\mathcal{D} \sqsubseteq \mathcal{L}$.

Definition 23.2 (Ownership of a node)**.** We sketch the ownership of a node of the queue. Owning a node with identifier η and physical location ℓ means the atomic ownership of $\ell + \mathtt{next}$'s field with the atomic period η. The history h of η can only be in two states (represented by n): (1) the initial state where $\ell + \mathtt{next}$ is null (0), or (2) the "linked" state where

$\ell + \mathsf{next}$ points to the next node (η', ℓ').

$$
\begin{array}{l}
\mathsf{ownNode}^{G,\mathcal{T}}(\eta, \ell, n) ::= \\
\quad \exists t_0, V_0, h.\, \ell + \mathsf{next} \overset{\eta}{\mapsto}_{\mathtt{con}} h * \\
\quad \begin{cases}
n = \mathsf{None} & h = [t_0 \leftarrow (0, V_0)] \\
n = \mathsf{Some}(\eta', \ell, d') & \exists v', V', e'.\, \mathcal{T}(\eta') = e' \\
& * \, h = [t_0 \leftarrow (0, V_0)][t_0 + 1 \leftarrow (\ell', V')] \\
& * \, @_{V'} \left(\mathsf{seenEnq}(G, e', v') * \ell' \sqsupseteq^{\eta'}_{\mathsf{sy}} [\_ \leftarrow (0, \_)] \right) \\
& * \, \mathbf{if}\ d'\ \mathbf{then}\ \mathsf{True}\ \mathbf{else}\ @_{V'}\ (\ell' + \mathtt{data} \mapsto v')
\end{cases}
\end{array}
$$

where

$$
\begin{array}{l}
\mathsf{seenEnq}(G, e', v') ::= \\
\quad G(e').\mathtt{type} = \mathsf{Enq}(v') * \mathsf{seenEvt}(G, e') * \\
\quad \mathsf{syncEnqLogView}(G, G(e').\mathtt{logview}) \\
\mathsf{seenEvt}(G, e') ::= \\
\quad \sqsupseteq G(e').\mathtt{view} * @_{G(e').\mathtt{view}}(\mathsf{seenLogView}(G, G(e').\mathtt{logview})) \\
\mathsf{seenLogView}(G, M) ::= \forall e \in M.\, \sqsupseteq G(e).\mathtt{view} \\
\mathsf{syncEnqLogView}(G, M) ::= \\
\quad \forall e \in M.\, G.(e) = \mathsf{Enq}(\_) \Rightarrow G(e).\mathtt{logview} \subseteq M
\end{array}
$$

(QUEUE-MS-SYNC-ENQS)

In the latter case, ownNode of η also owns $\mathsf{seenEnq}(G, e', v')$, the observation that the next node (η', ℓ') has been enqueued by the enqueue event e' in a snapshot graph G, with the value v', as well as the observation $\ell' + \mathsf{next} \sqsupseteq^{\eta'}_{\mathsf{sy}} [\_ \leftarrow (0, \_)]$ that $\ell' + \mathsf{next}$ field has been initialized to null. Furthermore, if the next node (η', ℓ') has *not* been dequeued—represented by $d' == \mathbf{false}$, then the ownership of η holds on to the data field of η', *i.e.*, $\ell' + \mathsf{data} \mapsto v'$.

The reason we put the data field of η' in the ownership of its previous node η is that, when dequeueing η', the implementation simply updates the head pointer from ℓ (of η) to ℓ' (of η'). Before doing that, the implementation would have read $\ell + \mathsf{next}$ and would have acquired only the value ℓ' and the view V'. That is, the dequeue can only acquire the ownership $\ell' + \mathsf{data} \mapsto v'$ safely after having acquired V' from the history h of $\ell + \mathsf{next}$. It is therefore necessary to tie $\ell' + \mathsf{data}$ field to $\ell + \mathsf{next}$ field.

Remark 23.3 (Logical happens-before vs. physical happens-before)**.** Note that the observation $\mathsf{seenEnq}(G, e', v')$ enforces strong synchronization properties that are satisfiable by the release-acquire Michael-Scott queue implementation in Figure 23.2, but not necessarily all implementations. It requires $\mathsf{seenEvt}(G, e')$ that says that the observation also observes not only the physical view of the event e' ($\sqsupseteq G(e').\mathtt{view}$), but also transitively (through the definition of $\mathsf{seenLogView}(G, M)$) all physical views of all events e's that *logically happen before* e'. We note that

the fact that $e \in G(e').\texttt{logview}$—$e$ logically happens before e'—does not necessarily always means that e *physically* happens before e'—it is the $\mathsf{seenLogView}(G, M)$ condition that enforces such interpretation. Furthermore, logical happens-before among graph events is not necessarily transitive: $e_1 \in G(e_2).\texttt{logview}$ and $e_2 \in G(e_3).\texttt{logview}$ does not need to imply $e_1 \in G(e_3).\texttt{logview}$. In fact, we only require QUEUE-MS-SYNC-ENQS (see the definition of $\mathsf{syncEnqLogView}(G, M)$) that says that logical happens-before is transitive at least for enqueue events. This generality of allowing logical happens-before to deviate from physical happens-before is not really needed for our release-acquire implementation, but can become useful to allow for more relaxed implementations (*e.g.*, the Herlihy-Wing queue or the Treiber stack) which interpret logical happens-before more "relaxedly".

Definition 23.4 (Ownership of all nodes)**.** All nodes of a queue include the sentinel node (η_0, s_0), the dequeued nodes $\mathcal{D}$, and the nodes that are still in the queue $\mathcal{L}_{\mathsf{in}}$

$$
\begin{aligned}
&\mathsf{ownNodes}^{G,\mathcal{T}}(\eta_0, s_0, \mathcal{D}, \mathcal{L}_{\mathsf{in}}) ::= \\
&\quad \textbf{let}\ \mathcal{L} := (\eta_0, s_0) +\!\!+ \mathcal{D} +\!\!+ \mathcal{L}_{\mathsf{in}}\ \textbf{in} \\
&\quad \textbf{let}\ \mathcal{L}' := \mathsf{nextNodes}(\eta_0, s_0, \mathcal{D}, \mathcal{L}_{\mathsf{in}})\ \textbf{in} \\
&\qquad \mathop{\Large *}_{(\eta,\ell,n)\in(\mathcal{L},\mathcal{L}')} \mathsf{ownNode}^{G,\mathcal{T}}(\eta, \ell, n)
\end{aligned}
$$

The ownership ownNodes simply collects all ownNode. The definition relies on the function nextNodes that computes the correct next node information n for a node. For all but the last node in $\mathcal{L}$, n will be Some of the next node in the list $\mathcal{L}$. For all but the last node in $(\eta_0, s_0) +\!\!+ \mathcal{D}$, n will have $d' == \textbf{true}$.

We note that all nodes in $\mathcal{L}$ are unique, because each is identified by a logical name η which in turn is uniquely identified by an enqueue event.

Definition 23.5 (Observations of node links)**.** We define the observations of node links.

$$
\begin{aligned}
&\mathsf{seenLink}^{G,\mathcal{T},\mathcal{L}}(\eta, \ell, \eta', \ell', e', v') ::= \\
&\quad \mathcal{T}(\eta') = e' * \mathsf{seenEnq}(G, e', v') * \ell' \sqsupseteq^{\eta'}_{\mathsf{sy}} [\_ \leftarrow (0, \_)] * \\
&\quad (\eta, \ell) +\!\!+ (\eta', \ell') \sqsubseteq \mathcal{L} * \ell \sqsupseteq^{\eta}_{\mathsf{sy}} [\_ \leftarrow (\ell', \_)] \\
&\mathsf{seenLinks}^{G,\mathcal{T},\mathcal{L}}(M) ::= \\
&\quad \forall e' \in M \cap G.\, G(e').\texttt{type} = \mathsf{Enq}(v') \Rightarrow \\
&\qquad \exists \eta, \ell, \eta', \ell'.\, \mathsf{seenLink}^{G,\mathcal{T},\mathcal{L}}(\eta, \ell, \eta', \ell', e', v') \\
&\mathsf{headLinks}^{G,\mathcal{T},\mathcal{D}}(\overline{V}) ::= \\
&\qquad \mathop{\Large *}_{(\eta',\ell',V)\in(\mathcal{D},\overline{V})} @_V(\exists e'.\, \mathsf{seenLink}^{G,\mathcal{T},\mathcal{D}}(\_, \_, \eta', \ell', e', \_) * (e', \_) \in G.\mathsf{so} * \ldots) \\
&\mathsf{tailLinks}^{G,\mathcal{T},\mathcal{L}}(h) ::= \\
&\quad \mathop{\Large *}_{(t,v,V)\in h} @_V(\exists \eta', \ell', e'.\, v = \ell' * \mathsf{seenLink}^{G,\mathcal{T},\mathcal{L}}(\_, \_, \eta', \ell', e', \_))
\end{aligned}
$$

- The observation $\mathsf{seenLink}^{G,\mathcal{T},\mathcal{L}}(\eta, \ell, \eta', \ell', e', v')$ says that the node (η', ℓ') has been enqueued to the graph G with the event e' (η'

is mapped to e' in $\mathcal{T}$) and with the value v'. We have seen this information tracked also in ownNode. Additionally, seenLink also observes that (η', ℓ') is linked to the node (η, ℓ), because it immediately follows (η, ℓ) in the node list $\mathcal{L}$.

- The observation $\mathsf{seenLinks}^{G,\mathcal{T},\mathcal{L}}(M)$ collects all of those single-link observations for all enqueue events in the logical view M.
- The observation $\mathsf{headLinks}^{G,\mathcal{T},\mathcal{D}}(\overline{V})$ collects the single-link observations for all nodes that the head has seen, *i.e.*, all nodes that have been *dequeued*, as signaled by the meta-variable $\mathcal{D}$. The list of views $\overline{V}$ will come from the **CAS**es to head. This tracks the fact that every **CAS** to the head—which dequeues a node η—is also synchronized with the enqueue event of η.
- The observation $\mathsf{tailLinks}^{G,\mathcal{T},\mathcal{L}}(h)$ collects the single-link observations for all nodes that the `tail` has seen, *i.e.*, all nodes that have been written to the history h of the `tail`. This will allow the `tail` to *release* those observations to whoever reads the `tail` with an acquire read.

Definition 23.6 (Ownership of a queue's physical locations)**.** The physical locations of a queue include all nodes and the head and the tail.

$$
\begin{aligned}
&\mathsf{ownQueueLocs}^{G,\mathcal{T}}_{\gamma_h,\gamma_t}(\eta_0, s_0, \mathcal{D}, \mathcal{L}_{\mathsf{in}}) ::= \\
&\quad \mathsf{ownNodes}^{G,\mathcal{T}}(\eta_0, s_0, \mathcal{D}, \mathcal{L}_{\mathsf{in}}) * \\
&\quad \exists \overline{V}.\ q + \mathsf{head} \mapsto^{\gamma_h}_{\mathtt{con}} (\_, s_0 +\!\!+ \mathcal{D}.2, \overline{V}) * \mathsf{headLinks}^{G,\mathcal{T},(\eta_0,s_0)+\!\!+\mathcal{D}}(\overline{V}) * \\
&\quad \exists h_t.\ q + \mathtt{tail} \mapsto^{\gamma_t}_{\mathtt{con}} h_t * \mathsf{tailLinks}^{G,\mathcal{T},(\eta_0,s_0)+\!\!+\mathcal{D}+\!\!+\mathcal{L}_{\mathsf{in}}}(h_t)
\end{aligned}
$$

Definition 23.7 (Ownership of a queue's state)**.** The ownership of a queue's state $\mathsf{Queue}^{\gamma_g}(q, vs, G)$ is implemented purely with ghost state. Since we are verifying the implementation against the $\mathtt{LAT}^{\mathrm{abs}}_{\mathsf{hb}}$ spec with an abstract queue vs, we additionally employ a master-snapshot instance for that abstract state. We expose halves of the master ghost state for G and vs. The other halves will be governed by the queue's internal invariant.

$$
\begin{aligned}
&\mathsf{Queue}^{\gamma_g}(q, vs, G) ::= \\
&\quad \mathsf{QueueConsistent}(vs, G) * \mathsf{master}^{\gamma_g}_{1/2}(G) * \mathsf{master}_{1/2}(vs)
\end{aligned}
$$

$\mathsf{Queue}^{\gamma_g}(q, vs, G)$ is trivially objective.

Definition 23.8 (The queue's internal invariant)**.** We define the internal invariant $\boxed{\mathsf{queuel}(q, \gamma, \gamma_g, \gamma_h, \gamma_t)}^{\mathcal{N}_{\mathrm{QUEUE}}}$ that is needed by verification and that is not exposed to clients of the queue specs.

$$
\begin{aligned}
&\mathsf{queuel}(q, \gamma, \gamma_g, \gamma_h, \gamma_t) ::= \\
&\quad \exists vs, G.\, \mathsf{Queue}^{\gamma_g}(q, vs, G) * \exists \mathcal{T}, \eta_0, s_0, \mathcal{D}, \mathcal{L}_{\mathsf{in}}. \\
&\quad \mathbf{let}\ \mathcal{L} := (\eta_0, s_0) +\!\!+ \mathcal{D} +\!\!+ \mathcal{L}_{\mathsf{in}}\ \mathbf{in} \\
&\quad \mathsf{master}^{\gamma\cdot 1}(\mathcal{L}) * \mathsf{master}^{\gamma\cdot 2}(\mathcal{D}) * \mathsf{master}^{\gamma\cdot 3}(\mathcal{T}) * \\
&\quad \exists V_b.\ @_{V_b}\mathsf{ownQueueLocs}^{G,\mathcal{T}}_{\gamma_h,\gamma_t}(\eta_0, s_0, \mathcal{D}, \mathcal{L}_{\mathsf{in}}) * \\
&\quad \mathsf{queueViews}(G, \mathcal{T}, \mathcal{L}) * \mathsf{queueProps}(vs, G, \mathcal{T}, \eta_0, s_0, \mathcal{D}, \mathcal{L}_{\mathsf{in}})
\end{aligned}
$$

where

$$\mathsf{queueViews}(G, \mathcal{T}, \mathcal{L}) ::=$$
$$\forall e \in G.\, @_{G(e).\mathtt{view}} \left(\mathsf{seenLinks}^{G,\mathcal{T},\mathcal{L}}(G(e).\mathtt{logview})\right)$$

$$\mathsf{queueProps}(vs, G, \mathcal{T}, \eta_0, s_0, \mathcal{D}, \mathcal{L}_{\mathsf{in}}) ::=$$
$$\wedge \begin{cases} ((\eta_0, s_0) +\!\!+ \mathcal{D} +\!\!+ \mathcal{L}_{\mathsf{in}}).1 \text{ does not have duplicated node ids} & \text{(QUEUE-MS-NO-DUP)} \\ \mathtt{dom}(\mathcal{T}) = (\mathcal{D} +\!\!+ \mathcal{L}_{\mathsf{in}}).1 & \text{(QUEUE-MS-ENQ-ONLY-1)} \\ \mathtt{codom}(\mathcal{T}) = \{e \in G \mid G(e).\mathsf{type} = \mathsf{Enq}(\_)\} & \text{(QUEUE-MS-ENQ-ONLY-2)} \\ \mathcal{T} \text{ is injective} & \text{(QUEUE-MS-MAP-INJ)} \\ \{e \mid \mathcal{T}(\eta) = e \wedge (\eta, \_) \in \mathcal{D}\} = G.\mathsf{so}.1 & \text{(QUEUE-MS-MAP-DEQ-GRAPH)} \\ \forall e_1, e_2, \eta_1, \eta_2.\, \mathcal{T}(\eta_1) = e_1 \Rightarrow \mathcal{T}(\eta_2) = e_2 \Rightarrow \\ \quad \eta_1 \text{ precedes } \eta_2 \text{ in } \mathcal{D} +\!\!+ \mathcal{L}_{\mathsf{in}} \Rightarrow e_1 \leq e_2 & \text{(QUEUE-MS-ENQ-MO)} \\ \forall e_1, e_2, \eta_1, \eta_2.\, \mathcal{T}(\eta_1) = e_1 \Rightarrow \mathcal{T}(\eta_2) = e_2 \Rightarrow \\ \quad \eta_1 \text{ precedes } \eta_2 \text{ in } \mathcal{D} +\!\!+ \mathcal{L}_{\mathsf{in}} \Rightarrow e_1 \in G(e_2).\mathtt{logview} & \text{(QUEUE-MS-ENQ-HB)} \\ vs = \{v \mid \eta \in \mathcal{L}_{\mathsf{in}}.1 \wedge \mathcal{T}(\eta) = e \wedge G(e).\mathsf{type} = \mathsf{Enq}(v)\} & \text{(QUEUE-MS-ABS)} \end{cases}$$

The invariant contains:

- The other halves of the ghost state for the abstract queue vs and the graph G, collected in $\mathsf{Queue}^{\gamma_g}(q, vs, G)$. This means that the invariant is always in agreement on the state with the client, who owns the "main" halves in their copy of $\mathsf{Queue}^{\gamma_g}(q, vs, G)$.

- The master (authoritative) ghost state instances for the list $\mathcal{L}$ of enqueued nodes ($\mathsf{master}^{\gamma \cdot 1}(\mathcal{L})$), the list $\mathcal{D}$ of dequeued nodes ($\mathsf{master}^{\gamma \cdot 2}(\mathcal{D})$), and the mapping $\mathcal{T}$ from node ids to the graph G's event ids ($\mathsf{master}^{\gamma \cdot 3}(\mathcal{T})$).

- The ownership $\mathsf{ownQueueLocs}^{G,\mathcal{T}}_{\gamma_h,\gamma_t}(\eta_0, s_0, \mathcal{D}, \mathcal{L}_{\mathsf{in}})$ of all physical locations of the queue, put under a view-at modality to make sure that the invariant is objective.

- The observations $\mathsf{queueViews}(G, \mathcal{T}, \mathcal{L})$ of node links per graph event. Intuitively, we require that the physical view of an event e in G justifies all single-link observations for all enqueued events that happen before e (that is, all enqueued events observed by e's logical view).

- The properties queueProps connecting the abstract state vs with the graph G and the ghost state $\mathcal{T}$ and $\mathcal{L}$.

  – QUEUE-MS-NO-DUP says that all nodes in a queue are unique.

- QUEUE-MS-ENQ-ONLY-1 says that the map $\mathcal{T}$ tracks all nodes in $(\mathcal{D} +\!\!+ \mathcal{L}_{\mathsf{in}})$ and QUEUE-MS-ENQ-ONLY-2 says that those nodes are all of those that have ever been put into the queue.
- QUEUE-MS-MAP-INJ makes sure that each enqueue event in G is tied to exactly on node id.
- QUEUE-MS-MAP-DEQ-GRAPH makes sure that $\mathcal{D}$ are all dequeued nodes.
- QUEUE-MS-ENQ-MO says that the order in the enqueued node list agrees with the insertion order of event ids into the graph G.
- QUEUE-MS-ENQ-HB requires that a node η_1 enqueued earlier than η_2 (earlier in the order in the enqueued node list) also happens before η_2. This is a very strong synchronization property that holds for the release-acquire Michael-Scott queue, but may *not* hold for all queue implementations.
- QUEUE-MS-ABS says that the node list $\mathcal{L}_{\mathsf{in}}$ indeed is the nodes that are in the queue and agrees with the abstract queue vs.

Definition 23.9 (Local observation of a queue's state)**.** The local observation of a queue's state $\mathsf{SeenQueue}(q, G, M)$ is implemented with the snapshots of the queue's ghost state instances, the queue internal invariant, the observations of the queue's node links, as well as the observations of the queue's `head` and `tail` physical locations.

$$
\begin{aligned}
&\mathsf{SeenQueue}^{\gamma_g}(q, G, M) ::= \\
&\quad \mathsf{snap}^{\gamma_g}(G) * \mathsf{seenLogView}(G, M) * \\
&\quad \exists \gamma, \gamma_h, \gamma_t.\ \boxed{\mathsf{queuel}(q, \gamma, \gamma_g, \gamma_h, \gamma_t)}^{\mathcal{N}_{\text{QUEUE}}} * \\
&\quad \exists \mathcal{T}, \eta_0, s_0, \mathcal{D}, \mathcal{L}_{\mathsf{in}}.\ \mathbf{let}\ \mathcal{L} := (\eta_0, s_0) +\!\!+ \mathcal{D} +\!\!+ \mathcal{L}_{\mathsf{in}}\ \mathbf{in} \\
&\quad \mathsf{snap}^{\gamma\cdot 1}(\mathcal{L}) * \mathsf{snap}^{\gamma\cdot 2}(\mathcal{D}) * \mathsf{snap}^{\gamma\cdot 3}(\mathcal{T}) * \\
&\quad \mathsf{syncEnqLogView}(G, M) * \\
&\quad \mathsf{seenLinks}^{G,\mathcal{T},\mathcal{L}}(M) * \\
&\quad q + \mathtt{tail} \sqsupseteq^{\gamma_t}_{\mathtt{sn}} \_ * \exists \mathcal{D}' \sqsubseteq \mathcal{D}.\ q + \mathsf{head} \sqsupseteq^{\gamma_h}_{\mathtt{sn}} (\_, s_0 +\!\!+ \mathcal{D}'.2, \_) * \ldots
\end{aligned}
$$

More concretely,

- $\mathsf{snap}^{\gamma_g}(G)$ enforces that G is a snapshot of the current graph (who is stored in the master ghost state in the internal invariant).
- $\mathsf{seenLogView}(G, M)$ enforces that the current thread has observed all physical views of all events in the logical view M. Additionally, $\mathsf{syncEnqLogView}(G, M)$ says that the thread observes all enqueue events that transitively happen-before the events in M.
- $\mathsf{snap}^{\gamma\cdot 1}(\mathcal{L})$, $\mathsf{snap}^{\gamma\cdot 2}(\mathcal{D})$, and $\mathsf{snap}^{\gamma\cdot 3}(\mathcal{T})$ together show the the current thread has extracted the snapshots $\mathcal{L}$, $\mathcal{D}$, and $\mathcal{T}$ of the corresponding ghost instances from the internal invariant.
- $\mathsf{seenLinks}^{G,\mathcal{T},\mathcal{L}}(M)$ says that the current thread observes all enqueuing links for all enqueue events in η, which must already be included in the snapshot $\mathcal{L}$.

- $q + \texttt{tail} \sqsupseteq^{\gamma_t}_{\texttt{sn}} \_$ says that the thread has some observation on the queue `tail`, which is enough to perform operations on the tail.

- $q + \mathsf{head} \sqsupseteq^{\gamma_h}_{\texttt{sn}} (\_, s_0 +\!\!+ \mathcal{D}'.2, \_)$ also says that the thread has some observation on the queue head, and that observation must follow the dequeued list snapshot $\mathcal{D}$ ($\mathcal{D}' \sqsubseteq \mathcal{D}$). This is because dequeues are done exclusively on the head (using CASes).

Below, we sketch the proofs of Abs-Hb-Try-Enq and Abs-Hb-Try-Deq.

Proof sketch of **`try_enq`**$([q, v])$. We start by unfolding the logically atomic triple (Definition 22.2), so we have in our context $\mathsf{SeenQueue}(q, G_0, M_0)$, $\sqsupseteq V$, and the atomic update $\langle G, vs.\ \mathsf{Queue}(q, vs, G) \mid b.\ \ldots \Rrightarrow \Phi \rangle^{\top \setminus \mathcal{N}_{\text{QUEUE}}}_{\mathcal{N}_{\text{HIST}}}$, and our goal is $\mathsf{wp}_{\top}$ **`try_enq`**$([q, v])$ in π $\{\Phi\}$.

Lines 1 to 3 (see Figure 23.2) are easy, as they allocate and initialize a fresh node non-atomically. Afterwards, we have $n + \texttt{data} \mapsto v$ and $n + \texttt{next} \mapsto 0$.

Next, in lines 4 and 5, to make a call to **`find_tail`**$([q])$, we apply its specs, given below (and proven separately).

Abs-Hb-Find-Tail

$$\left\{ q + \texttt{tail} \sqsupseteq^{\gamma_t}_{\texttt{sn}} \_ * \boxed{\mathsf{queuel}(q, \gamma, \gamma_g, \gamma_h, \gamma_t)}^{\mathcal{N}_{\text{QUEUE}}} \right\}$$

`find_tail`$([q])$ in π

$$\left\{ \begin{array}{l} t.\, t = 0 \lor \\ \quad \exists \ell_t.\, t = \ell_t \land \exists \eta_t, G_t, \mathcal{T}_t, \mathcal{L}_t.\, \mathsf{snap}^{\gamma_g}(G_t) * \mathsf{snap}^{\gamma \cdot 1}(\mathcal{L}_t) * \mathsf{snap}^{\gamma \cdot 3}(\mathcal{T}_t) \\ \quad * \mathsf{seenLink}^{G_t, \mathcal{T}_t, \mathcal{L}_t}(\_, \_, \eta_t, \ell_t, \_, \_) \end{array} \right\}$$

That is, when **`find_tail`**$([q])$ returns a (non-null) location ℓ_t (which the current thread sees as a tail because $\ell + \texttt{next}$ is still 0), it also returns the observation that ℓ_t has been enqueued as some node η' in the snapshots of G_t, $\mathcal{T}_t$, and $\mathcal{L}_t$. The proof of this spec simply open the internal invariant $\mathsf{queuel}(\ldots)$ to access the atomic points-to of the `tail`.

After line 5, we have $t = \ell_t$ and $\mathsf{seenLink}^{G_t, \mathcal{T}_t, \mathcal{L}_t}(\_, \_, \eta_t, \ell_t, \_, \_)$. In line 6, we have a CAS that tries to set $\ell_t + \texttt{next}$ from null (0) to n. To verify this CAS, we use an iRC11 rule with atomic points-to (like AT-CAS-SN-GEN in Figure 10.5). We therefore need to extract (1) the seen-history observation $\ell_t \sqsupseteq^{\eta_t}_{\texttt{sn}} [\_ \leftarrow (0, \_)]$ from $\ell_t \sqsupseteq^{\eta_t}_{\texttt{sy}} [\_ \leftarrow (0, \_)]$ in $\mathsf{seenLink}^{G_t, \mathcal{T}_t, \mathcal{L}_t}(\ldots)$, and (2) the atomic points-to $\ell_t + \texttt{next} \overset{\eta}{\mapsto}_{\texttt{con}} \_$ from $\mathsf{ownQueueLocs}^{G, \mathcal{T}}_{\gamma_h, \gamma_t}(\ldots)$ from the queue invariant $\mathsf{queuel}(q, \gamma, \gamma_g, \gamma_h, \gamma_t)$.

If the CAS fails, we simply return the points-to to the invariant—we do not update the ghost state instances. Nevertheless, we need to commit the atomic update $\langle G, vs.\ \mathsf{Queue}(q, vs, G) \mid b.\ \ldots \Rrightarrow \Phi \rangle^{\top \setminus \mathcal{N}_{\text{QUEUE}}}_{\mathcal{N}_{\text{HIST}}}$ from our context, with the return value $b =$ **`false`** and $G' = G$ and $M' = M_0$.

On the other hand, if the CAS succeeds, we have to

(1) turn the non-atomic points-to of n into an atomic points-to with a new node id η which is also the atomic points-to's ghost location;

(2) update the graph to $G' = G[e \mapsto (\mathsf{Enq}(v), V', M')]$ for some newly picked event id $e \notin G$ and logical view M';

(3) map the node id η for the newly enqueued node n to the current master $\mathcal{T}$ (from $\mathsf{master}^{\gamma\cdot 3}(\mathcal{T})$ in the invariant) to e;

(4) append the current master enqueued list $\mathcal{L}$ (from $\mathsf{master}^{\gamma\cdot 1}(\mathcal{L})$ in the invariant) with (η, n);

(5) establish the updated observations $\mathsf{queueViews}(\ldots)$ and properties $\mathsf{queueProps}(\ldots)$ for the updated queue;

(6) establish the queue consistency $\mathsf{QueueConsistent}(vs{+\!\!+}[(v, V')], G')$.

Here, we commit the atomic update with the return value $b = \mathbf{true}$. We update of the graph (step (2)) using both halves of the graph master ($\mathsf{master}^{\gamma_g}_{1/2}(G)$) from the two pieces of $\mathsf{Queue}^{\gamma_g}(q, vs, G)$, one provided by the client through our committing of the atomic update and one through our opening of the invariant queuel.

In step (3), we update $\mathsf{master}^{\gamma\cdot 3}(\mathcal{T})$ to $\mathsf{master}^{\gamma\cdot 3}(\mathcal{T}')$ where $\mathcal{T}' = \mathcal{T}[\eta \mapsto e]$. In step (4), we update $\mathsf{master}^{\gamma\cdot 1}(\mathcal{L})$ to $\mathsf{master}^{\gamma\cdot 1}(\mathcal{L}')$ where $\mathcal{L}' = \mathcal{L} +\!\!+ [(\eta, n)]$.

By accomplishing step (1), we get $\mathsf{ownNode}^{G',\mathcal{T}'}(\eta, n, \mathsf{None})$ for the new node (η, n). Furthermore, to update the previous tail (η_t, ℓ_t) to link to the new node, we need to produce $\mathsf{ownNode}^{G',\mathcal{T}'}(\eta_t, \ell_t, \mathsf{Some}(\eta, n))$. To do so, we need to show $@_{V'}\mathsf{seenEnq}(G', e, v)$, which in turn requires us to show $\mathsf{syncEnqLogView}(G', M')$. In other words, we need to maintain QUEUE-MS-SYNC-ENQS for the new logical view. For that reason, we pick $M' = M_0 \cup \{e\} \cup G(e_t).\mathtt{logview}$ where $\mathcal{T}(\eta_t) = e_t$. That is, after the successful enqueue, the thread extends its logical view (logical happens-before set) to include not only the new enqueue event e but also all events from the previous tail η_t. This encodes the fact that, in this release-acquire implementation, enqueues are synchronized with one another.

In a related node, in step (5), to show $\mathsf{queueViews}(G', \mathcal{T}', \mathcal{L}')$, we need to show

$$@_{V'}\mathsf{seenLinks}^{G',\mathcal{T}',\mathcal{L}'}(M')$$

For all the nodes in the η part of M', we get the observations from the original $\mathsf{SeenQueue}(q, G_0, M_0)$ assumption we have received at the very beginning. For all the nodes in the $G(e_t).\mathtt{logview}$ part of M', we get the observations the old $\mathsf{queueViews}(G, \mathcal{T}, \mathcal{L})$ from the invariant. It remains to show

$$@_{V'}\mathsf{seenLink}^{G',\mathcal{T}',\mathcal{L}'}(\eta_t, \ell_t, \eta, n, e, v)$$

which follows from $@_{V'}\mathsf{seenEnq}(G', e, v)$ that has been shown right above.

In step (5), we also need to establish

$$\mathsf{queueProps}(vs +\!\!+ [(v, V')], G', \mathcal{T}', \eta_0, s_0, \mathcal{D}, \mathcal{L}_{\mathsf{in}} +\!\!+ [(\eta, n)])$$

The most two important properties are QUEUE-MS-ENQ-MO and QUEUE-MS-ENQ-HB (see Definition 23.8). We achieve QUEUE-MS-ENQ-MO by maintaining an event id allocating scheme that follows the relation $\leq$ among event

ids. We achieve QUEUE-MS-ENQ-HB through the way we pick $G(e).\texttt{logview}$ for a new enqueue event e: as can be seen from M', $G(e).\texttt{logview}$ always includes e itself, and $G(e).\texttt{logview}$ also includes $G(e_t).\texttt{logview}$ of the preceding enqueue event (the previous tail) e_t. Consequently, $e_t \in G(e).\texttt{logview}$.

Step (6) is rather straightforward, because the properties in the queue consistency mostly concern dequeues. With that, we achieve the two copies of the queue state $\mathsf{Queue}^{\gamma_g}(q, vs', G')$, one needed to re-establish our internal invariant, and one needed to commit the atomic update (we have to give back the queue state to the client). This concludes the successful CAS sub-proof in line 6.

In line 7, we update the queue's tail to point to the newly enqueued node n. We can do this simply by opening the internal invariant $\mathsf{queuel}(\ldots)$ again to access the atomic points-to of the `tail`. We also need a local history-seen observation of $q + \texttt{tail} \sqsupseteq^{\gamma_t}_{\mathbf{sn}} \_$, which we do have from the assumption SeenQueue. In this step we do not update the queue's ghost state, so we can easily re-establish the invariant.

Lastly, before returning, we need to construct $\mathsf{SeenQueue}^{\gamma_g}(q, G', M')$. Fortunately, this step only accumulates all of the observations we have proven previously in this proof. □

Proof sketch of **`try_deq`**$([q])$. The proof of **`try_deq`** is very similar to that of **`try_enq`**, except that the latter works on the `tail`, while the former works on the `head`. In line 1 when reading $q + \mathsf{head}$, we open the internal invariant queuel to access the atomic points-to of $q + \mathsf{head}$ (from ownQueueLocs, see Definition 23.6). We also get a local history-seen observation of $q + \mathsf{head} \sqsupseteq^{\gamma_h}_{\mathbf{sn}} \_$ from the assumption SeenQueue. In this **acq** read, we acquire the observation $\mathsf{seenLink}(\_, \_, \eta_h, h, e_h, v_h)$ for the read sentinel node h. From that observation we can extract $h \sqsupseteq^{\eta_h}_{\mathbf{sy}} [\_ \leftarrow (0, \_)]$, which we need for the read of $h + \texttt{next}$ in line 2 (recall that $\texttt{next} = 0$).

For the read in line 2, again, we need to open the invariant queuel to access the atomic points-to of $h + \texttt{next}$. We know that the atomic points-to is indeed in the invariant (in ownNodes, see Definition 23.4) thanks to the observation $\mathsf{seenLink}(\_, \_, \eta_h, h, e_h, v_h)$ acquired from line 1.

In line 3, we check if the read value n in line 2 is null (0). If it is indeed the case, the caller thread *subjectively* sees the queue as empty. We then commit the atomic update $\langle G, vs.\ \mathsf{Queue}(q, vs, G) \mid b.\ \ldots \Rrightarrow \Phi \rangle^{\top \setminus \mathcal{N}_{\text{QUEUE}}}_{\mathcal{N}_{\text{HIST}}}$ from our context, with the return value $v = \epsilon = 0$ and $vs' = vs$, and $G' = G[d \mapsto (\mathsf{Deq}(\epsilon), V', M')]$. We pick the new $M' = M_0 \cup M^{\mathsf{prev}}_{e_h} \cup \{d\}$, where

- d is a fresh event id for our empty dequeue event; and
- M_0 is the local logical view of the thread when it started the call to **`try_deq`**; and
- $M^{\mathsf{prev}}_{e_h}$ is the set of enqueue events that are enqueued before e_h, *i.e.*, $M^{\mathsf{prev}}_{e_h} = \{e' \in G(e_h).\texttt{logview} \mid G(e').\mathsf{type} = \mathsf{Enq}(\_)\}$.

That is, the empty dequeue d is synchronized with all enqueues that come before (in $\mathcal{L}$) the sentinel node h that the dequeue reads from. The

synchronization comes from (1) all enqueues are synchronized with one another,[8] and (2) the write to the `next` field of a node is a **`rel`** write and the read in line 2 is an **`acq`** read.

This precise pick of M' is crucial to re-establish QUEUE-EMPDEQ-FULL for QueueConsistent(vs', G') (see Figure 23.1): we have to show that for any enqueue event e in $G(d).\mathsf{logview}$[9] is already dequeued and that dequeue d' also happens before d. We can prove this property because

(1) all enqueue events in M' must have come before e_h in $\mathcal{L}$; and

(2) e_h is the enqueue event of (η_h, h) which has been written to $q{+}\mathsf{head}$, so (η_h, h) must have been dequeued, *i.e.*, $(\eta_h, h) \in \mathcal{D}$; and

(3) all dequeues (those in $\mathcal{D}$) are synchronized with one another, due to the use of release-acquire synchronization to $q+\mathsf{head}$ in **`try_deq`**.

Other obligations for QueueConsistent(vs', G') are rather straightforward, so we elide them and conclude this case here.

If n is not null, line 6 uses a **`CAS`** on $q + \mathsf{head}$ to try to make n the new head, effectively dequeueing n from the queue. Here we open the invariant queueI again to access the atomic points-to of $q + \mathsf{head}$. If the **`CAS`** fails, someone already dequeued n, so we commit the atomic update with $v = \bot = -1$ and without any update to the graph G or the abstract state vs, and with $M' = M_0$.

If the **`CAS`** succeeds, we have $\mathsf{ownNode}^{G,\mathcal{T}}(\eta_h, h, \mathsf{Some}(\eta, n, d))$ of the node h where $d =$ **`false`**, *i.e.*, (η, n) has not been dequeued in G. Here, we switch n to be dequeued in $G' = G[d \mapsto (\mathsf{Deq}(v), V', M')]$, which means producing $\mathsf{ownNode}^{G',\mathcal{T}}(\eta_h, h, \mathsf{Some}(\eta, n, \textbf{\texttt{true}}))$. This means that we can take $@_{V'}\ (n + \mathtt{data} \mapsto \_)$ out of the invariant. As the acquire read in line 2 already gives us $\sqsupseteq V'$, later in line 7 we can safely read $n + \mathtt{data}$ non-atomically.

In the proof of line 6 we move (η, n) from the head of $\mathcal{L}_{\mathsf{in}}$ to the tail of $\mathcal{D}$, *i.e.*, $\mathcal{L}_{\mathsf{in}} = [(\eta, n)] +\!\!+ \mathcal{L}'_{\mathsf{in}}$ and $\mathcal{D}' = \mathcal{D} +\!\!+ [(\eta, n)]$. We also commit the atomic update with $G'.\mathsf{so} = \{(e, d)\} \cup G.\mathsf{so}$, where e is the enqueue event of n, and with $M' = M_0 \cup G(e).\mathsf{logview} \cup \{d\}$. That is, the dequeue d is also synchronized with all events that happen before the enqueue e that d is synchronized with.

Then, the gist of re-establishing QueueConsistent(vs', G') is to prove the FIFO property QUEUE-FIFO-FULL for G'. This can be achieved because we track that all enqueues are synchronized with one another, all dequeues are synchronized with one another, and dequeues are pairwise synchronized with enqueues.

Note that in all cases we have to establish SeenQueue$(q, _, M')$. We elide these details and conclude the sketch here. □

Remark 23.10 (Verification against specs as refinement proofs)**.** We note that, as can be seen from the proof sketches, the verification is effectively showing a refinement between the operations of the $\mathtt{LAT}^{\mathsf{abs}}_{\mathsf{hb}}$ specs and the implementations: it proves that the abstract state and the graph are *refined* by the set of physical locations used by the implementation. In these proofs, we not only have to show that we maintain the invariant

[8] see the pick of M' in the successful case of the **`try_enq`** proof

[9] recall that this is the meaning of $(e, d) \in G.\mathsf{lhb}$

of the abstract (*e.g.*, QueueConsistent(vs, G)) as well as the invariant of the concrete (*e.g.*, queuel) but we also have show that every update in the state of the abstract (the commit of the atomic update) is always matched with the corresponding updates of the physical locations of the nodes and the head or the tail.

23.2 Stack Specs and Verification of the Treiber Stack

We present the $\texttt{LAT}_{\mathsf{hb}}$-style specs for stacks, that is, those without an abstract state, in Figure 23.4. These specs are very similar to those of queues: we have two basic assertions, the ownership of the stack state Stack(s, G), which is now only a graph, and the local observation SeenStack(s, G_0, M_0) on some snapshot G_0 and some logical view M_0. The stack consistency conditions are given in Figure 23.5, which are also very similar to those of queues.[10] Similar to queues, the stack can have successful push events (Push(v)), successful pop events (Pop(v)), and empty pop events (Pop(ϵ)).

[10] see Figure 23.1

We briefly explain the specs of stacks.

- HB-NEW-STACK allocates a new stack s with the empty graph $\emptyset$. It returns the initial stack state Stack$(s, \emptyset)$ and the initial observation SeenStack$^{\mathcal{N}_{\text{STACK}}}(q, \emptyset, \emptyset)$. The namespace $\mathcal{N}_{\text{STACK}}$ is picked by the client to host the internal invariants needed by the implementation. Recall that Stack(s, G) is objective, so it can be easily put inside invariants. Just as we have seen in the proof of queues, this can be achieved by implementing Stack(s, G) with just ghost state.

- HB-TRY-PUSH specifies the try version of the **push** function. If the try fails, there is no interesting information about the failure. We note that, for stacks, queues, or any data structure, the "specifier" may also choose to include other events, such as failures due to contention, and specify the conditions on those events accordingly. If the try succeeds, a new push event is added to the current graph G, with appropriate physical view V' and logical view M'.

- HB-TRY-POP says that the try version of the **pop** function can return empty (ϵ), or fails due to contention (returning $\perp$). If the try succeeds, it pops a value in the stack, which must have been pushed and has not been popped yet. An appropriate pop event is then added to the current graph G.

- HB-PUSH and HB-POP just rule out the failure case (due to contention) for a push and a pop, respectively.

- StackConsistent(G) is very similar to the consistency conditions of queues: the only difference is the rule STACK-LIFO which requires that the push and pop order must follow the last-in-first-out principle, stated using the local happens-before lhb relation.

The verification of the Treiber stack implementation in Figure 12.8 against the $\texttt{LAT}_{\mathsf{hb}}$ specs is also very similar to that of the Michael-Scott

$$\mathit{StackEvent} ::= \mathsf{Push}(v) \mid \mathsf{Pop}(v) \mid \mathsf{Pop}(\epsilon)$$
$$\mathit{Event} ::= \mathit{StackEvent} \times \mathit{View} \times \mathit{LogView}$$

HB-NEW-STACK
$$\{\mathsf{True}\}\ \texttt{new\_stack}()\ \{s.\ \mathsf{SeenStack}^{\mathcal{N}_{\text{STACK}}}(q, \emptyset, \emptyset) * \mathsf{Stack}(s, \emptyset)\}$$

HB-TRY-PUSH
$$\mathsf{SeenStack}(s, G_0, M_0) * \sqsupseteq V \vdash \langle G.\ \mathsf{Stack}(s, G)\rangle$$
$$\texttt{try\_push}(s, v)$$
$$\left\langle b.\ \exists G' \sqsupseteq G, M' \sqsupseteq M_0, V' \sqsupseteq V.\ \mathsf{Stack}(s, G') * \mathsf{SeenStack}(s, G', M') * \sqsupseteq V' \right.$$
$$\left. * \vee \begin{cases} b = \texttt{false} & \wedge\ G' = G \\ b = \texttt{true} & \wedge\ \exists e \notin G.\ e \in M' \wedge G' = G[e \mapsto (\mathsf{Push}(v), V', M')] \end{cases} \right\rangle_{\mathcal{N}_{\text{STACK}}}$$

HB-TRY-POP
$$\mathsf{SeenStack}(s, G_0, M_0) * \sqsupseteq V \vdash \langle G.\ \mathsf{Stack}(s, G)\rangle$$
$$\texttt{try\_pop}(s)$$
$$\left\langle v.\ \exists G' \sqsupseteq G, M' \sqsupseteq M_0, V' \sqsupseteq V.\ \mathsf{Stack}(s, G') * \mathsf{SeenStack}(s, G', M') * \sqsupseteq V' \right.$$
$$\left. * \vee \begin{cases} v = \bot & \wedge\ G' = G \\ v = \epsilon & \wedge\ \exists d \notin G.\ d \in M' \wedge G' = G[d \mapsto (\mathsf{Pop}(\epsilon), V', M')] \\ v \notin \{\bot, \epsilon\} & \wedge\ \exists e, V_e, M_e, d \notin G.\ G(e) = (\mathsf{Enq}(v), V_e, M_e) \wedge (e, \_) \notin G.\mathsf{so} \wedge V_e \sqsubseteq V' \\ & \quad \wedge\ M_e \cup \{e, d\} \sqsubseteq M' \wedge G' = G[d \mapsto (\mathsf{Pop}(v), V', M')] \wedge G'.\mathsf{so} = \{(e, d)\} \cup G.\mathsf{so} \end{cases} \right\rangle_{\mathcal{N}_{\text{STACK}}}$$

HB-PUSH
$$\mathsf{SeenStack}(s, G_0, M_0) * \sqsupseteq V \vdash \langle G.\ \mathsf{Stack}(s, G)\rangle$$
$$\texttt{push}(s, v)$$
$$\left\langle ().\ \exists G' \sqsupseteq G, M' \sqsupseteq M_0, V' \sqsupseteq V.\ \mathsf{Stack}(s, G') * \mathsf{SeenStack}(s, G', M') * \sqsupseteq V' \right.$$
$$\left. * \exists e \notin G.\ e \in M' \wedge G' = G[e \mapsto (\mathsf{Push}(v), V', M')] \right\rangle_{\mathcal{N}_{\text{STACK}}}$$

HB-POP
$$\mathsf{SeenStack}(s, G_0, M_0) * \sqsupseteq V \vdash \langle G.\ \mathsf{Stack}(s, G)\rangle$$
$$\texttt{pop}(s)$$
$$\left\langle v.\ \exists G' \sqsupseteq G, M' \sqsupseteq M_0, V' \sqsupseteq V.\ \mathsf{Stack}(s, G') * \mathsf{SeenStack}(s, G', M') * \sqsupseteq V' \right.$$
$$\left. * \vee \begin{cases} v = \epsilon & \wedge\ \exists d \notin G.\ d \in M' \wedge G' = G[d \mapsto (\mathsf{Pop}(\epsilon), V', M')] \\ v \notin \{\bot, \epsilon\} & \wedge\ \exists e, V_e, M_e, d \notin G.\ G(e) = (\mathsf{Enq}(v), V_e, M_e) \wedge (e, \_) \notin G.\mathsf{so} \wedge V_e \sqsubseteq V' \\ & \quad \wedge\ M_e \cup \{e, d\} \sqsubseteq M' \wedge G' = G[d \mapsto (\mathsf{Pop}(v), V', M')] \wedge G'.\mathsf{so} = \{(e, d)\} \cup G.\mathsf{so} \end{cases} \right\rangle_{\mathcal{N}_{\text{STACK}}}$$

HB-STK-OBJ
$$\mathsf{objective}(\mathsf{Stack}(s, G))$$

FIGURE 23.4: $\texttt{LAT}_{\text{hb}}$ specs for stack

HB-STACK-CONSISTENCY

$$\mathsf{Stack}(s, G) \vdash \mathsf{StackConsistent}(G)$$

$$\mathsf{StackConsistent}(G) :=$$

$$\wedge \begin{cases} \forall (e,d) \in G.\mathsf{so}.\, \exists v.\, G(e).\mathtt{type} = \mathsf{Push}(v) \wedge G(d).\mathtt{type} = \mathsf{Pop}(v) \\ \qquad \wedge\, G(e).\mathtt{view} \sqsubseteq G(d).\mathtt{view} \wedge G(e).\mathtt{logview} \sqsubseteq G(d).\mathtt{logview} & \text{(STACK-MATCHES)} \\ G.\mathsf{so} \text{ and } G.\mathsf{so}^{-1} \text{ are functional.} & \text{(STACK-SO-FUNCTIONAL)} \\ \forall d.\, G(d).\mathtt{type} = \mathsf{Pop}(\_) \to (\_, d) \notin G.\mathsf{so} \to G(d).\mathtt{type} = \mathsf{Pop}(\epsilon) & \text{(STACK-UNMATCHED-EMPPOP)} \\ \forall (e,d), (e',d') \in G.\mathsf{so}, (e,e') \in G.\mathsf{lhb} \to (d,d') \in G.\mathsf{lhb} \to (e',d) \notin G.\mathsf{lhb} & \text{(STACK-LIFO)} \\ \forall d, e.\, G(d).\mathtt{type} = \mathsf{Pop}(\epsilon) \to G(e).\mathtt{type} = \mathsf{Push}(\_) \to (e,\_) \notin G.\mathsf{so} \to (e,d) \notin G.\mathsf{lhb} & \text{(STACK-EMPPOP)} \end{cases}$$

FIGURE 23.5: $\mathtt{LAT}_{\mathsf{hb}}$ Stack Consistency

queue (against the $\mathtt{LAT}^{\mathrm{abs}}_{\mathsf{hb}}$ specs). In fact, the proof is simpler because we only work with the head of the stack, while for the queue, we have to coordinate between the head and the tail. As we already presented another proof of the Treiber stack against the simpler bag specs in §12.4.3, we elide the proof here. We note that the $\mathtt{LAT}_{\mathsf{hb}}$ specs presented here imply the bag specs in Figure 12.9.

CHAPTER SUMMARY. In this chapter we have demonstrated concrete Compass specs and verifications through queues and stacks, which, unsurprisingly, show the extra proof obligations required by the stronger logically-atomic specs.

24

Exchangers and the Elimination Stack

In this chapter, we demonstrate the application of our specs to verify an RMC implementation of the elimination stack.[1] We show that the RMC elimination stack satisfies the $\texttt{LAT}_{\texttt{hb}}$ specs for stacks given in §23.2. This verification is both a *client* verification and a *library* verification: the elimination stack is a client that composes an underlying base stack and an *exchanger*. In §24.1 we give an overview of the RMC implementation for the elimination stack. In §24.2 we present the strong specs with *helping* for exchangers. In §24.3 we sketch the verification of the elimination stack as a simulation proof using the specs of the base stack and the exchanger.

[1] Hendler et al., "A scalable lock-free stack algorithm" [HSY04].

24.1 The Elimination Stack

The idea for the elimination stack (ES) comes from a simple observation: if a **push** is immediately followed by a **pop**, then the stack appears unchanged, and that **push** and **pop** are said to *eliminate* each other. The elimination mechanism can be implemented with an exchanger (which in turn can be implemented as an array of exchangers) that supports concurrent exchanges of data with arbitrary matching. A thread simply calls $\texttt{exchange}(x, v_1)$ on the exchanger object x with some value $v_1 \neq \bot$. If the return value is $\bot$, the exchange has failed, but if it is some $v_2 \neq \bot$, then the thread has successfully exchanged v_1 for v_2 with another thread. Additionally, the two threads *synchronize with each other*,[2] which from the separation logic perspective supports *resource exchanges* between the matching threads.

[2] Technically, the two commit points of the matching exchanges are not both in hb with each other—it is counterintuitive to have cycles in hb—but it is the case that the *beginning* of one **exchange** call happens before the *end* of its matching **exchange** call. We needed to extend our specs to account for this subtlety, but, due to space constraints, we elide it from the specs and from the discussion here.

The ES *try* operations, which can fail due to contention, can be implemented simply by composing the two libraries without any extra synchronization, as follows:

$$
\begin{aligned}
\texttt{try\_push}(s, v) ::= {}& \textbf{if}\ \texttt{try\_push}'(s.\mathsf{base}, v)\ \textbf{then}\ \textbf{true} \\
& \textbf{else}\ \texttt{exchange}(s.\mathsf{ex}, v) == \mathsf{SENTINEL} \\
\texttt{try\_pop}(s) ::= {}& \textbf{let}\ v = \texttt{try\_pop}'(s.\mathsf{base})\ \textbf{in} \\
& \textbf{if}\ v\ !{=}\ \mathsf{FAIL\_RACE}\ \textbf{then}\ v\ \textbf{else} \\
& \textbf{let}\ v' = \texttt{exchange}(s.\mathsf{ex}, \mathsf{SENTINEL})\ \textbf{in} \\
& \textbf{if}\ v' \notin \{\mathsf{SENTINEL}, \bot\}\ \textbf{then}\ v' \\
& \textbf{else}\ \mathsf{FAIL\_RACE}
\end{aligned}
$$

Each operation first tries the base stack's corresponding operation, and if that fails due to contention, it tries to use the exchanger to match another operation without going through the base stack. More specifically, **try_push**(s, v) calls the base stack's own **try_push**′ and returns **true** (signifying success) if that succeeds. Otherwise, it calls **exchange** (on s.ex) and returns **true** only if its exchange is successfully matched with a pop operation, signified by the SENTINEL value. Similarly, **try_pop**(s) calls the base stack's **try_pop**′ and returns v only if **try_pop**′ did not fail due to contention (FAIL_RACE). (**try_pop** returns empty ϵ if **try_pop**′ does.) Otherwise, **try_pop** calls **exchange** with SENTINEL, and only succeeds with the returned value v' if it is matched ($v' \neq \bot$) with a push ($v' \neq$ SENTINEL).

24.2 A Strong Spec for Exchangers

An exchanger event carries an additional physical view (`view_in`) for recording the *input view* of the event. It is used for describing the mutual synchronization of matching exchange events (EXCHANGER-MATCHES).

A simplified $\texttt{LAT}_{\text{hb}}$-style spec for the **exchange** function is shown in HB-EXCHANGE (Figure 24.1). The spec involves a local logical view assertion SeenExchanges(x, G_0, M_0), and an atomically shared ownership assertion Exchanger(x, G) for the exchanger object x. At the commit point, the current graph G is extended with a new event e_1 with type Exchange(v_1, v_2), where v_2 is the returned value. If the exchange fails, the return value v_2 is $\bot$ and the event type is Exchange$(v_1, \bot)$. If the exchange succeeds, it can only succeed together with another exchange identified by e_2, and the G.so relation is extended with the two events in both directions ($\{(e_1, e_2), (e_2, e_1)\}$), signifying that they are synchronized with each other.

The remaining part of the spec is to maintain the perspective that *a matching pair of exchanges is committed atomically together*: it is important that there can be *no interference* between the two commits of the matching exchanges. In other words, no other thread should be able to observe an incomplete state of the exchanger where one successful exchange has been committed but its matching exchange has not. But how can *two* commit points be atomic? This conflicts with the intuitive interpretation of LATs that there exists a committing instruction I within each logical operation! To resolve this conundrum, we need *helping*.

HELPING FOR ATOMICITY Helping is a pattern where one operation—the helper—helps to perform the commit (the update to the shared state) of another operation—the helpee. This means that the commit point of the helpee is not within its own execution, but rather within the helper's execution. For the matching exchange pairs, the commit points coincide: at the helper exchange's commit point, it atomically performs the helpee exchange's commit and then its own commit. This is materialized in the successful case of HB-EXCHANGE with (1) a *commit order* ($<$) of the events and (2) the addition of a *local postcondition* (in red, $\{\ldots\}$) that only holds once the function returns (rather than at the commit point).

$$\textit{ExchangeEvent} ::= \mathsf{Exchange}(v_1, v_2)$$

$$\textit{Event} ::= \textit{ExchangeEvent} \times \textit{View} \times \textit{View} \times \textit{LogView}$$

HB-NEW-EXCHANGER

$$\{\mathsf{True}\}\ \mathbf{new\_exchanger}()\ \{x.\ \mathsf{SeenExchanges}^{\mathcal{N}_{\text{XCHG}}}(x, \emptyset, \emptyset) * \mathsf{Exchanger}(x, \emptyset)\}$$

HB-EXCHANGE

$$\mathsf{SeenExchanges}(x, G_0, M_0) * v_1 \neq \bot * \sqsupseteq V_0 \vdash$$

$$\langle b, G.\ \mathsf{Exchanger}(x, G)\rangle$$

$$\mathbf{exchange}(x, v_1)$$

$$\left\langle \begin{array}{l} v_2.\ \exists b', G' \sqsupseteq G, M' \sqsupseteq M_0, V_1 \sqsupseteq V_0, V_{21}, V_{22}, e_1 \notin G, e_2. \\ \quad \mathsf{Exchanger}(x, G') * \sqsupseteq V_1 * e_1 \in M' * G' = G[e_1 \mapsto (\mathsf{Exchange}(v_1, v_2), V_0, V_1, M')] \\ \quad * \bigvee \begin{cases} v_2 = \bot & * \mathsf{SeenExchanges}(x, G', M') * \mathsf{ExchangerConsistent}(G') \\ v_2 \neq \bot \land e_1 < e_2 & * e_2 \in M' * \mathsf{SeenExchanges}(x, G', M' \setminus \{e_2\}) \\ v_2 \neq \bot \land e_2 < e_1 & * e_2 \in M' * \mathsf{SeenExchanges}(x, G', M') * \mathsf{ExchangerConsistent}(G') \\ & * G(e_2) = (\mathsf{Exchange}(v_2, v_1), V_{21}, V_{22}, M') * G'.\mathsf{so} = \{(e_1, e_2), (e_2, e_1)\} \cup G.\mathsf{so} \end{cases} \end{array} \right\rangle_{\mathcal{N}_{\text{XCHG}}}$$

$$\left\{ \begin{array}{l} (v_2 \neq \bot \land e_1 < e_2) \implies \exists G'' \sqsupseteq G'.\ \mathsf{SeenExchanges}(x, G'', M') * \mathsf{ExchangerConsistent}(G'') \\ \qquad * G''(e_2) = (\mathsf{Exchange}(v_2, v_1), V_{21}, V_{22}, M') * G''.\mathsf{so} = \{(e_1, e_2), (e_2, e_1)\} \cup G'.\mathsf{so} \end{array} \right\}$$

HB-EXCHANGER-CONSISTENCY

$$\mathsf{Exchanger}(x, G) \twoheadrightarrow \mathsf{SeenExchanges}(x, G', M') \Rrightarrow_{\mathcal{N}_{\text{XCHG}}} \mathsf{Exchanger}(x, G) * \mathsf{ExchangerConsistent}(G)$$

$$\mathsf{ExchangerConsistent}(G) :=$$

$$\bigwedge \begin{cases} \forall (e_1, e_2) \in G.\mathsf{so}.\ |e_1 - e_2| = 1 \\ \quad \land \exists v_1 v_2.\ G(e_1).\mathtt{type} = \mathsf{Exchange}(v_1, v_2) \land G(e_2).\mathtt{type} = \mathsf{Exchange}(v_2, v_1) \\ \quad \land G(e_1).\mathtt{view\_in} \sqsubseteq G(e_2).\mathtt{view} \land G(e_2).\mathtt{view\_in} \sqsubseteq G(e_1).\mathtt{view} & \text{(EXCHANGER-MATCHES)} \\ G.\mathsf{so} \text{ is symmetric and irreflexive.} & \text{(EXCHANGER-SO-SYM-IRREFL)} \\ G.\mathsf{so} \text{ and } G.\mathsf{so}^{-1} \text{ are functional.} & \text{(EXCHANGER-SO-FUNCTIONAL)} \\ \forall e.\ (e, \_) \notin G.\mathsf{so} \to G(e).\mathtt{type} = \mathsf{Exchange}(\_, \bot) & \text{(EXCHANGER-UNMATCHED)} \end{cases}$$

FIGURE 24.1: $\mathtt{LAT}_{\mathsf{hb}}$ specs for exchangers (excerpt, simplified).

The commit order $<$ on events is the logical order in which the events are committed to the shared graph G. In HB-EXCHANGE, the commit order between a matching exchange pair dictates who the helper is, and how each commit updates the shared graph G. If the *current* exchange e_1 is committed before the *other* exchange e_2, *i.e.*, $e_1 < e_2$, then e_2 is the helper. Otherwise, if $e_2 < e_1$, then e_1 is the helper.

Since the helper atomically performs the helpee's update and then its own update, it always knows the result of the helpee's update, while the helpee will only learn about the helper's update *after* both commits have been completed. This is the asymmetry in HB-EXCHANGE: if e_1 is the helpee, it only adds itself to the current graph G: $G' = G[e_1 \mapsto (\mathsf{Exchange}(v_1, v_2), V_1, M')]$; but if e_1 is the helper, it knows that the helpee's event e_2 must already be in the current graph: $G(e_2) = (\mathsf{Exchange}(v_2, v_1), V_2, M')$, and the helper not only adds itself to the current graph G, but also extends $G.\mathsf{so}$ with the pairs $\{(e_1, e_2), (e_2, e_1)\}$. The client thread of the helper learns all of this information about the updated G' atomically right after the helper's commit, by which point it has also locally observed both e_1 and e_2, via $\mathsf{SeenExchanges}(x, G', M')$ and $\{e_1, e_2\} \subseteq M'$. The client thread of the helpee, on the other hand, right after its own commit has only locally observed its own event e_1, via $\mathsf{SeenExchanges}(x, G', M' \setminus \{e_2\})$, because the helper commit has not been performed yet and e_2 has not been added to G'. Only in the local postcondition (in $\{\dots\}$), after both commits have been performed, can the helpee learn about the new graph G'' (e_2's G') that completes e_1's G' (e_2's G) with e_2 and the so pairs, and locally observe both events, via $\mathsf{SeenExchanges}(x, G'', M')$.

INTERMEDIATE STATES That matching exchange pairs are committed atomically together is also reflected by the fact that we do not always have consistency: the ownership $\mathsf{Exchanger}(x, G)$ does *not* imply $\mathsf{ExchangerConsistent}(G)$. Instead, we have $\mathsf{ExchangerConsistent}(G')$ only with a completed graph G', *i.e.*, after the failure case or after the helper's commit. Between the helpee's commit and the helper's commit, the exchanger is in an incomplete intermediate state.

As such, those intermediate states can appear in a client invariant. However, it is important that the client needs to handle such states *only when* it uses the exchanger, and that other non-exchanger-related operations will never observe those states. For example, the invariant of the elimination stack needs to consider the intermediate state where a push event created by a successful exchange is inserted into the graph, but the matching pop event by the matching exchange is not. A successful push using the base stack and running concurrently with the exchange pair should *not* observe the client invariant in such an intermediate state, because it would not be able to prove LIFO then.

Our full exchanger spec (in Coq) supports this form of intermediate state reasoning: when using the exchanger, the client need not maintain its invariant for the intermediate state between the two commits; it only needs to re-establish its invariant after both commits. When *not* using the exchanger, the client invariant is never in such intermediate states.

STRENGTH OF THE SPECS To the best of our knowledge, the full exchanger spec is the first ever proposed CSL spec for RMC exchangers. It is strong enough for the proof of the elimination stack (§24.1), and we have also used it to derive a spec that supports resource exchanges, where each **exchange** call needs to provide the resources to be exchanged only at its commit point, and only if the exchange succeeds.

24.3 Verifying the Elimination Stack

VERIFICATION RESULTS. Assuming the $\texttt{LAT}_{\texttt{hb}}$-style specs for the base stack and the exchanger (Figure 24.1), we have verified that our relaxed ES implementation satisfies the same $\texttt{LAT}_{\texttt{hb}}$ specs as the base stack. Since we have verified, *separately*, that our RMC Treiber stack and exchanger implementations satisfy their $\texttt{LAT}_{\texttt{hb}}$ specs, we easily get a closed-proof verification of an ES built from those two implementations. We assumed the $\texttt{LAT}_{\texttt{hb}}$ specs for the base stack to demonstrate that the verification does not rely on very strong properties, but, since the proofs are modular enough, we conjecture that the same proofs can be applied (with minor modifications) to show that, if the base stack satisfies the stronger $\texttt{LAT}^{\text{abs}}_{\texttt{hb}}$ specs, then the ES implementation also satisfies the same stronger specs.

COMPOSITIONAL VERIFICATION. The ES verification does not involve RMC reasoning, because the implementation does not add any new atomic instructions. The core work is composing the base stack's events and the exchanger's events into the ES events in a way that satisfies the consistency conditions for stacks. This can be seen as a simulation proof with a simple simulation relation: every base stack operation is simulated by a corresponding ES operation, and successful matching exchange pairs between a non-SENTINEL value and a SENTINEL value are simulated by an ES push and an ES pop respectively. (Other exchange events are ignored by the simulation.)

The non-trivial parts of the proof are where the simulation needs to simulate commit points and maintain consistency: whenever a base operation commits, the ES operation needs to commit accordingly, and needs to re-establish the ES consistency conditions using the consistency conditions of the base stack and the exchanger. The re-establishment of consistency relies crucially on the fact that eliminations are *atomic*: the commits of ES push and pop events that originate from a pair of matching exchanges need to be performed together at once, so that the pushed element is popped immediately, and no (commit points of) other concurrent ES operations can observe the intermediate state where the ES push has already been committed but the ES pop has not. This atomicity property of the exchange-based ES event pairs is crucially needed for LIFO.

25

Related Work

Our specification styles build on extensive prior work in relaxed correctness conditions, and in program logics for fine-grained concurrent SC and RMC programs.

RELAXED CORRECTNESS CONDITIONS Various alternative correctness conditions to linearizability have been developed,[1] particularly for distributed systems[2] and relaxed memory.[3] Most of these were developed outside a program logic, directly on complex low-level concurrency semantics, and with little support for client reasoning or mechanization. As discussed in §1.3, we believe the Yacovet approach[4] is the most general of these. By enhancing Yacovet specs in Compass with logical atomicity, we demonstrate that existing relaxed correctness conditions can be used in combination with separation logic to achieve stronger and better modular client reasoning as well as more foundational (mechanized) verifications. We consider it future work to encode more of these relaxed correctness conditions in Compass.

[1] [Hen+13; JR14; Der+14; Haa+16; ON22].

[2] [Nei94; AKY10; Bur+14; CRR15].

[3] [Bur+12; BDG13; Jag+13; Doh+18; Don+18; Raa+19; EE19; Kri+20].

[4] Raad et al., "On library correctness under weak memory consistency: specifying and verifying concurrent libraries under declarative consistency models" [Raa+19].

Recently, Singh and Lahav[5] use abstract programs as specifications for RMC libraries, and define the correctness condition as contextual refinement[6] between the implementation and the specification programs. To be sound, the refinement is defined as trace refinement between histories (event graphs) who additionally track the propagation of method calls and returns among thread (propagation of observations of library actions, similar to that of Compass' logical views). It is interesting to encode such a correctness condition in CSL, which corresponds to showing contextual refinements for RMC programs in separation logic.

[5] Singh and Lahav, "An Operational Approach to Library Abstraction under Relaxed Memory Concurrency" [SL23].

[6] where the context must adhere to the library's calling policy

SC PROGRAM LOGICS Logical atomicity is just one CSL alternative to linearizability. Another is to avoid identifying commit points and instead reason directly about *refinements* between a sequential "specification" program and the concurrent implementation program.[7] However, sequential specs are not always suitable as correctness conditions (*e.g.*, for exchangers), and *non-sequential* refinement is still an open problem for RMC logics. Our work demonstrates the usefulness of logical atomicity in RMC. As future work, one can consider adapting *prophecy variables*[8] to our framework, as they may help simplify our specs.

[7] [LF13; TDB13; FKB18; KSB17].

[8] [AL88; AL91; Jun+20].

FCSL[9] and the rely-guarantee-based Hoare logics by Hemed et al.[10] and Khyzha et al.[11] support specifying non-linearizable SC data structures

[9] [SNB15; Ser+16; Del+17].

[10] [HRV15].

[11] [Khy+17].

with histories by encoding histories as auxiliary (ghost) state and by exposing partial, subjective views of the histories to clients. This is similar to our construction of graphs or linear histories. Compass can be seen as extending these logics with logical atomicity and RMC.

RMC PROGRAM LOGICS Dalvandi and Dongol,[12] in parallel work, try to achieve the same goal of providing compositional specs and modular client reasoning for RMC data structures. Their approach uses an Owicki-Gries-style Hoare logic[13] for a more limited fragment of RC11 called RAR (which only has release-acquire and relaxed accesses, not non-atomics or fences). They specify libraries with view-based, atomic *abstract object semantics* for the library's operations, treating the object methods as primitives of the language. Client verifications rely on Hoare-triple specs derived directly from the abstract object semantics. To verify an implementation against a spec, they prove refinement, showing that synchronizations (in the view semantics) of the abstract object are simulated by synchronizations in the implementation. Their approach therefore shares similar ideas to ours. The main limitation is in their simulation method: it applies only to *synchronization-free* clients, *i.e.*, those who synchronize only through the library in question. This is because it is non-trivial to characterize how external synchronizations affect the simulation relation. Consequently, they cannot obtain an end-to-end proof for clients that use external synchronizations, *e.g.*, the MP client in Figure 21.2. Furthermore, the use of Owicki-Gries-style logic means that they have to deal with additional *interference freedom* proofs. They report only one mechanized library verification, for the Treiber stack, with 12KLOC in Isabelle. In comparison, our mechanization results are more extensive, and our Treiber stack verification takes only 2.2KLOC in Iris, in Coq.

Several CSLs for RMC have been developed within Iris.[14] Compass extends iRC11 and follows Cosmo in exposing more view information in specs. Our key innovation is the use of logical views on library operations, allowing us to give stronger specifications that can describe interactions with external synchronization. In retrospect, we believe that views are a concise, compositional, and user-friendly tool to describe the different kinds of synchronization that may occur in and around a data structure, and thus are useful for formulating full functional correctness specs under RMC.

Very recently, Park et al.[15] improve RMC linearizability proofs with Compass-style specifications by (1) introducing a general *object modification order* (omo) for libraries, (2) introducing a general *commit-with* relation to help with the simulation relation between commit points of the specification and the implementation, and (3) applying proof automation with Diaframe[16] to infer the logical event graph state. The result is that, in some cases, proofs are significantly shortened by an order of magnitude. This demonstrates the extensibility of Diaframe, and signifies the benefits of more general relations as well as automation in scaling RMC verification efforts further—a topic that this thesis did not discuss.

Finally, it would be interesting to apply the Compass approach to

[12] [DD21b; DD21a].

[13] Dalvandi et al., "Owicki-Gries Reasoning for C11 RAR" [Dal+20].

[14] [Kai+17; Dan+20a; MJP20].

[15] Park et al., "A Proof Recipe for Linearizability in Relaxed Memory Separation Logic (to appear in PLDI 2024)" [Par+24].

[16] Mulder et al., "Diaframe: automated verification of fine-grained concurrent programs in Iris" [MKG22]; Mulder and Krebbers, "Proof Automation for Linearizability in Separation Logic" [MK23].

more sophisticated RMC libraries such as work-stealing queues[17] and safe memory reclamation schemes for lock-free data structures.[18]

[17] Chase and Lev, "Dynamic circular work-stealing deque" [CL05]; Lê et al., "Correct and efficient work-stealing for weak memory models" [Lê+13].

[18] Michael, "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects" [Mic04]; Fraser, "Practical lock-freedom" [Fra04]; Jung et al., "Modular Verification of Safe Memory Reclamation in Concurrent Separation Logic" [Jun+23].

26

Conclusion

This thesis tries to sketch a recipe to formally build and apply separation logics for verifying safety and functional correctness of programs with relaxed memory concurrency (RMC). Undoubtedly, building and applying such logics require substantial efforts, and are composed of several steps, which can partly be seen from the thesis structure in Figure 1.1. These steps can be grouped into the following 4 general tasks.

(1) selecting a trusted semantic model of RMC, combined with other language features;

(2) constructing a low-level logic that provides core separation reasoning principles;

(3) deriving high-level reasoning principles that may be application-specific;

(4) applying the high-level logic to the target programs.

SELECTING A SEMANTIC MODEL OF RMC. Multiple RMC semantic models have been proposed, in both axiomatic and operational style. We proposed to use an operational version of RC11, called ORC11, in Chapter 3, where we also include non-atomics whose sematics can invoke undefined behaviors (UBs). Triggering UBs is enforced by a race detector on non-atomics. In Chapter 4, we incorporated the ORC11 memory model with the semantics of λ_{Rust}, which has other pure expressions, and which also includes a conservative pointer comparison scheme. The pure fragment of language can trigger more UBs (such as incompatible comparison).

The integration of a race detector and a conservative pointer comparison scheme shows a serious challenge in formal verifications with realistic languages. Modern languages have become so complicated that individual research works on formal semantics have mostly been contended with only *fragments* of a language. It is unclear if we can keep the semantics of those fragments modularly unentangled, or how much work would be needed to compose their semantics so that we can consider programs that exploit multiple language features all at once. In Chapter 4, we saw that the interaction of our conservative pointer comparison scheme and the RMC semantics resulted in a rather global semantic of **CAS** (RMW) operations, whose complication also bubbles up into the logics.

We have chosen to use the operational-style semantics of ORC11, and in §3.6 only provided an on-paper correspondence proof between RC11 and ORC11 + λ_{Rust}, partly because the resulting semantics encompasses more features and more UB-triggering behaviors. But the main driver of such decision is that it is easier to build separation logics with ORC11 (as well as other views-based semantics), because views restructure the semantic in a more modular way (an effect focuses on a thread's view) and therefore are easier to "separated". Nevertheless, it should still be possible to build separation logics in Iris using an operational model that is much closer to the axiomatic model than ORC11. We consider it future work to build separation logics in Iris directly on the Operational Graph Semantics (OGS) mentioned in §3.6, which means putting a substantial part of the correspondence proof inside Iris, and thus cutting down the TCB.

Finally, very recent work by Lee et al.[1] suggests that one could forbid load-buffering (LB), and therefore the out-of-thin-air (OOTA) problem, at least in the source language memory model.[2] That would mean that ORC11 would be more suitable to model C and C++ RMC programs.

[1] Lee et al., "Putting Weak Memory in Order via a Promising Intermediate Representation" [Lee+23].

[2] see also Remark 2.21

CONSTRUCTING A CORE LOGIC. With an operational RMC model, we then can start constructing the core separation logic iRC11 for our RMC semantics. In Chapter 7, we demonstrated how to use Iris to acquire a base logic that provides basic separation on the RMC states. Most importantly, we introduced thread-local assertions such as the seen thread-view observation $\mathsf{Seen}(\mathcal{V})$ to account for views, the *thread-local state* of our RMC semantics. Then, in Chapter 8, we introduced a way to hide views in the logic, because most of the time when performing *intra*-thread reasoning, we do not need to care about views. The interaction with views only becomes interesting when we have to reason about synchronizations between threads, *i.e.*, when we need *inter*-thread reasoning. For inter-thread reasoning, we developed several modalities that allow us to temporarily expose views again, which we can then use with general invariants (Chapter 11) to establish inter-thread communications or resource transfer. We note that this approach is inspired by the Cosmo logic,[3] and is applicable not only to views-based semantics,[4] but can also be generalized to handle other *thread-local states*, such as call stacks or physical-virtual address translations.

[3] Mével et al., "Cosmo: a concurrent separation logic for multicore OCaml" [MJP20].

[4] for example, see a separation logic for non-volatile memory by Vindum and Birkedal ([VB23]).

Finally, respectively in Chapter 9 and Chapter 10, we have developed abstractions for non-atomic and atomic accesses. The atomic points-to is general enough to support most usage patterns of atomics, and is also convertible to non-atomic points-to. This setup therefore should be applicable to stronger memory models. The key feature of the atomic points-to is that it interacts well with general invariants (with explicit views), which makes it easier to state protocols spanning multiple locations.

DERIVING HIGH-LEVEL REASONING PRINCIPLES. With a solid set of fundamental low-level abstractions, we started building higher-level abstractions that can be application-specific. In Chapter 16, we have integrated

the SC model of the lifetime logic with our new abstractions to make the lifetime logic sound in RMC. Most RMC effects can be hidden, except for atomic borrows, where we have interactions with explicit views, in the very same fashion as in general invariants. In Chapter 17, we have combined atomic points-to with general invariants to reconstruct GPS protocols in iRC11. In Chapter 22, we have combined atomic points-to, general invariants, and logical atomicity to specify and verify strong functional correctness for RMC libraries.

Applying the high-level logic to verifications. Ultimately, the designs of low-level and high-level reasoning principles were driven by the goal to verify strong specifications of concrete, realistic RMC programs. In Chapter 12, we have demonstrated several verifications using the core logic of iRC11. In Chapter 18 and Chapter 19, we have demonstrated the semantic type-checking of `RwLock` and `Arc` using the lifetime logic and GPS protocols. In Chapter 23 and Chapter 24, we have demonstrated the verifications of stacks, queues, and exchangers against strong Compass specifications. The various applications together have demonstrated the generality and extensibility of our separation logic iRC11.

Bibliography

[AL88] Martín Abadi and Leslie Lamport. “The Existence of Refinement Mappings”. In: *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS ’88), Edinburgh, Scotland, UK, July 5-8, 1988*. IEEE Computer Society, 1988, pp. 165–175. DOI: 10.1109/LICS.1988.5115. URL: https://doi.org/10.1109/LICS.1988.5115 (cit. on p. 297).

[AL91] Martín Abadi and Leslie Lamport. “The Existence of Refinement Mappings”. In: *Theor. Comput. Sci.* 82.2 (1991), pp. 253–284. DOI: 10.1016/0304-3975(91)90224-P. URL: https://doi.org/10.1016/0304-3975(91)90224-P (cit. on p. 297).

[AKY10] Yehuda Afek, Guy Korland, and Eitan Yanovsky. “Quasi-Linearizability: Relaxed Consistency for Improved Concurrency”. In: *Principles of Distributed Systems - 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings*. Vol. 6490. Lecture Notes in Computer Science. Springer, 2010, pp. 395–410. DOI: 10.1007/978-3-642-17653-1_29. URL: https://doi.org/10.1007/978-3-642-17653-1_29 (cit. on pp. 5, 297).

[Ahm+10] Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. “Semantic foundations for typed assembly languages”. In: *ACM Trans. Program. Lang. Syst.* 32.3 (2010), 7:1–7:67. DOI: 10.1145/1709093.1709094. URL: https://doi.org/10.1145/1709093.1709094 (cit. on p. 173).

[BDG13] Mark Batty, Mike Dodds, and Alexey Gotsman. “Library abstraction for C/C++ concurrency”. In: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’13, Rome, Italy - January 23 - 25, 2013*. ACM, 2013, pp. 235–248. DOI: 10.1145/2429069.2429099. URL: https://doi.org/10.1145/2429069.2429099 (cit. on pp. 5, 297).

[Bat+11] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. “Mathematizing C++ concurrency”. In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. ACM, 2011, pp. 55–66. DOI: 10.1145/1926385.1926394. URL: https://doi.org/10.1145/1926385.1926394 (cit. on pp. 2, 13, 15, 23).

[BP19] John Bender and Jens Palsberg. “A formalization of Java’s concurrent access modes”. In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019), 142:1–142:28. DOI: 10.1145/3360568. URL: https://doi.org/10.1145/3360568 (cit. on p. 2).

[BBW19] Frédéric Besson, Sandrine Blazy, and Pierre Wilke. “CompCertS: A Memory-Aware Verified C Compiler Using a Pointer as Integer Semantics”. In: *J. Autom. Reason.* 63.2 (2019), pp. 369–392. DOI: 10.1007/s10817-018-9496-y. URL: https://doi.org/10.1007/s10817-018-9496-y (cit. on p. 51).

[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. URL: https://doi.org/10.1145/3473586 (cit. on p. 260).

[BD14] Hans-Juergen Boehm and Brian Demsky. “Outlawing ghosts: avoiding out-of-thin-air results”. In: *Proceedings of the workshop on Memory Systems Performance and Correctness, MSPC ’14, Edinburgh, United Kingdom, June 13, 2014*. Ed. by Jeremy Singer, Milind Kulkarni, and Tim Harris. ACM, 2014, 7:1–7:6. DOI: 10.1145/2618128.2618134. URL: https://doi.org/10.1145/2618128.2618134 (cit. on p. 22).

[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. DOI: 10.1145/1040305.1040327. URL: https://doi.org/10.1145/1040305.1040327 (cit. on p. 241).

[Boy03] John Boyland. “Checking interference with fractional permissions”. In: *SAS*. LNCS. 2003. DOI: 10.1007/3-540-44898-5_4 (cit. on pp. 81, 239).

[Bro07] Stephen Brookes. “A Semantics for Concurrent Separation Logic”. In: *Theoretical Computer Science* 375.1–3 (2007). DOI: 10.1016/j.tcs.2006.12.034 (cit. on p. 1).

[Bur+12] Sebastian Burckhardt, Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. “Concurrent Library Correctness on the TSO Memory Model”. In: *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 2012, pp. 87–107. DOI: 10.1007/978-3-642-28869-2_5. URL: https://doi.org/10.1007/978-3-642-28869-2_5 (cit. on pp. 5, 297).

[Bur+14] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. “Replicated data types: specification, verification, optimality”. In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’14, San Diego, CA, USA, January 20-21, 2014*. ACM, 2014, pp. 271–284. DOI: 10.1145/2535838.2535848. URL: https://doi.org/10.1145/2535838.2535848 (cit. on pp. 5, 297).

[CRR15] Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. “Specifying Concurrent Problems: Beyond Linearizability and up to Tasks - (Extended Abstract)”. In: *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*. Vol. 9363. Lecture Notes in Computer Science. Springer, 2015, pp. 420–435. DOI: 10.1007/978-3-662-48653-5_28. URL: https://doi.org/10.1007/978-3-662-48653-5_28 (cit. on pp. 5, 297).

[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: *OSDI*. USENIX Association, 2021, pp. 423–439 (cit. on p. 1).

[CV19] Soham Chakraborty and Viktor Vafeiadis. “Grounding thin-air reads with event structures”. In: *Proc. ACM Program. Lang.* 3.POPL (2019), 70:1–70:28. DOI: 10.1145/3290383. URL: https://doi.org/10.1145/3290383 (cit. on pp. 2, 24).

[CL05] David Chase and Yossi Lev. “Dynamic circular work-stealing deque”. In: *SPAA 2005: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures, July 18-20, 2005, Las Vegas, Nevada, USA*. ACM, 2005, pp. 21–28. DOI: 10.1145/1073970.1073974. URL: https://doi.org/10.1145/1073970.1073974 (cit. on pp. 168, 299).

[Cho+21a] 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: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, 16–31. ISBN: 9781450383912. DOI: 10.1145/3453483.3454027. URL: https://doi.org/10.1145/3453483.3454027 (cit. on p. 58).

[Cho+21b] Minki Cho, Sung-Hwan Lee, Chung-Kil Hur, and Ori Lahav. “Modular data-race-freedom guarantees in the promising semantics”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, 867–882. ISBN: 9781450383912. DOI: 10.1145/3453483.3454082. URL: https://doi.org/10.1145/3453483.3454082 (cit. on pp. 2, 57).

[Cho+22] Minki Cho, Sung-Hwan Lee, Dongjae Lee, Chung-Kil Hur, and Ori Lahav. “Sequential reasoning for optimizing compilers under weak memory concurrency”. In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, 213–228. ISBN: 9781450392655. DOI: 10.1145/3519939.3523718. URL: https://doi.org/10.1145/3519939.3523718 (cit. on pp. 2, 57, 58).

[Www] *Crossbeam: Tools for concurrent programming in Rust*. Available at https://github.com/crossbeam-rs/crossbeam. 2021 (cit. on p. 168).

[Dal+20] Sadegh Dalvandi, Simon Doherty, Brijesh Dongol, and Heike Wehrheim. “Owicki-Gries Reasoning for C11 RAR”. In: *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*. Vol. 166. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 11:1–11:26. DOI: 10.4230/LIPIcs.ECOOP.2020.11. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2020.11 (cit. on p. 298).

[DD21a] Sadegh Dalvandi and Brijesh Dongol. “Verifying C11-Style Weak Memory Libraries via Refinement”. In: *CoRR* abs/2108.06944 (2021). arXiv: 2108.06944. URL: https://arxiv.org/abs/2108.06944 (cit. on p. 298).

[DD21b] Sadegh Dalvandi and Brijesh Dongol. “Verifying C11-style weak memory libraries”. In: *PPoPP ’21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*. ACM, 2021, pp. 451–453. DOI: 10.1145/3437801.3441619. URL: https://doi.org/10.1145/3437801.3441619 (cit. on p. 298).

[Dan+20a] 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. URL: https://doi.org/10.1145/3371102 (cit. on pp. 1, 3, 9, 136, 298).

[Dan+20b] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. *Technical Appendix: RustBelt Meets Relaxed Memory*. 2020. URL: https://plv.mpi-sws.org/rustbelt/rbrlx/appendix.pdf (cit. on pp. 44, 45).

[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. DOI: 10.1145/3519939.3523451. URL: https://doi.org/10.1145/3519939.3523451 (cit. on pp. 3, 9, 10).

[Del+17] Germán Andrés Delbianco, Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. "Concurrent Data Structures Linked in Time". In: *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*. Vol. 74. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 8:1–8:30. DOI: 10.4230/LIPIcs.ECOOP.2017.8. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2017.8 (cit. on p. 297).

[Der+14] John Derrick, Brijesh Dongol, Gerhard Schellhorn, Bogdan Tofan, Oleg Travkin, and Heike Wehrheim. "Quiescent Consistency: Defining and Verifying Relaxed Linearizability". In: *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*. Vol. 8442. Lecture Notes in Computer Science. Springer, 2014, pp. 200–214. DOI: 10.1007/978-3-319-06410-9_15. URL: https://doi.org/10.1007/978-3-319-06410-9_15 (cit. on pp. 5, 297).

[DY+10] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. "Concurrent abstract predicates". In: *ECOOP*. LNCS. 2010. DOI: 10.1007/978-3-642-14107-2_24 (cit. on p. 1).

[Doh+18] Simon Doherty, Brijesh Dongol, Heike Wehrheim, and John Derrick. "Making Linearizability Compositional for Partially Ordered Executions". In: *Integrated Formal Methods - 14th International Conference, IFM 2018, Maynooth, Ireland, September 5-7, 2018, Proceedings*. Vol. 11023. Lecture Notes in Computer Science. Springer, 2018, pp. 110–129. DOI: 10.1007/978-3-319-98938-9_7. URL: https://doi.org/10.1007/978-3-319-98938-9_7 (cit. on pp. 5, 297).

[Doh+19] Simon Doherty, Brijesh Dongol, Heike Wehrheim, and John Derrick. "Verifying C11 programs operationally". In: *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*. Ed. by Jeffrey K. Hollingsworth and Idit Keidar. ACM, 2019, pp. 355–365. DOI: 10.1145/3293883.3295702. URL: https://doi.org/10.1145/3293883.3295702 (cit. on p. 57).

[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*. Vol. 9583. Lecture Notes in Computer Science. Springer, 2016, pp. 413–430. DOI: 10.1007/978-3-662-49122-5_20. URL: https://doi.org/10.1007/978-3-662-49122-5_20 (cit. on pp. 1, 2, 24, 25, 61, 99, 102, 124, 133, 167).

[DV17] Marko Doko and Viktor Vafeiadis. "Tackling Real-Life Relaxed Concurrency with FSL++". 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*. Vol. 10201. Lecture Notes in Computer Science. Springer, 2017, pp. 448–475. DOI: 10.1007/978-3-662-54434-1_17. URL: https://doi.org/10.1007/978-3-662-54434-1_17 (cit. on pp. 1, 2, 5, 24, 99, 102, 124, 133, 167, 251).

[DSM18] Stephen Dolan, K. C. Sivaramakrishnan, and Anil Madhavapeddy. "Bounding data races in space and time". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. ACM, 2018, pp. 242–255. DOI: 10.1145/3192366.3192421. URL: https://doi.org/10.1145/3192366.3192421 (cit. on pp. 2, 6).

[Don+18] Brijesh Dongol, Radha Jagadeesan, James Riely, and Alasdair Armstrong. "On abstraction and compositionality for weak-memory linearisability". In: *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings*. Vol. 10747. Lecture Notes in Computer Science. Springer, 2018, pp. 183–204. DOI: 10.1007/978-3-319-73721-8_9. URL: https://doi.org/10.1007/978-3-319-73721-8_9 (cit. on pp. 5, 297).

[EE19] Michael Emmi and Constantin Enea. “Weak-consistency specification via visibility relaxation”. In: *Proc. ACM Program. Lang.* 3.POPL (2019), 60:1–60:28. DOI: 10.1145/3290373. URL: https://doi.org/10.1145/3290373 (cit. on pp. 5, 297).

[Fen09] Xinyu Feng. “Local Rely-Guarantee Reasoning”. In: *POPL*. 2009. DOI: 10.1145/1594834.1480922 (cit. on p. 1).

[FFS07] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. “On the Relationship Between Concurrent Separation Logic and Assume-Guarantee Reasoning”. In: *ESOP*. 2007 (cit. on p. 1).

[Flu+17] Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. “Mixed-size concurrency: ARM, POWER, C/C++11, and SC”. 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. 429–442. DOI: 10.1145/3009837.3009839. URL: https://doi.org/10.1145/3009837.3009839 (cit. on pp. 2, 58).

[Fra04] Keir Fraser. “Practical lock-freedom”. PhD thesis. University of Cambridge, 2004 (cit. on p. 299).

[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*. ACM, 2018, pp. 442–451. DOI: 10.1145/3209108.3209174. URL: https://doi.org/10.1145/3209108.3209174 (cit. on p. 297).

[FKB21] Dan Frumin, Robbert Krebbers, and Lars Birkedal. “ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity”. In: *Logical Methods in Computer Science* Volume 17, Issue 3 (July 2021). DOI: 10.46298/lmcs-17(3:9)2021. URL: https://lmcs.episciences.org/7708 (cit. on p. 1).

[Fu+10] Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. “Reasoning about Optimistic Concurrency Using a Program Logic for History”. In: *CONCUR*. 2010 (cit. on p. 1).

[G+22] Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. “Simuliris: A Separation Logic Framework for Verifying Concurrent Program Optimizations”. In: *POPL*. 2022 (cit. on p. 1).

[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*. Ed. by Zhong Shao. Vol. 4807. Lecture Notes in Computer Science. Springer, 2007, pp. 19–37. DOI: 10.1007/978-3-540-76637-7_3. URL: https://doi.org/10.1007/978-3-540-76637-7_3 (cit. on pp. 167, 175).

[Haa+16] Andreas Haas, Thomas A. Henzinger, Andreas Holzer, Christoph M. Kirsch, Michael Lippautz, Hannes Payer, Ali Sezgin, Ana Sokolova, and Helmut Veith. “Local Linearizability for Concurrent Container-Type Data Structures”. In: *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada*. Vol. 59. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, 6:1–6:15. DOI: 10.4230/LIPIcs.CONCUR.2016.6. URL: https://doi.org/10.4230/LIPIcs.CONCUR.2016.6 (cit. on pp. 5, 297).

[Ham+24] Angus Hammond, Zongyuan Liu, Thibaut Pérami, Peter Sewell, Lars Birkedal, and Jean Pichon-Pharabod. “An Axiomatic Basis for Computer Programming on the Relaxed Arm-A Architecture: The AxSL Logic”. In: *Proc. ACM Program. Lang.* 8.POPL (2024). DOI: 10.1145/3632863. URL: https://doi.org/10.1145/3632863 (cit. on p. 168).

[He+18] Mengda He, Viktor Vafeiadis, Shengchao Qin, and João F. Ferreira. “GPS+: Reasoning About Fences and Relaxed Atomics”. In: *Int. J. Parallel Program.* 46.6 (2018), pp. 1157–1183. DOI: 10.1007/s10766-017-0518-x. URL: https://doi.org/10.1007/s10766-017-0518-x (cit. on pp. 1, 133, 167).

[HRV15] Nir Hemed, Noam Rinetzky, and Viktor Vafeiadis. “Modular Verification of Concurrency-Aware Linearizability”. In: *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*. Vol. 9363. Lecture Notes in Computer Science. Springer, 2015, pp. 371–387. DOI: 10.1007/978-3-662-48653-5_25. URL: https://doi.org/10.1007/978-3-662-48653-5_25 (cit. on pp. 5, 297).

[HSY04] Danny Hendler, Nir Shavit, and Lena Yerushalmi. “A scalable lock-free stack algorithm”. In: *SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June 27-30, 2004, Barcelona, Spain*. ACM, 2004, pp. 206–215. DOI: 10.1145/1007912.1007944. URL: https://doi.org/10.1145/1007912.1007944 (cit. on p. 291).

[Hen+13] Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. "Quantitative relaxation of concurrent data structures". In: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. ACM, 2013, pp. 317–328. DOI: 10.1145/2429069.2429109. URL: https://doi.org/10.1145/2429069.2429109 (cit. on pp. 5, 297).

[HW90] Maurice Herlihy and Jeannette M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (1990), pp. 463–492. DOI: 10.1145/78969.78972. URL: https://doi.org/10.1145/78969.78972 (cit. on pp. 5, 266, 273).

[HAN08] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. "Oracle Semantics for Concurrent Separation Logic". In: *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Ed. by Sophia Drossopoulou. Vol. 4960. Lecture Notes in Computer Science. Springer, 2008, pp. 353–367. DOI: 10.1007/978-3-540-78739-6_27. URL: https://doi.org/10.1007/978-3-540-78739-6_27 (cit. on pp. 167, 175).

[Iri22] Iris Team (The). *The Iris 3.6 Technical Appendix*. Tech. rep. 2022. URL: https://plv.mpi-sws.org/iris/appendix-3.6.pdf (cit. on pp. 136, 138, 143).

[IO01] Samin S. Ishtiaq and Peter W. O'Hearn. "BI as an Assertion Language for Mutable Data Structures". In: *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*. Ed. by Chris Hankin and Dave Schmidt. ACM, 2001, pp. 14–26. DOI: 10.1145/360204.375719. URL: https://doi.org/10.1145/360204.375719 (cit. on p. 63).

[Jag+13] Radha Jagadeesan, Gustavo Petri, Corin Pitcher, and James Riely. "Quarantining Weakness - Compositional Reasoning under Relaxed Memory Models (Extended Abstract)". In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. Vol. 7792. Lecture Notes in Computer Science. Springer, 2013, pp. 492–511. DOI: 10.1007/978-3-642-37036-6_27. URL: https://doi.org/10.1007/978-3-642-37036-6_27 (cit. on pp. 5, 297).

[JR14] Radha Jagadeesan and James Riely. "Between Linearizability and Quiescent Consistency - Quantitative Quiescent Consistency". In: *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II*. Vol. 8573. Lecture Notes in Computer Science. Springer, 2014, pp. 220–231. DOI: 10.1007/978-3-662-43951-7_19. URL: https://doi.org/10.1007/978-3-662-43951-7_19 (cit. on pp. 5, 297).

[JB12] Jonas Braband Jensen and Lars Birkedal. "Fictional Separation Logic". In: *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. Ed. by Helmut Seidl. Vol. 7211. Lecture Notes in Computer Science. Springer, 2012, pp. 377–396. DOI: 10.1007/978-3-642-28869-2_19. URL: https://doi.org/10.1007/978-3-642-28869-2_19 (cit. on pp. 1, 73).

[Jou18] Jacques-Henri Jourdan. *Insufficient synchronization in Arc::get_mut*. Rust issue #51780, https://github.com/rust-lang/rust/issues/51780. 2018 (cit. on pp. 5, 8, 177).

[Jun+23] Jaehwang Jung, Janggun Lee, Jaemin Choi, Jaewoo Kim, Sunho Park, and Jeehoon Kang. "Modular Verification of Safe Memory Reclamation in Concurrent Separation Logic". In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (2023). DOI: 10.1145/3622827. URL: https://doi.org/10.1145/3622827 (cit. on p. 299).

[Jun+18a] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "RustBelt: Securing the Foundations of the Rust Programming Language". In: *PACMPL* 2.POPL (2018) (cit. on pp. 3, 4, 10, 13, 47, 143, 173, 180, 181, 187, 188, 190, 191, 225).

[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*. ACM, 2016, pp. 256–269. DOI: 10.1145/2951913.2951943. URL: https://doi.org/10.1145/2951913.2951943 (cit. on pp. 3, 63).

[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. URL: https://doi.org/10.1017/S0956796818000151 (cit. on pp. 1, 3, 63, 67, 75, 174).

[Jun+20] 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. URL: https://doi.org/10.1145/3371113 (cit. on pp. 6, 257, 258, 268, 297).

[Jun+15] 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*. ACM, 2015, pp. 637–650. DOI: 10.1145/2676726.2676980. URL: https://doi.org/10.1145/2676726.2676980 (cit. on pp. 1, 3, 6, 63, 73, 258).

[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*. Vol. 74. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 17:1–17:29. DOI: 10.4230/LIPIcs.ECOOP.2017.17. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2017.17 (cit. on pp. 1, 3, 5, 8, 9, 27, 42, 43, 61, 75, 99, 115, 124, 133, 149, 159, 167, 177, 201, 208, 298).

[Kan+17] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. "A promising semantics for relaxed-memory concurrency". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. ACM, 2017, pp. 175–189. DOI: 10.1145/3009837.3009850. URL: https://doi.org/10.1145/3009837.3009850 (cit. on pp. 2, 24, 27, 32, 44, 45, 57).

[Kan+15] Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. "A formal C memory model supporting integer-pointer casts". 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 Stephen M. Blackburn. ACM, 2015, pp. 326–335. DOI: 10.1145/2737924.2738005. URL: https://doi.org/10.1145/2737924.2738005 (cit. on p. 51).

[Khy+17] Artem Khyzha, Mike Dodds, Alexey Gotsman, and Matthew J. Parkinson. "Proving Linearizability Using Partial Orders". 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*. Vol. 10201. Lecture Notes in Computer Science. Springer, 2017, pp. 639–667. DOI: 10.1007/978-3-662-54434-1_24. URL: https://doi.org/10.1007/978-3-662-54434-1_24 (cit. on p. 297).

[KN18] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. 2018. URL: https://doc.rust-lang.org/stable/book/ (cit. on pp. 3, 173, 174).

[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: *PACMPL* 2.ICFP (2018) (cit. on pp. 3, 63, 174).

[Kre+17] 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*. Vol. 10201. Lecture Notes in Computer Science. Springer, 2017, pp. 696–723. DOI: 10.1007/978-3-662-54434-1_26. URL: https://doi.org/10.1007/978-3-662-54434-1_26 (cit. on pp. 3, 63).

[KTB17] Robbert Krebbers, Amin Timany, and Lars Birkedal. "Interactive Proofs in Higher-Order Concurrent Separation Logic". In: *POPL*. 2017. DOI: 10.1145/3009837.3009855 (cit. on pp. 3, 63, 174).

[Kri+20] Siddharth Krishna, Michael Emmi, Constantin Enea, and Dejan Jovanovic. "Verifying Visibility-Based Weak Consistency". In: *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*. Lecture Notes in Computer Science. Springer, 2020, pp. 280–307. DOI: 10.1007/978-3-030-44914-8_11. URL: https://doi.org/10.1007/978-3-030-44914-8_11 (cit. on pp. 5, 297).

[KSB17] Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. "A relational model of types-and-effects 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*. ACM, 2017, pp. 218–231. DOI: 10.1145/3009837.3009877. URL: https://doi.org/10.1145/3009837.3009877 (cit. on p. 297).

[Kro+20] Morten Krogh-Jespersen, Amin Timany, Marit Edna Ohlenbusch, Simon Oddershede Gregersen, and Lars Birkedal. “Aneris: A Mechanised Logic for Modular Reasoning about Distributed Systems”. In: *ESOP*. Vol. 12075. Lecture Notes in Computer Science. Springer, 2020, pp. 336–365 (cit. on p. 1).

[LGV16] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. “Taming release-acquire consistency”. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. ACM, 2016, pp. 649–662. DOI: 10.1145/2837614.2837643. URL: https://doi.org/10.1145/2837614.2837643 (cit. on p. 27).

[Lah+17] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. “Repairing sequential consistency in C/C++11”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. ACM, 2017, pp. 618–632. DOI: 10.1145/3062341.3062352. URL: https://doi.org/10.1145/3062341.3062352 (cit. on pp. 2, 3, 13, 15, 16, 18, 21, 23, 27).

[Lam79] Leslie Lamport. “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs”. In: *IEEE Trans. Computers* 28.9 (1979), pp. 690–691. DOI: 10.1109/TC.1979.1675439. URL: https://doi.org/10.1109/TC.1979.1675439 (cit. on p. 1).

[Lê+13] Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. “Correct and efficient work-stealing for weak memory models”. In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP ’13, Shenzhen, China, February 23-27, 2013*. ACM, 2013, pp. 69–80. DOI: 10.1145/2442516.2442524. URL: https://doi.org/10.1145/2442516.2442524 (cit. on p. 299).

[Lee+18] Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. “Reconciling high-level optimizations and low-level code in LLVM”. In: *Proc. ACM Program. Lang.* 2.OOPSLA (2018), 125:1–125:28. DOI: 10.1145/3276495. URL: https://doi.org/10.1145/3276495 (cit. on p. 51).

[Lee+17] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. “Taming undefined behavior in LLVM”. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. Ed. by Albert Cohen and Martin T. Vechev. ACM, 2017, pp. 633–647. DOI: 10.1145/3062341.3062343. URL: https://doi.org/10.1145/3062341.3062343 (cit. on pp. 30, 57).

[Lee+23] Sung-Hwan Lee, Minki Cho, Roy Margalit, Chung-Kil Hur, and Ori Lahav. “Putting Weak Memory in Order via a Promising Intermediate Representation”. In: PLDI (2023) (cit. on pp. 2, 25, 57, 302).

[Lee+20] Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. “Promising 2.0: global optimizations in relaxed memory concurrency”. In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. ACM, 2020, pp. 362–376. DOI: 10.1145/3385412.3386010. URL: https://doi.org/10.1145/3385412.3386010 (cit. on pp. 2, 57).

[Lep+22] Rodolphe Lepigre, Michael Sammler, Kayvan Memarian, Robbert Krebbers, Derek Dreyer, and Peter Sewell. “VIP: verifying real-world C idioms with integer-pointer casts”. In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–32. DOI: 10.1145/3498681. URL: https://doi.org/10.1145/3498681 (cit. on p. 51).

[Ler+12] Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. *The CompCert Memory Model, Version 2*. Research Report RR-7987. INRIA, June 2012, p. 26. URL: https://hal.inria.fr/hal-00703441 (cit. on p. 30).

[LF13] 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*. ACM, 2013, pp. 459–470. DOI: 10.1145/2491956.2462189. URL: https://doi.org/10.1145/2491956.2462189 (cit. on p. 297).

[Mat16] Nicholas D. Matsakis. *Introducing MIR*. 2016. URL: https://blog.rust-lang.org/2016/04/19/MIR.html (cit. on p. 50).

[Mem+19] Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. “Exploring C semantics and pointer provenance”. In: *Proc. ACM Program. Lang.* 3.POPL (2019), 67:1–67:32. DOI: 10.1145/3290380. URL: https://doi.org/10.1145/3290380 (cit. on p. 51).

[Mem+16] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. “Into the depths of C: elaborating the de facto standards”. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. Ed. by Chandra Krintz and Emery D. Berger. ACM, 2016, pp. 1–15. DOI: 10.1145/2908080.2908081. URL: https://doi.org/10.1145/2908080.2908081 (cit. on p. 51).

[MJ21] 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. URL: https://doi.org/10.1145/3473571 (cit. on pp. 6, 257).

[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. URL: https://doi.org/10.1145/3408978 (cit. on pp. 1, 5, 6, 61, 99, 106, 115, 124, 298, 302).

[Mic04] Maged M. Michael. “Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects”. In: *IEEE Trans. Parallel Distrib. Syst.* 15.6 (June 2004), 491–504. ISSN: 1045-9219. DOI: 10.1109/TPDS.2004.8. URL: https://doi.org/10.1109/TPDS.2004.8 (cit. on p. 299).

[MS96] 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*. ACM, 1996, pp. 267–275. DOI: 10.1145/248052.248106. URL: https://doi.org/10.1145/248052.248106 (cit. on pp. 134, 268, 273).

[MK23] Ike Mulder and Robbert Krebbers. “Proof Automation for Linearizability in Separation Logic”. In: *Proc. ACM Program. Lang.* 7.OOPSLA1 (2023). DOI: 10.1145/3586043. URL: https://doi.org/10.1145/3586043 (cit. on p. 298).

[MKG22] Ike Mulder, Robbert Krebbers, and Herman Geuvers. “Diaframe: automated verification of fine-grained concurrent programs in Iris”. In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, 809–824. ISBN: 9781450392655. DOI: 10.1145/3519939.3523432. URL: https://doi.org/10.1145/3519939.3523432 (cit. on p. 298).

[Nan+14] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. “Communicating state transition systems for fine-grained concurrent resources”. In: *ESOP*. LNCS. 2014. DOI: 10.1007/978-3-642-54833-8_16 (cit. on p. 1).

[Nei94] Gil Neiger. “Set-Linearizability”. In: *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, Los Angeles, California, USA, August 14-17, 1994*. ACM, 1994, p. 396. DOI: 10.1145/197917.198176. URL: https://doi.org/10.1145/197917.198176 (cit. on pp. 5, 297).

[O’H07] Peter W. O’Hearn. “Resources, concurrency, and local reasoning”. In: *Theoretical Computer Science* 375.1-3 (2007). DOI: 10.1016/j.tcs.2006.12.035 (cit. on p. 1).

[OP99] Peter W. O’Hearn and David J. Pym. “The logic of bunched implications”. In: *Bull. Symb. Log.* 5.2 (1999), pp. 215–244. DOI: 10.2307/421090. URL: https://doi.org/10.2307/421090 (cit. on p. 63).

[ON22] Joakim Öhman and Aleksandar Nanevski. “Visibility Reasoning for Concurrent Snapshot Algorithms”. In: *POPL*. 2022 (cit. on p. 297).

[OD18] Peizhao Ou and Brian Demsky. “Towards understanding the costs of avoiding out-of-thin-air results”. In: *Proc. ACM Program. Lang.* 2.OOPSLA (2018), 136:1–136:29. DOI: 10.1145/3276506. URL: https://doi.org/10.1145/3276506 (cit. on p. 24).

[Par+24] Sunho Park, Jaewoo Kim, Ike Mulder, Jaehwang Jung, Janggun Lee, Robbert Krebbers, and Jeehoon Kang. “A Proof Recipe for Linearizability in Relaxed Memory Separation Logic (to appear in PLDI 2024)”. In: PLDI (2024) (cit. on p. 298).

[PLV19] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. “Bridging the gap between programming languages and hardware weak memory models”. In: *Proc. ACM Program. Lang.* 3.POPL (2019), 69:1–69:31. DOI: 10.1145/3290382. URL: https://doi.org/10.1145/3290382 (cit. on p. 2).

[PSN16] Anton Podkopaev, Ilya Sergey, and Aleksandar Nanevski. “Operational Aspects of C/C++ Concurrency”. In: *CoRR* abs/1606.01400 (2016). URL: http://arxiv.org/abs/1606.01400 (cit. on pp. 27, 57).

[Pul+18] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. “Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8”. In: *Proc. ACM Program. Lang.* 2.POPL (2018), 19:1–19:29. DOI: 10.1145/3158107. URL: https://doi.org/10.1145/3158107 (cit. on pp. 2, 58, 168).

[Pul+19] Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung Hwan Lee, and Chung-Kil Hur. "Promising-ARM/RISC-V: a simpler and faster operational concurrency model". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. ACM, 2019, pp. 1–15. DOI: 10.1145/3314221.3314624. URL: https://doi.org/10.1145/3314221.3314624 (cit. on pp. 2, 58).

[Raa+19] Azalea Raad, Marko Doko, Lovro Rozic, Ori Lahav, and Viktor Vafeiadis. "On library correctness under weak memory consistency: specifying and verifying concurrent libraries under declarative consistency models". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 68:1–68:31. DOI: 10.1145/3290381. URL: https://doi.org/10.1145/3290381 (cit. on pp. 5, 6, 263, 266, 276, 297).

[RPDG14] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. "TaDA: A Logic for Time and Data Abstraction". In: *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings*. Vol. 8586. Lecture Notes in Computer Science. Springer, 2014, pp. 207–231. DOI: 10.1007/978-3-662-44202-9_9. URL: https://doi.org/10.1007/978-3-662-44202-9_9 (cit. on pp. 1, 6, 257, 258).

[RP+16] Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner, and Julian Sutherland. "Modular Termination Verification for Non-blocking Concurrency". In: *ESOP*. Vol. 9632. Lecture Notes in Computer Science. Springer, 2016, pp. 176–201 (cit. on p. 1).

[SLS05] William Scherer, Doug Lea, and Michael Scott. "A scalable elimination-based exchange channel". In: (2005) (cit. on p. 5).

[SNB15] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. "Specifying and Verifying Concurrent Algorithms with Histories and Subjectivity". In: *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Vol. 9032. Lecture Notes in Computer Science. Springer, 2015, pp. 333–358. DOI: 10.1007/978-3-662-46669-8_14. URL: https://doi.org/10.1007/978-3-662-46669-8_14 (cit. on p. 297).

[Ser+16] Ilya Sergey, Aleksandar Nanevski, Anindya Banerjee, and Germán Andrés Delbianco. "Hoare-style specifications as correctness conditions for non-linearizable concurrent objects". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. ACM, 2016, pp. 92–110. DOI: 10.1145/2983990.2983999. URL: https://doi.org/10.1145/2983990.2983999 (cit. on p. 297).

[SWT18] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. "Programming and Proving with Distributed Protocols". In: 2.POPL (2018). DOI: 10.1145/3158116. URL: https://doi.org/10.1145/3158116 (cit. on p. 1).

[Sew+10] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. "x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors". In: *Commun. ACM* 53.7 (2010), pp. 89–97. DOI: 10.1145/1785414.1785443. URL: https://doi.org/10.1145/1785414.1785443 (cit. on pp. 15, 17).

[Sim+22] Ben Simner, Alasdair Armstrong, Jean Pichon-Pharabod, Christopher Pulte, Richard Grisenthwaite, and Peter Sewell. "Relaxed virtual memory in Armv8-A". In: *Programming Languages and Systems*. Ed. by Ilya Sergey. Cham: Springer International Publishing, 2022, pp. 143–173. ISBN: 978-3-030-99336-8 (cit. on p. 58).

[Sim+20] Ben Simner, Shaked Flur, Christopher Pulte, Alasdair Armstrong, Jean Pichon-Pharabod, Luc Maranget, and Peter Sewell. "ARMv8-A System Semantics: Instruction Fetch in Relaxed Architectures". In: *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings*. Ed. by Peter Müller. Vol. 12075. Lecture Notes in Computer Science. Springer, 2020, pp. 626–655. DOI: 10.1007/978-3-030-44914-8_23. URL: https://doi.org/10.1007/978-3-030-44914-8_23 (cit. on pp. 2, 58).

[SL23] Abhishek Kr Singh and Ori Lahav. "An Operational Approach to Library Abstraction under Relaxed Memory Concurrency". In: *Proc. ACM Program. Lang.* 7.POPL (2023). DOI: 10.1145/3571246. URL: https://doi.org/10.1145/3571246 (cit. on p. 297).

[SN04] Robert C. Steinke and Gary J. Nutt. "A unified theory of shared memory consistency". In: *J. ACM* 51.5 (2004), pp. 800–849. DOI: 10.1145/1017460.1017464. URL: https://doi.org/10.1145/1017460.1017464 (cit. on p. 27).

[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*. Vol. 8410. Lecture Notes in Computer Science. Springer, 2014, pp. 149–168. DOI: 10.1007/978-3-642-54833-8_9. URL: https://doi.org/10.1007/978-3-642-54833-8_9 (cit. on pp. 1, 6, 257, 258).

[Sve+18] Kasper Svendsen, Jean Pichon-Pharabod, Marko Doko, Ori Lahav, and Viktor Vafeiadis. “A Separation Logic for a Promising Semantics”. In: *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Vol. 10801. Lecture Notes in Computer Science. Springer, 2018, pp. 357–384. DOI: 10.1007/978-3-319-89884-1_13. URL: https://doi.org/10.1007/978-3-319-89884-1_13 (cit. on pp. 1, 24, 167).

[TDV15] Joseph Tassarotti, Derek Dreyer, and Viktor Vafeiadis. “Verifying read-copy-update in a logic for weak memory”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. ACM, 2015, pp. 110–120. DOI: 10.1145/2737924.2737992. URL: https://doi.org/10.1145/2737924.2737992 (cit. on pp. 5, 133, 167).

[Tre86] R.K. Treiber. *Systems Programming: Coping with Parallelism*. Research Report RJ. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986. URL: https://books.google.lu/books?id=YQg3HAAACAAJ (cit. on pp. 145, 273).

[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*. ACM, 2013, pp. 377–390. DOI: 10.1145/2500365.2500600. URL: https://doi.org/10.1145/2500365.2500600 (cit. on pp. 1, 297).

[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*. ACM, 2014, pp. 691–707. DOI: 10.1145/2660193.2660243. URL: https://doi.org/10.1145/2660193.2660243 (cit. on pp. 1, 2, 5, 8, 99, 115, 124, 133, 134, 159, 177, 201, 245, 273).

[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*. ACM, 2013, pp. 867–884. DOI: 10.1145/2509136.2509532. URL: https://doi.org/10.1145/2509136.2509532 (cit. on pp. 1, 2, 24, 25, 99, 124, 133, 168).

[VP07] Viktor Vafeiadis and Matthew J. Parkinson. “A Marriage of Rely/Guarantee and Separation Logic”. In: *CONCUR*. 2007, pp. 256–271 (cit. on p. 1).

[VB23] Simon Friis Vindum and Lars Birkedal. “Spirea: A Mechanized Concurrent Separation Logic for Weak Persistent Memory”. In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (2023). DOI: 10.1145/3622820. URL: https://doi.org/10.1145/3622820 (cit. on p. 302).

[Wat+20] Conrad Watt, Christopher Pulte, Anton Podkopaev, Guillaume Barbier, Stephen Dolan, Shaked Flur, Jean Pichon-Pharabod, and Shu-yu Guo. “Repairing and mechanising the JavaScript relaxed memory model”. In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. ACM, 2020, pp. 346–361. DOI: 10.1145/3385412.3385973. URL: https://doi.org/10.1145/3385412.3385973 (cit. on p. 2).

[WRP19] Conrad Watt, Andreas Rossberg, and Jean Pichon-Pharabod. “Weakening WebAssembly”. In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019), 133:1–133:28. DOI: 10.1145/3360559. URL: https://doi.org/10.1145/3360559 (cit. on p. 2).