Scaling Up
Relaxed Memory Verification
with Separation Logics

Hoang-Hai Dang
June 6, 2022

Dissertation zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
(Dr.-Ing.)
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

DATE OF THE COLLOQUIUM: ???

EXAMINERS:
DEAN:

Work done at
the Max Planck Institute for Software Systems
Kaiserslautern and Saarbrücken
Abstract

Reasoning about concurrency is hard. Reasoning about concurrency in a full-blown, non-toy language like C/C++ or Rust, which encompasses many interweaving complex features, is even harder. Yet, realistic concurrency involves relaxed memory models, which are significantly harder to reason about than the simple, traditional concurrency model that is sequential consistency. In order to scale up verifications to realistic concurrency in complex languages, 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 miss potential unsoundness in our verifications. Only recently was it possible to acquire all of these ingredients, with the help of the concurrent separation logics framework Iris.

Even so, the intricacy of relaxed memory features and the ingenuity of programmers who exploit those features are not to be taken lightly. To tackle such monumental complexity, in this dissertation, I present how to build strong, abstract, modular, extensible, and machine-checked separation logics in Iris, using multiple layers of abstractions. I report two main applications of such logics: (i) the verification of the Rust type system with relaxed memory models, in which relaxed memory effects are safely hidden from the types, and (ii) the compositional specification and verification of relaxed memory libraries, in which relaxed memory effects are exposed to clients.

Zusammenfassung

Es kommt noch.
Acknowledgments

It’s coming.
Contents

Abstract iii
Zusammenfassung iii
Acknowledgments v
Contents viii
List of Figures xi
List of Tables xi
Glossary xiii
I Introduction 1
1.1 Reasoning about Relaxed Memory Concurrency 2
1.2 RustBelt Relaxed: Verifying Rust’s Type System in RMC 3
1.3 Compass: Strong and Compositional Specifications of Relaxed-Memory Libraries 5
1.4 Structure 7
1.5 Publications and Collaborations 8

I SEPARATION LOGICS 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 \( \lambda_{\text{Rust}} \) Language 45
4.1 Language Syntax 45
4.2 Language Expression Reductions 48
4.3 The Complete Operational Semantics of Relaxed \( \lambda_{\text{Rust}} \) 52

5 More Background: Iris, A Framework for Concurrent Separation Logics 55
5.1 Basic Rules 56
5.2 Ghost State and Resource Algebras 57
5.3 Invariants and Fancy Updates 59
5.4 Hoare Triples 61
5.5 Adequacy 61
5.6 Some Common Rules for WPs and Hoare Triples 62
5.7 Weakest Pre-conditions and Invariants 63
5.8 Properties of Propositions 64
### Contents

14.3 Adaption of the Lifetime Logic’s Model in iRC11 ........................................ 183

15 GPS Single-Location Protocols ........................................................................ 189

16 Verification of RwLock ......................................................................................... 191

17 Verification of Arc
    17.1 The Core Arc library ..................................................................................... 194
    17.2 Verification of Core Arc with Cancelable GPS Protocols ............................. 194
    17.3 Verification of Arc’s Full APIs ....................................................................... 201
    17.4 Insufficient Synchronization in get_mut ....................................................... 205

III COMPASS ........................................................................................................... 207

18 Background: Strong Specifications with Logical Atomicity ............................... 211
    18.1 Sequential Specifications for Queues .............................................................. 211
    18.2 SC Specifications with Logical Atomicity ..................................................... 212
    18.3 Logically Atomic Specifications in RMC with Views .................................... 214

19 Strong Compass Specifications with Richer Partial Orders .............................. 217
    19.1 Graph-Based Specs to Encode Partial Orders ................................................. 217
    19.2 Weaker Specs by Abandoning Abstract States .............................................. 222

20 Verifications of Stacks and Queues ..................................................................... 225

21 Helping in Exchangers Specifications .................................................................. 227

22 Verifications of Exchangers and the Elimination Stack ...................................... 229

23 Discussions on Specifications ............................................................................... 231

24 Conclusion ............................................................................................................. 233

Bibliography ............................................................................................................. 235

Versicherung an Eides Statt ....................................................................................... 243
# List of Figures

1.1 Depdency graph of this dissertation's chapters (contracts) ........................................... 10
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 (OOTIA) 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 $\lambda_{Rust}$ language syntax. .................................................................... 46
4.2 Some syntactic sugars for $\lambda_{Rust}$. ......................................................................... 47
4.3 CPS notations for $\lambda_{Rust}$. ...................................................................................... 48
4.4 Relaxed $\lambda_{Rust}$ expression semantics. ................................................................. 50
4.5 The combined 1-thread semantics of ORC11 machine semantics and $\lambda_{Rust}$ expression semantics. 52
4.6 Threadpool semantics. ............................................................................................... 53
5.1 An excerpt of Iris grammar. ......................................................................................... 56
5.2 Basic rules of several Iris connectives. ....................................................................... 57
5.3 Basic rules of Iris ghost ownership and basic updates. .............................................. 58
5.4 Some rules for Iris invariants and fancy updates. ....................................................... 60
5.5 Some common rules for Iris weakest pre-conditions and Hoare triples. .................. 63
5.6 Some rules for Iris weakest pre-conditions and invariants. ........................................ 64
5.7 Some properties of timeless propositions and persistent propositions. .................... 65
5.8 Several rules for the Auth(M) RA. .............................................................................. 66
6.1 Pure primitive WPs in the RMC base logic ................................................................. 72
6.2 Main properties of the base logic's local assertions .................................................... 75
6.3 The base logic's primitive Hoare rules for fences ...................................................... 77
6.4 The base logic's primitive Hoare rules for non-atomic reads and writes ................... 78
6.5 The base logic's primitive Hoare rules for atomic reads and writes .......................... 79
6.6 The base logic's primitive Hoare rule for CASEs ....................................................... 80
6.7 The base logic's primitive WP rule for CASEs .......................................................... 83
6.8 Several agreements between the global ghost state and local assertions .................. 87
6.9 Several update rules for the global ghost state and local assertions ........................ 87
7.1 $iRC11$ rules for fence modalities ............................................................................... 95
7.2 $iRC11$ rules for objective propositions and the objective modality ........................ 98
7.3 $iRC11$ rules for view-explicit modalities .................................................................. 100
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.4</td>
<td>iRC11 rules for the subjective modality</td>
<td>102</td>
</tr>
<tr>
<td>8.1</td>
<td>Rules for iRC11 non-atomic points-to</td>
<td>104</td>
</tr>
<tr>
<td>9.1</td>
<td>Basic properties of assertions related to the atomic points-to</td>
<td>109</td>
</tr>
<tr>
<td>9.2</td>
<td>Conversions between the non-atomic and atomic points-to assertion</td>
<td>111</td>
</tr>
<tr>
<td>9.3</td>
<td>iRC11 read rules with the atomic points-to assertion</td>
<td>113</td>
</tr>
<tr>
<td>9.4</td>
<td>iRC11 write rules with the atomic points-to assertion</td>
<td>115</td>
</tr>
<tr>
<td>9.5</td>
<td>An iRC11 CAS rule with the atomic points-to assertion</td>
<td>117</td>
</tr>
<tr>
<td>9.6</td>
<td>An iRC11 CAS rule with the atomic points-to in single-writer mode</td>
<td>118</td>
</tr>
<tr>
<td>9.7</td>
<td>Several properties of ghost abstractions for the atomic RA</td>
<td>120</td>
</tr>
<tr>
<td>10.1</td>
<td>iRC11 rules for objective invariants</td>
<td>127</td>
</tr>
<tr>
<td>10.2</td>
<td>iRC11 rules for cancelable invariants</td>
<td>129</td>
</tr>
<tr>
<td>10.3</td>
<td>Stronger iRC11 rules for cancelable invariants</td>
<td>131</td>
</tr>
<tr>
<td>10.4</td>
<td>Properties of the RA FRACVIEWR for cancelable invariants</td>
<td>134</td>
</tr>
<tr>
<td>10.5</td>
<td>The interface of non-atomic invariants</td>
<td>136</td>
</tr>
<tr>
<td>11.1</td>
<td>Message-Passing with Loops</td>
<td>138</td>
</tr>
<tr>
<td>11.2</td>
<td>Hoare proof outlines for mp</td>
<td>140</td>
</tr>
<tr>
<td>11.3</td>
<td>Message-Passing with Reclamation</td>
<td>142</td>
</tr>
<tr>
<td>11.4</td>
<td>Hoare proof outlines for mp_reclaim</td>
<td>143</td>
</tr>
<tr>
<td>11.5</td>
<td>Derived iRC11 atomic access rules with the view-join modality</td>
<td>145</td>
</tr>
<tr>
<td>11.6</td>
<td>A Spawn-and-Join library</td>
<td>146</td>
</tr>
<tr>
<td>11.7</td>
<td>Hoare proof outlines for SPAWN-SPEC</td>
<td>148</td>
</tr>
<tr>
<td>11.8</td>
<td>A simple release-acquire implementation for Treiber stacks</td>
<td>149</td>
</tr>
<tr>
<td>11.9</td>
<td>Bag or per-element specifications for Treiber stacks</td>
<td>151</td>
</tr>
<tr>
<td>11.10</td>
<td>Hoare proof outlines for try_push_swap</td>
<td>155</td>
</tr>
<tr>
<td>11.11</td>
<td>Hoare proof outlines for try_pop</td>
<td>156</td>
</tr>
<tr>
<td>12.1</td>
<td>Key rules for cancellable invariants in Iris-SC</td>
<td>163</td>
</tr>
<tr>
<td>13.1</td>
<td>Selected rules of SC RustBelt's lifetime logic</td>
<td>170</td>
</tr>
<tr>
<td>13.2</td>
<td>The life cycle of borrows and lifetimes</td>
<td>170</td>
</tr>
<tr>
<td>13.3</td>
<td>MP verified with the lifetime logic in Iris-SC</td>
<td>172</td>
</tr>
<tr>
<td>14.1</td>
<td>More selected rules for lifetimes and full borrows, ported to λRust + ORC11</td>
<td>177</td>
</tr>
<tr>
<td>14.2</td>
<td>Selected rules for other borrow alternatives, sound in λRust + ORC11</td>
<td>181</td>
</tr>
<tr>
<td>17.1</td>
<td>Implementation of Core Arc</td>
<td>194</td>
</tr>
<tr>
<td>17.2</td>
<td>Selected iRC11 rules for GPS protocols</td>
<td>195</td>
</tr>
<tr>
<td>17.3</td>
<td>Counting permissions for Core Arc</td>
<td>197</td>
</tr>
<tr>
<td>17.4</td>
<td>An excerpt of Rust's Arc&lt;T&gt; and Weak&lt;T&gt; APIs</td>
<td>201</td>
</tr>
<tr>
<td>17.5</td>
<td>Rust's implementation (excerpt) of Arc::get_mut and Arc::drop</td>
<td>202</td>
</tr>
<tr>
<td>17.6</td>
<td>A truncated history of the Arc counter</td>
<td>203</td>
</tr>
<tr>
<td>18.1</td>
<td>Specifications of Queue operations, from sequential, to SC concurrency and strong RMC</td>
<td>213</td>
</tr>
<tr>
<td>18.2</td>
<td>A Message-Passing (MP) client with Queues</td>
<td>215</td>
</tr>
<tr>
<td>19.1</td>
<td>Compass Specs for Queues</td>
<td>218</td>
</tr>
<tr>
<td>19.2</td>
<td>A proof sketch of Message Passing with queues</td>
<td>221</td>
</tr>
</tbody>
</table>
List of Tables

14.1 Comparison of borrow types .............................................................. 180
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.

**CONCURRENT SEPARATION LOGICS**\(^2\) (hereafter, CSLs) provide a feasible approach to abstract and modular control of interferences: 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.

**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

---

\(^1\) Lamport, “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs” [Lam79].

\(^2\) 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; 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+20; MJP20].
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. 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.

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., and is now broadly adopted by the RMC models of Rust, Java, OCaml, JavaScript, and WebAssembly. 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. 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.

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.

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.

In order to achieve realistic guarantees for actual concurrent code in the wild, it is important to scale the reasoning principles of RMC logics to full-blown, non-toy languages like C/C++ or Rust, which encompass many interweaving complex features. To this end, 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, which comes with strong tactics support in Coq. Using Iris, Jung et al. 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, Kaiser et al. have 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.

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. In this dissertation, I present the “abstraction layers” needed to assimilate such logics. I report two main applications:

1. RustBelt Relaxed: the verification of the Rust’s type system in RMC, in which relaxed memory effects are safely hidden from the types; and
2. Compass: the compositional specification and verification of relaxed memory libraries, in which relaxed memory effects are exposed to clients.

1.2 RustBelt Relaxed: Verifying Rust’s Type System in RMC

Rust is a young and evolving programming language that aims to...
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.

*RustBelt* 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 states. 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 *RBrlx* for short), the first formal validation of the soundness of Rust under RMC. Although based closely on the original *RustBelt*, *RBrlx* 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

---

\[19\] Jung et al., “RustBelt: Securing the Foundations of the Rust Programming Language” [Jun+18a].
Rust codebase.\textsuperscript{20}

**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 intertwine resource accounting and physical synchronization, and thus ubiquitously affects all relaxed memory reasoning rules. Fortunately, in \texttt{RBrlx} 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,\textsuperscript{21} and reference counting,\textsuperscript{22} as well as the \texttt{RBrlx} work. However, these works (except \texttt{Cosmo}\textsuperscript{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 \texttt{RBrlx} 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 \texttt{GPS}\textsuperscript{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.

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\textsuperscript{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\textsuperscript{26}). Therefore, various linearizability-like criteria have been proposed as alternatives,\textsuperscript{27} especially for relaxed memory.\textsuperscript{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 operations respect some partial orders. These works, however, have little support for modular client reasoning. Therefore, we want to improve the

\textsuperscript{20}Jourdan, *Insufficient synchronization in Arc::get_mut* [Jou18].

\textsuperscript{21}Tassarotti et al., “Verifying read-copy-update in a logic for weak memory” [TDV15].

\textsuperscript{22}Doko and Vafeiadis, “Tackling Real-Life Relaxed Concurrency with FSL++” [DV17].

\textsuperscript{23}Mével et al., “Cosmo: a concurrent separation logic for multicore OCaml” [MJP20].

\textsuperscript{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].

\textsuperscript{25}Herlihy and Wing, “Linearizability: A Correctness Condition for Concurrent Objects” [HW90].

\textsuperscript{26}[SLS05; HRV15].

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

\textsuperscript{28}[Bur+12; BDG13; Jag+13; Doh+18; Don+18; Raa+19; EE19; Kri+20].
Introduction

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, 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.

Logical atomicity has been applied mostly in the SC setting, and only recently did Mével and Jourdan 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, is sound only for the Multicore OCaml memory model, and they only gave one spec for a concurrent queue and verified one client.

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. (“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.
1.4 Structure

This dissertation is composed of three parts: Part I presents the basic layers needed to build RMC separation logics with Iris, while Part II and Part III discuss how such logics can be extended and/or applied for RustBelt Relaxed and Compass, respectively. Part II and Part III are independent from each other, but both rely on materials presented in Part I. Each part will discuss the context, the challenges, the solutions, and the results separately, as well as related and future work in details. The conclusion (Chapter 24) provides a high level summary and potential future research directions. Note that Figure 1.1 (page 10) provides the dependency graph for all chapters in this dissertation.

Part I discusses the features and the construction of iRC11, our extensible RMC separation logic for RC11. It provides a brief background review on relaxed memory models and the Iris framework, for which readers who are familiar with the topics can skip. It presents ORC11, an operational variant of RC11 that is needed to instantiate Iris. 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. The remaining chapters of Part I 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 cancellable 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 II 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 II elaborate on how iRC11 synchronized ghost state and cancellable 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 cancellable invariants and GPS single-location protocols. A bit of history on how the bug in Arc was found will be provided.

Part III 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. Finally, the flexibility

34 It indeed delayed the publication of the RBrlx work by a year.


36 Jourdan, Insufficient synchronization in Arc::get_mut [Jou18].
of the specs and the relations among them will be discussed.

1.5 Publications and Collaborations

This dissertation contains the work of the following two papers:


While much of the text from these two papers are reused in this dissertation, 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;
- §6: the detailed model of the iRC11 base logic;
- §7: the models of various iRC11’s modalities;
- §8: the model of the non-atomic points-to assertion, which depends tightly on ORC11’s race detector;
- §9: the model of the atomic points-to assertion;
- §10: the detailed interfaces and models of iRC11 objective invariants and cancelable invariants;
- §11: several example verifications demonstrating many mid-level features of iRC11;
- §14: more details on how the lifetime logic was ported to iRC11;
- §15: the detailed model of GPS single-location protocols, built atop atomic points-to;
- §17: the detailed verification of the Rust’s standard library Arc;
- §19: the detailed interpretations of Compass specs in iRC11 with logical atomicity;
- §20: the verifications of stacks and queues against Compass specs;
- §21: the complete specs of the exchanger with helping;
- §22: the verifications of the exchanger and its client—the elimination stack.
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-Aquire 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, which this dissertation is based on, are the results of delightful collaborations. Although I led the efforts in both works, they would not be completed without the team efforts with many fellow researchers.

For RBtlx, 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 that used iRC11 to re-prove the soundness of the lifetime logic. I re-verified the Rust concurrent libraries by substantial extending the original proofs in SC RustBelt. It was also Jacques-Henri’s original suggestion to prove GPS protocols on top of iRC11, but I only completed that task 2 years later.

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 linearizability-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+21].

COQ ARTIFACTS. Unless noted explicitly, the 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/orc11

- iRC11:
  https://gitlab.mpi-sws.org/iris/gpfsl

- RBtlx:
  https://gitlab.mpi-sws.org/iris/lambda-rust/-/tree/masters/weak_mem

- Compass:
  https://gitlab.mpi-sws.org/iris/gpfsl/-/tree/ci/compass

37 Jung et al., “RustBelt: Securing the Foundations of the Rust Programming Language” [Jun+18a].
Figure 1.1: Dependency graph of this dissertation's chapters (constructs)
Part I

SEPARATION LOGICS FOR RELAXED MEMORY
This part discusses the features and the construction of iRC11, a concurrent separation logic that is sound for the relaxed memory model RC11. We do not assume prior knowledge either on relaxed memory models or concurrent separation logics. Therefore, we start with Chapter 2 to review relaxed memory models defined in axiomatic style, specifically for the C11 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 $\lambda_{Rust}$ language used in RustBelt, which is presented in Chapter 4. Developing such an operational semantics for RC11 is a necessary prerequisite for instantiating Iris. We give a brief review of the Iris separation logic framework in Chapter 5 and discuss the instantiation of Iris with ORC11 in Chapter 6, 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 7, following iGPS, 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. Chapter 8 and Chapter 9 present the construction for the core ownership assertions of iRC11: the non-atomic and atomic points-to. Chapter 10 introduces invariants—the standard principle for concurrently sharing resources—but with RMC-specific limitations. Finally, Chapter 11 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.
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) 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 and RC11. 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 will present ORC11, the operational version of RC11.

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 2.1a, we initialize two memory locations $\ell_x$ and $\ell_y$ to 0 non-atomically, then spawn two threads $\pi$ (on the left) and $\rho$ (on the right). Thread $\pi$ intends to pass a “message” to $\rho$. The message, 42, is stored in $\ell_x$ (line $\pi 1$). $\pi$ then sets the boolean flag $\ell_y$ to 1 (line $\pi 2$), to signal to $\rho$ that the message is ready to be received. Once $\rho$ sees the flag set (line $\rho 1$), it attempts to read the message from $\ell_x$ (line $\rho 2$). However, both the intended value of 42 as well as the initial value of 0 could be read.


\[ \ell_x := rlx 42; \ \rho_1: \text{if}\ \text{rlx} \ell_y \neq 0 \text{ then} \]
\[ \ell_y := rlx 1; \ \rho_2: \text{acq} \ell_y; // 0 \text{ or } 42 \]

(a) MP with relaxed accesses.

\[ \ell_x := na 0; \ \ell_y := na 0; \]
\[ \pi_1: \ell_x := na 42; \ \rho_1: \text{if}\ \text{acq} \ell_y \neq 0 \text{ then} \]
\[ \pi_2: \ell_y := rel 1; \ \rho_2: \text{acq} \ell_y; // 42 \]

(c) MP with release-acquire accesses.

\[ \ell_x := na 0; \ \ell_y := na 0; \]
\[ \pi_1: \ell_x := na 42; \ \rho_1: \text{if}\ \text{acq} \ell_y \neq 0 \text{ then} \]
\[ \pi_2: \ell_y := sc 1; \ \rho_2: \text{fence}; // 0 \text{ or } 42 \]

(d) MP with relaxed accesses and fences.

\[ \ell_x := na 0; \ \ell_y := na 0; \]
\[ \pi_1: \ell_x := na 42; \ \rho_1: \text{if}\ \text{acq} \ell_y \neq 0 \text{ then} \]
\[ \pi_2: \ell_y := rel 1; \ \rho_2: \text{fence}; // 42 \]

(b) MP with SC accesses.

That is, even though \( \rho \) has read 1 from \( \ell_y \), it is not guaranteed to read 42 from \( \ell_x \). This is because the relaxed accesses of \( \ell_y \) are not enough to establish synchronization between \( \pi \) and \( \rho \).

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 2.1a, thread \( \rho \) may see that \( \pi_2 \) (\( \pi \)'s write to \( \ell_y \)) is executed out-of-order, before \( \pi_1 \) (\( \pi \)'s write to \( \ell_x \)), and therefore \( \rho \) reads 0 from \( \ell_x \) in line \( \rho_2 \). What is happening under the hood is that hardware and/or compilers may deduce that \( \pi \)'s writes are of independent memory locations, and thus may reorder them.\(^4\)

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 \( \pi \)'s write to \( \ell_x \) (\( \pi_1 \)) and \( \rho \)'s read from \( \ell_x \) (\( \rho_2 \)). 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 2.1a, the relaxed accesses on \( \ell_y \) do not establish synchronization between the accesses on \( \ell_x \).

**SC accesses** (sc) are the strongest option to establish synchronization, and we use them in Example 2.1b for the accesses of \( \ell_y \).\(^5\) If \( \rho \)'s read of \( \ell_y \) (line \( \rho_1 \)) is not zero, then it reads from \( \pi \)'s write of 1 to \( \ell_y \) (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 \( \rho \)'s read of \( \ell_x \) is synchronized with \( \pi \)'s write to it. Since the write of 42 is the most recent write to \( \ell_x \), we know that thread \( \rho \) must read 42 in \( \rho_2 \).

**RelEASE-AcQUIRE accesses**. Instead of using the costly SC accesses, we can use the release-acquire idiom to establish synchronization, as in Example 2.1c. Here, \( \pi \) uses a release (rel) write in \( \pi_2 \), and \( \rho \) uses an

\(^{4}\text{Note that from thread } \pi \text{'s point of view, such reordering does not really matter as it cannot distinguish the effects, which, on the contrary, are distinguishable to the concurrently running thread } \rho.\)

\(^{5}\text{According to C11, SC accesses can have subtle behaviors when mixed with other kinds of accesses. We refer interested readers to the RC11 paper ([Lah+17]) for more details. In this dissertation, we do not focus on SC accesses. We only mention them here for the purpose of demonstration.}\)
acquire (\textit{acq}) read in \(\rho_1\). If \(\rho_1\) reads 1 from \(\pi_2\), C11’s release-acquire semantics on the location \(\ell_x\) 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

\textbf{RELEASE-ACQUIRE FENCES.} We can also achieve release-acquire synchronization using relaxed accesses with fences, as in Example 2.1d. Here, thread \(\pi\) performs a release fence (\textit{fence}_{rel}) after the write to \(\ell_x\), and then relaxedly (\textit{rlx}) writes to \(\ell_y\). Meanwhile, \(\rho\) performs an acquire fence (\textit{fence}_{acq}) once it relaxedly reads 1 from \(\ell_y\). Note that there is no happens-before relation between the relaxed accesses of \(\ell_y\), but C11 guarantees happens-before between the accesses of \(\ell_x\) (lines \(\pi\) and \(\rho\) 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 \(\rho_3\).

\textbf{DATA RACES.} Note that in Example 2.1a, where we do not have sufficient synchronization between the accesses to \(\ell_x\), the worst thing that can happen is that \(\rho\) would read unwanted values. However, if we were to replace the \textit{rlx} accesses of \(\ell_x\) with non-atomic accesses (\textit{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 \(\ell_x\), 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.

⁶In x86-TSO ([Sew+10]), release and acquire accesses are the default and weakest accesses.
In the following, we provide an excerpt of the RC11 formalization—following Lahav et al.\textsuperscript{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.

### 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 $\text{Loc}$ for memory locations and $\text{Val}$ for values stored in memory, with meta-variables $\ell \in \text{Loc}$ and $v \in \text{Val}$, respectively.

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

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

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

AccessMode’s LATTICE

\[
\begin{array}{ccc}
\text{na} \sqsubseteq o & & \text{rlx} \sqsubseteq \text{acq}
\
\text{rel} \sqsubseteq \text{relacq} & & \text{acq} \sqsubseteq \text{relacq}
\
& & o \sqsubseteq \text{sc}
\end{array}
\]

**Definition 2.3 (Memory Access Event).** Each memory access generates an event of type $\text{MemEvent}$, with the meta-variable $\varepsilon$.

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

Specifically:

- $R^o(\ell, v)$: a Read of $v$ from $\ell$, with access mode $o \in \{\text{na, rlx, acq, sc}\}$.
- $W^o(\ell, v)$: a Write of $v$ to $\ell$, with access mode $o \in \{\text{na, rlx, rel, sc}\}$.
- $U^{r_o, o_w}(\ell, v_r, v_w)$: a read-modify-write (Update) to $\ell$, with read value $v_r$ and write value $v_w$, and read access mode $o_r \in \{\text{rlx, acq, sc}\}$, and write access mode $o_w \in \{\text{rlx, rel, sc}\}$.

- $F^o$: a memory Fence, with $o \in \{\text{acq, rel, relacq, sc}\}$.

**Definition 2.4 (Memory Event Projections).** For a memory event $\varepsilon$, the projections loc, mod, val$_r$, and val$_w$ respectively give $\varepsilon$’s location, access mode, read value and write value when applicable. More specifically, loc is 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:

- $U^{\text{rlx, rlx}(\_)}$.mod ::= $\text{rlx}$
- $U^{\text{rlx, rel}(\_)}$.mod ::= $\text{rel}$
- $U^{\text{acq, rlx}(\_)}$.mod ::= $\text{acq}$
- $U^{\text{acq, rel}(\_)}$.mod ::= $\text{relacq}$
• $\text{U}^{\text{sc}, \text{sc}}(\_).\text{mod} ::= \text{sc}$

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

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

**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., $RW:=R\cup W$. We use subscript and superscript respectively to filter the sets by accessed location and access mode, e.g., \( W^{\text{rel}}_{\text{loc}} ::= \{ e \in W \mid e.\text{loc} = \ell \land e.\text{mod} \sqsupseteq \text{rel} \}$.

**Notation 2.7** (Memory Event Relations). For a binary relation on events $R \in \text{MemEvent} \times \text{MemEvent}$, $R^+$, and $R^*$ respectively denote its reflexive, transitive, and reflexive-transitive closures. $\text{dom}(R)$ and $\text{codom}(R)$ denote the domain and codomain 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 $\text{[A]}$ stands for the identity relation on the set $A$. Consequently, $\text{[A]}: R$ can be understood as filtering $R$ on the left with $A$, while $R: [B]$ filters $R$ on the right with $B$. That is, $\text{[A]}: R = \{(e_a, e_b) \in R \mid e_a \in A\}$, and $R: [B] = \{(e_a, e_b) \in R \mid e_b \in B\}$. Finally, $\text{[A]}: R: [B] = R \cap (A \times B)$.

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

$$=_f ::= \{(e_a, e_b) \mid f(e_a) = f(e_b)\}$$
$$\neq_f ::= \{(e_a, e_b) \mid f(e_a) \neq f(e_b)\}$$

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, $R|_{\text{loc}}$ and $R|_{\not\text{loc}}$ 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 $(E, \text{po}, \text{rf}, \text{mo})$:

- $E$ is the set of memory events (\text{MemEvent}) in $G$.
- The program order $\text{po}$ is a strict\footnote{It is irreflexive, i.e., $(\epsilon, \epsilon) \not\in \text{po}$.} partial order that orders each thread’s event by the program’s control flow. For simplicity, RC11 assumes that for each location $\ell$, $E$ contains a Write event $e_0^\ell ::= \text{w}^\ell(\ell, 0)$ as the initialization for $\ell$. $\text{po}$ is then required to order initialization events before all other events, i.e., $E_0 \times (E \setminus E_0) \subseteq \text{po}$ where $E_0 ::= \{ e_0^\ell \in E \}$ is the set of $E$’s initialization events.
- The reads-from relation $\text{rf}$ relates a write with a read that reads from it, i.e.,
Background: Relaxed Memory Models

(i) \( \text{rf} \subseteq [\text{wu}] = \text{loc} : [\text{ru}] \);\(^{10}\) and

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

(iii) \( \text{rf} \) is injective: if \( (\varepsilon^1_w, \varepsilon_r) \in \text{rf} \) and \( (\varepsilon^2_w, \varepsilon_r) \in \text{rf} \) then \( \varepsilon^1_w = \varepsilon^2_w \).

• The modification order \( \text{mo} \) is a strict partial order that gives a strict total order on the write events of each location. That is, \( \text{mo} \) is a disjoint union of the relations \( \{\text{mo}_\ell\}_{\ell \in \text{loc}} \) where \( \text{mo}_\ell \) is a strict total order on \( (\text{wu})_\ell \).

The components are also used as projections, e.g., \( G.\text{mo} \). In cases where \( G \) is clear in the context, we may also drop the “\( G \)” part and just use \( \text{mo} \).

**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 \( \text{po} \), \( \text{mo} \), and \( \text{rf} \) 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., \( G.\text{r} \subseteq \text{codom}(G.\text{rf}) \). A candidate execution is always complete, but the reverse is not always true.
Definition 2.12 (Derived Relations). RC11 defines the following derived partial orders on execution graphs.

\[
\begin{align*}
rb & ::= (rf^{-1} : mo) \setminus [E] & \text{(reads before)} \\
eco & ::= (rf \cup mo \cup rb)^+ & \text{(extended coherence order)} \\
rs & ::= [U] ; \text{po}|_{\text{loc}}^\geq ; [WU]^{rlx} ; (rf ; [U])^* & \text{(release sequence)} \\
sw & ::= [E;\text{rel}] ; ([F] ; \text{po})^* ; rs ; rf ; [RU]^{rlx} ; (po ; [F])^* ; [E;\text{acq}] & \text{(synchronized-with)} \\
\text{hb} & ::= (\text{po} \cup \text{sw})^+ & \text{(happens-before)} \\
\text{psc} & ::= \ldots \text{(elided)} & \text{(partial SC)}
\end{align*}
\]

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

Example 2.13 (Illustrations of Derived Relations). Figure 2.3 demonstrates the derived relations on several execution graphs. Figure 2.3e 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).
22 Background: Relaxed Memory Models

We use dotted arrows, dash-dot-dotted arrows, filled arrows, filled arrows, and dash-dotted arrows, respectively for \( \text{rb} \), \( \text{eco} \), \( \text{rs} \), \( \text{sw} \), and \( \text{hb} \) edges.

2.2.3 Consistency

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

- \( \text{hb} ; \text{eco} \) is irreflexive; and \( \quad \) (RC11-COHERENCE)
- \( \text{psc} \) is acyclic; and \( \quad \) (RC11-SC)
- \( \text{po} \cup \text{rf} \) is acyclic. \( \quad \) (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.

\(^{12}\text{Boehm and Demsky, “Outlawing ghosts: avoiding out-of-thin-air results” [BD14].} \)
2.2.4 Data Races

Definition 2.15 ( Races). Two events \( \varepsilon_a \) and \( \varepsilon_b \) 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 \( \varepsilon_a.\text{loc} = \varepsilon_b.\text{loc} \text{ and } \{ \varepsilon_a, \varepsilon_b \} \cap G.(\mathbb{W}) \neq \emptyset \).

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

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

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 \( \ell_x \), where no \( \mathbb{hb} \) edge is established between the accesses, because we use only relaxed accesses for \( \ell_y \).

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:\(^\text{13}\)

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

Example 2.20 (C11 Coherence). C11 coherence axioms are demonstrated by several forbidden (non-consistent) behaviors\(^\text{14}\) in Figure 2.5.

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

\(^{13}\) Lahav et al., “Repairing sequential consistency in C/C++11” [Lah+17], §3.4, Proposition 1.

\(^{14}\) Batty et al., “Mathematizing C++ concurrency” [Bat+11], §2.7.
Background: Relaxed Memory Models

\[ \text{C11-NO-FUTURE-READ.} \]

\[ \text{C11-CO-RW.} \]

\[ \text{C11-CO-WW.} \]

\[ \text{C11-CO-WR.} \]

\[ \text{C11-CO-RR.} \]

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

- **C11-CORW** (Figure 2.5b) requires that a read may not happen before a write that \( \text{mo}\)-before the write it reads from.
- **C11-COWW** (Figure 2.5c) requires that \( \text{mo} \) and \( \text{hb} \) may not disagree.
- **C11-CORWR** (Figure 2.5d) requires that a read may not read from a write that is already hidden by (\( \text{mo}\)-before) another write that happens-before it.
- **C11-CORRR** (Figure 2.5e) requires that two reads connected by \( \text{hb} \) may not read from writes with the inverse order in \( \text{mo} \).

**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 2.6a gives an example program with an execution demonstrating its LB behavior in Figure 2.6c. 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 2.6c is consistent in C11, and so such behavior is allowed in C11.

The problem with LB is that, the same execution in Figure 2.6c justifies an undesirable behavior of the program in Figure 2.6b, 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 \(^{16}\) for an attempt.

RC11 resolves to a simpler solution: forbidding LB behaviors altogether, by requiring \( \text{po} \cup \text{rf} \) 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 overhead of working with RC11 vs. C11 may not be so significant in practice.

---

\(^{15}\) Kang et al., “A promising semantics for relaxed-memory concurrency” [Kan+17]; Chakraborty and Vafeiadis, “Grounding thin-air reads with event structures” [CV19].


\(^{17}\) Ou and Demsky, “Towards understanding the costs of avoiding out-of-thin-air results” [OD18].
Remark 2.22 (Consume Accesses and Locks). Unlike C11, RC11 does not consider consume accesses, a premature feature that is not implemented by major compilers, nor locks, which can be implemented with release-acquire accesses.

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

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,\textsuperscript{18} 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.\textsuperscript{19} In the next chapter, we present ORC11, an operational version of RC11 that is more convenient to build our separation logic iRC11 in Iris.

\textsuperscript{18}In contrast, the formulation of ORC11 in Chapter 3 is much more verbose.

\textsuperscript{19}Yet it is still achievable, by annotating resources on incoming and outcoming edges of an event node, as Vafeaidis et al. demonstrated with RSL and FSL (\cite{VN13,DV16}).
3

ORC11: Operational Repaired C11

Following iGPS, 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.

Features-wise, ORC11 is closely related to the axiomatic semantics of RC11. Most importantly, it forbids load-buffering (LB) behaviors, i.e., $po \cup rf$ is acyclic. Construction-wise, ORC11 follows the view-based approach to operational semantics for relaxed memory. More concretely, it follows the promising semantics formalization but without promises, and thus forbids LB. The promising semantics, however, does not model non-atomics. 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 $\lambda_{\text{Rust}}$ language which combines the expression reductions together with ORC11.

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

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 $\pi$’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 $PC_\pi$ of the thread $\pi$. More concretely, if we follow the language of iGPS and track writes to memory in views, we can defined a view as a map from memory locations.

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

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

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


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

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

7And thus the semantics of the relaxed $\lambda_{\text{Rust}}$ language is a combination of three sub-semantics: the expression semantics, the view-based machine semantics, and the race-detector semantics.

8Kaiser et al., “Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris” [Kai+17].
to timestamps: View ::= Loc → Time, where the timestamps are indices into an ordering of the writes to a location. With $V(ℓ) = t$, 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 $ℓ$.

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 $V_ρ$: $V_π ⊑ V_ρ$. 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 3.1a.

- After thread $π$ writes 42 to $ℓ_x$, its local view is $V_π^1$, as illustrated in Figure 3.1a 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 $t_1^x$ for the write 42 to $ℓ_x$, i.e., $V_π^1(ℓ_x) = t_1^x$.

- Similarly, after thread $π$ writes 1 to $ℓ_y$, its local view $V_π^2$ includes the timestamp $t_2^y$: $V_π^2(ℓ_y) = t_2^y$. More importantly, $V_π^1 ⊑ V_π^2$: 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 $t_1^x$ (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 $t_1^x$. 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.
Put it differently, that thread $\rho$'s view before the non-atomic read of $\ell_x$ does not include $\ell_x$ approximates the fact that the non-atomic write to $\ell_x$ 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 2.1c (Example 2.1) can also be explained with views, using Figure 3.1b. Because we instead use a release write of 1 to $\ell_y$, thread $\pi$ releases its local view $V^2_\pi$ through the write (which also include the write itself). When thread $\rho$ reads that write using an acquire read, it acquires that view into its local view $V^1_\rho$. Accordingly, $V^2_\pi \subseteq V^1_\rho$, so thread $\rho$ has observed all writes to $\ell_x$, and can safely read $\ell_x$ non-atomically. Furthermore, thread $\rho$ reads from $\ell_x$’s latest write, which is 42.

In other words, $V^2_\pi \subseteq V^1_\rho$ encodes the synchronized-with (sw) relation between the release-acquire pair, and transitively the happens-before relation (hb). Effectively, thread $\pi$’s write of 42 to $\ell_x$ happens before thread $\rho$’s read from $\ell_x$, 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 2.1d (Example 2.1) is a bit more interesting, using Figure 3.1c.

- Here, thread $\pi$’s relaxed write to $\ell_y$, like in Example 3.1, does not release the current local view $V^1_\pi$ at the point of the write. However, unlike in Example 3.1, $\pi$’s release fence which comes before guarantees that the write to $\ell_y$ does release $\pi$’s local view before the fence, i.e., $V^1_\pi$.

- On the other side, $\rho$’s read of $\ell_y$, if reads 1, will acquire $V^1_\rho$, but does not immediately join $V^1_\pi$ into its local view $V^1_\rho$ right after the read, i.e., $V^2_\pi \not\subseteq V^1_\rho$. Instead, later, thread $\rho$’s acquire fence will perform that join, so that after the acquire fence, $V^1_\rho \subseteq V^2_\rho$. Consequently, we again have the hb relation between the non-atomic write and read of $\ell_x$.

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 $\lambdaRust$ language (Chapter 4) adopts the CompCert model for locations,\(^\text{10}\) 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.

**Uninitialized Memory** $\lambdaRust$ 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 $\lambdaRust$, which in turn follows Lee et al.\(^\text{11}\) to use a poison value $\dagger$ for uninitialized memory.

**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).**

$$\begin{align*}
\pi, \rho & \in Thread ::= \mathbb{N}^+ \\
\ell & \in Loc ::= (i, n), \quad i \in \mathbb{N}^+, n \in \mathbb{Z} \\
v & \in Val ::= \dagger | \ldots \\
\omega & \in MemVal ::= \dagger | \dagger | v \in Val \\
t & \in Time ::= \mathbb{N}^+ \\
\alpha & \in ActIds ::= 2^{\mathbb{N}^+}
\end{align*}$$

- A thread-id $\pi$ or $\rho$ is a positive number.
• A location \( \ell \) 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 \( \perp \).

• 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 \( \dagger \) and \( \ddagger \) to respectively mark the allocated and deallocated states of a location.

• A timestamp \( t \) is a positive number.

• A set of actions \( \alpha \) is a set of positive numbers, which will be used to track sets of reads and writes.

Definition 3.5 (ORC11 Memory Access Event).

\[ \epsilon \in MemEvent ::= | R^o(\ell, v) | W^o(\ell, v) | U^{o_r-o_w}(\ell, vr, vw) | F^o | A(\ell, n \in \mathbb{N}^+) | D(\ell, n \in \mathbb{N}^+). \]

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 \( \ell \) of a block, and the size \( n \) of that block.

Definition 3.6 (Views).

\[ V \in View ::= Loc \times \{ w : Time, aw : ActIds, nr : ActIds, ar : ActIds \} \]

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 \( \ell \),

• \( V(\ell).w \) is the timestamp of the latest write to \( \ell \) that \( V \) has seen.

• \( V(\ell).aw \) is the set of atomic writes to \( \ell \) that \( V \) has seen.

• \( V(\ell).nr \) is the set of non-atomic reads from \( \ell \) that \( V \) has seen.

• \( V(\ell).ar \) is the set of atomic reads from \( \ell \) that \( V \) has seen.

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

\[ V_1 \subseteq V_2 ::= \forall \ell. V_1(\ell).w \leq V_2(\ell).w \wedge V_1(\ell).aw \subseteq V_2(\ell).aw \]
\[ \wedge V_1(\ell).nr \subseteq V_2(\ell).nr \wedge V_1(\ell).ar \subseteq V_2(\ell).ar \]
\[ V_1 \sqcup V_2 ::= \lambda \ell. \{ w := \max(\{V_1(\ell).w, V_2(\ell).w\}); \]
\[ aw := V_1(\ell).aw \cup V_2(\ell).aw; \]
\[ nr := V_1(\ell).nr \cup V_2(\ell).nr; \]
\[ ar := V_1(\ell).ar \cup V_2(\ell).ar \} \]

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., \( \text{dom}(V_1 \sqcup V_2) = \text{dom}(V_1) \cup \text{dom}(V_2) \). If some location
\( \ell \) is not in one view, then the value in the other view takes over for the join.

For view inclusion, we consider by default:

\[
V_1(\ell) \subseteq V_2(\ell) \text{ if } \ell \notin \text{dom}(V_1) \\
V_1(\ell) \not\subseteq V_2(\ell) \text{ if } \ell \in \text{dom}(V_1) \land \ell \notin \text{dom}(V_2)
\]

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

\( V \in \text{ThreadView} ::= \{ \text{rel} : \text{Loc} \rightarrow \text{View}, \text{frel} : \text{View}, \text{cur} : \text{View}, \text{acq} : \text{View} \} \)

A thread-view\(^{12}\) is used to track the observations of a thread, and has four components. For a thread \( \pi \) to have the thread-view \( V \) at its current program counter \( \text{PC}_\pi \),

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

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

- \( \text{dom}(V.\text{rel}) \subseteq \text{dom}(V.\text{cur}) \) \hspace{1cm} (tview-dom)
- \( \forall \ell. V.\text{rel}(\ell) \subseteq V.\text{cur} \) \hspace{1cm} (tview-rel)
- \( V.\text{frel} \subseteq V.\text{cur} \) \hspace{1cm} (tview-frel)
- \( V.\text{cur} \subseteq V.\text{acq} \) \hspace{1cm} (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.

\[
V_1 \subseteq V_2 ::= (\forall \ell. V_1(\ell) \subseteq V_2(\ell)) \land V_1.\text{frel} \subseteq V_2.\text{frel} \\
\land V_1.\text{cur} \subseteq V_2.\text{cur} \land V_1.\text{acq} \subseteq V_2.\text{acq}
\]

\[
V_1 \sqcup V_2 ::= \begin{cases} 
\text{rel} := \lambda \ell. V_1(\ell) \sqcup V_2(\ell); \\
\text{frel} := V_1.\text{frel} \sqcup V_2.\text{frel}; \\
\text{cur} := V_1.\text{cur} \sqcup V_2.\text{cur}; \\
\text{acq} := V_1.\text{acq} \sqcup V_2.\text{acq}
\end{cases}
\]

\(^{12}\)This is inspired by thread-views of the promising semantics ([Kan+17]).
Definition 3.11 (Global Memory).

\[ M \in \text{MsgPool} ::= \text{Loc} \xrightarrow{\text{fin}} \text{Time} \xrightarrow{\text{fin}} \{ \text{val} : \text{MemVal}, \text{view} : \text{View} \} \]

\[ m \in \text{ExtMsg} ::= \{ \text{ts} : \text{Time}, \text{val} : \text{MemVal}, \text{view} : \text{View} \} \]

The global memory, or the message pool \( M \) 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 (\( \text{View} \)).

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

For some timestamp \( t \), the pair \( M(\ell)(t) \) carries the information about a write to \( \ell \) identified by the timestamp \( t \). If the write is an non-atomic write, then \( M(\ell)(t).\text{view} = \text{None} \). Otherwise, if the write is an atomic write, then \( M(\ell)(t).\text{view} = \text{Some}(V) \) 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 \( \text{None} \), and simply write \( V \) for \( \text{Some}(V) \).\(^{13}\)

The message type \( \text{ExtMsg} \) 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: \( \text{MsgPool} \approx \text{Loc} \xrightarrow{\text{fin}} \text{ExtMsg} \).

Property 3.12 (View Closedness). \[ V \in M \quad \forall V \in M \]

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

Property 3.13 (Global Memory Wellformedness). A global memory \( M \) is wellformed if the following hold.

\[ \forall \ell, m. m \in M(\ell) \land m.\text{view} \neq \bot \Rightarrow m.\text{view}(\ell') \in M \quad (\text{WF-MEM-CLOSED}) \]

\[ \forall \ell, m. m \in M(\ell) \land m.\text{view} \neq \bot \Rightarrow m.\text{view}(\ell).w = m.\text{ts} \quad (\text{WF-MSG-VIEW}) \]

\[ \forall \ell, t, M(\ell)(t).\text{val} = \uparrow \Rightarrow t = \min(\text{dom}(M(\ell))) \quad (\text{WF-MEM-ALLOC}) \]

\[ \forall \ell, t, M(\ell)(t).\text{val} = \uparrow \Rightarrow t = \max(\text{dom}(M(\ell))) \quad (\text{WF-MEM-DEALLOC}) \]

\( \text{WF-MEM-CLOSED} \) requires that \( M \) is closed in itself, i.e., any view of any write messages in \( M \) only refers to messages also in \( M \). \( \text{WF-MSG-VIEW} \) requires that the view of a write message contains exactly the timestamp of that message. \( \text{WF-MEM-ALLOC} \) (resp. \( \text{WF-MEM-DEALLOC} \)) 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).

\[ N \in \text{RaceView} ::= \text{View} \]

\[ \varsigma \in \text{GlobalState} ::= \text{MsgPool} \times \text{RaceView} \]

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

Property 3.15 (Global Machine State Wellformedness). A global state \( (M, N) \) is wellformed if

\(^{13}\)Note that instead of using the option type, we can also require that the view \( V \) of a non-atomic write to be empty (\( \text{None} \)).
• \( \mathcal{M} \) is wellformed (Property 3.13); and
• \( \mathcal{N} \) is closed in \( \mathcal{M} \); and
• \( \mathcal{N} \) observe the deallocations in \( \mathcal{M} \), i.e., \( \forall \ell, t. \mathcal{M}(\ell)(t).\text{val} = \dagger \Rightarrow t \leq \mathcal{N}(\ell).w. \)

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).

The injection of memory values (\( \text{MemVal} \)) into values (\( \text{Val} \)) is defined by the following rules.

\[
\begin{align*}
\text{MV-VAL} & : m \equiv m \\
\text{MV-AVAL} & : \dagger \equiv \dagger
\end{align*}
\]

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 \( \dagger \) is returned. There is no injection of the deallocated value \( \dagger \) into values.

Definition 3.17 (Unallocated Locations).

A location \( \ell \) is called unallocated in \( \mathcal{M} \) if it has not been allocated or it has been deallocated in \( \mathcal{M} \).

\[
\begin{align*}
\ell \not\in \text{dom}(\mathcal{M}) & \quad \ell \in \text{unalloc}(\mathcal{M}) \\
\exists t. \mathcal{M}(\ell)(t) = (\dagger, \_ ) & \quad \ell \in \text{unalloc}(\mathcal{M})
\end{align*}
\]

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.

\( \text{(ALLOC-SAFE)} \)

• The deallocation of a block must happen-after all accesses to it.

\( \text{(DEALLOC-SAFE)} \)

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).**

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

- $V_{\text{cur}}(\ell).w \leq t$: 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 $\ell$ 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, $V.w$ is the timestamp $t$ of the write. The read id $r$ is added to the non-atomic read component $V.nr$ if this is a non-atomic read ($o = \text{na}$), otherwise $r$ is added to the atomic read component $V.ar$.

- A read only changes the current and acquire (simple) views of $V$.

- The view $V$ is joined into both the new current and acquire views $V'_{\text{cur}}$ and $V'_{\text{acq}}$, so that both views observe at least the read and the write.

- If this is an atomic read ($r1x \sqsubseteq o$), then the view $V_r$ of the write is also joined into the acquire view $V'_{\text{acq}}$. This encodes the delayed synchronization of relaxed reads, where the view $V_r$ sent over the write is temporarily stored in the acquire view $V'_{\text{acq}}$ 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 ($\text{acq} \sqsubseteq o$), then the view $V_r$ of the write is immediately joined into the current view $V'_{\text{cur}}$ (recall Example 3.2).

- The computation maintains wellformedness of thread-views (Property 3.9).

\(^\dagger\)Note that if $o = \text{na}$, then $V_r = \bot$ (None).

**Definition 3.21 (Post-Write Thread-Views).**

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

$V = [\ell \leftarrow \{ w := t; aw := \emptyset; nr := (o = \text{na}) \? \{ r \} : \emptyset; ar := (o \subseteq \text{rlx}) \? \{ r \} : \emptyset \}]$

$V_{\text{cur}}' = (\text{acq} \subseteq o) \? V_{\text{cur}} \cup V \cup V_{r} : V_{\text{cur}} \cup V$

$V_{\text{acq}}' = (\text{rlx} \subseteq o) \? V_{\text{acq}} \cup V \cup V_{r} : V_{\text{acq}} \cup V$

$\left( V_{\text{rel}}, V_{\text{freel}}, V_{\text{cur}}, V_{\text{acq}} \right) \xrightarrow{\text{mod, } t, \ell, V_{r}, V_{\text{cur}}, V_{\text{acq}}} \left( V_{\text{rel}}, V_{\text{freel}}, V_{\text{cur}}, V_{\text{acq}}' \right)$

OM-POST-WRITE-TVIEW

$V_{\text{cur}}(\ell).w < t$

$V = [\ell \leftarrow \{ w := t; aw := (\text{rlx} \subseteq o) \? \{ t \} : \emptyset; nr := \emptyset; ar := \emptyset \}]$

$V_{\text{cur}}' = V_{\text{cur}} \cup V \quad V_{\text{acq}}' = V_{\text{acq}} \cup V$

$V_{r}' = V_{\text{rel}}(\ell) \cup (\text{rel} \subseteq o) \? V_{r}' : V \quad V_{\text{rel}}' = V_{\text{rel}}[\ell \leftarrow V']$

$V_{w}' = (\text{rlx} \subseteq o) \? V_{r}' \cup V_{\text{freel}} \cup V_{r} : \perp$

$\left( V_{\text{rel}}, V_{\text{freel}}, V_{\text{cur}}, V_{\text{acq}} \right) \xrightarrow{\text{mod, } t, \ell, V_{r}, V_{\text{cur}}, V_{\text{acq}}} \left( V_{\text{rel}}', V_{\text{freel}}, V_{\text{cur}}', V_{\text{acq}}' \right)$

$\begin{align*}
\text{Figure 3.2: Computations of post thread-views for read and write operations.}
\end{align*}$

- $V_{\text{cur}}(\ell).w < t$: the fresh timestamp $t$ picked for the write must be mo-later than the current view of $\ell$.
- The view $V$ tracks the identifying information of the write. In particular, $V.w$ is the timestamp $t$ of the write. $t$ is also added to the atomic write component $V.aw$ if this is an atomic write.
- The write updates the current and acquire (simple) views, and the component $\ell$ of the per-location release view $V_{\text{rel}}$ of $V$.
- The view $V$ is joined into both the new current and acquire views $V_{\text{cur}}'$ and $V_{\text{acq}}'$, so that both views observe at least the write.
- The view $V'$ is the new release write for the location $\ell$, and is updated into to $V_{\text{rel}}'(\ell)$ ($V_{\text{rel}}'(\ell) = V_{\text{rel}}[\ell \leftarrow V']$). In particular, if this is at least a release write, then $\ell$’s new release view $V'$ also includes the new current view $V_{\text{cur}}'$. (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 $\ell$, so that po-later relaxed writes to the same $\ell$ will indeed release $V_{\text{cur}}'$.
- The view $V_{w}$ of the write itself is $\perp$ if this is a non-atomic write. Otherwise, it includes at least (i) the new release view $V_{\text{rel}}'(\ell)$ for $\ell$, and (ii) the view $V_{\text{rel}}$ of the most recent same-thread release fence, and (iii) the view $V$ 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 $\ell$ (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.
The relation \( M \mid V \xrightarrow{\ell \notin \text{unalloc}(M)} M' \mid V' \) relates a pair of global memory \( M \) and a local thread-view \( V \) before a machine step that generates a memory event \( \ell \) to a corresponding pair \( M' \) and \( V' \) after the step. \( \ell \) is an optional action id associated with the event if it is a read, and \( V \) 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 \( \xrightarrow{\ell}(\ell, v) \) does not change the global memory \( M \), and is only possible if \( \ell \) is alive in \( M \), whose memory value \( \omega \) is injected into the read value \( v \) so that a read of an uninitialized location will return the poison value \( \dagger \). The thread-view is updated from \( V \) to \( V' \) using the timestamp \( \ell \) and the view \( V \) of

\[ \forall \ell \notin \text{unalloc}(M), \quad M(\ell)(t) = (\omega, V), \quad \omega \equiv v \]

\[ V \xrightarrow{\text{R.}\ell, \ell, V_v, r} V' \]

\[ M \mid V \xrightarrow{\ell \notin \text{unalloc}(M)} M' \mid V' \]

**OM-write**

\[ \ell \notin \text{unalloc}(M) \quad t \notin M(\ell) \]

\[ M' = M[\ell \leftarrow M(\ell)[t \leftarrow (v, V_w)]] \]

\[ V \xrightarrow{\text{W.}\ell, \ell, V_v, r} V' \]

\[ M \mid V \xrightarrow{\ell \notin \text{unalloc}(M)} M' \mid V' \]

**OM-update**

\[ \ell \notin \text{unalloc}(M) \quad M(\ell)(t) = (v, V) \quad t_w = t_r + 1 \quad t_w \notin M(\ell) \]

\[ M' = M[\ell \leftarrow M(\ell)[t_w \leftarrow (t_w, V_w)]] \]

\[ V \xrightarrow{\text{R.}\ell, \ell, V_v, r, \text{W.}\ell, V_v, V_w} V' \]

\[ M \mid V \xrightarrow{\ell \notin \text{unalloc}(M)} M' \mid V' \]

**OM-acq-fence**

\[ \ell \notin \text{unalloc}(M) \quad M \mid (V, \text{rel}, V, \text{fre}, V, \text{acq}, \text{acq}) \]

**OM-rel-fence**

\[ \ell \notin \text{unalloc}(M) \quad M \mid (\ell \notin \text{dom}(V, \text{rel}) \cup V, \text{cur}, V, \text{acq}, V, \text{acq}) \]

**OM-alloc**

\[ \ell = (i, n') \quad \{i\} \times \mathbb{N} \neq \text{dom}(M) \]

\[ M' = M[\ell + m \leftarrow (t_m, \perp) \mid m \in [0, n]] \]

\[ V \xrightarrow{\\text{na.}\ell, m \leftarrow (t_m, \perp), \parallel} \ldots \parallel \text{na.}\ell + (n-1), m \leftarrow (t_m, \perp), \parallel} \]

\[ m_s = [(t_m, \perp) \mid m \in [0, n]] \]

**OM-free**

\[ \ell = (i, n') \quad \text{dom}(M) \cap \{i\} \times \mathbb{N} = \{i\} \times ([0, n') \cup [n, n + n]) \]

\[ \forall m \in [0, n), \ell + m \notin \text{unalloc}(M) \quad \forall m \in [0, n], t \in \text{dom}(M(\ell + m)), t < t_m \]

\[ M' = M[\ell + m \leftarrow (t_m, \perp) \mid m \in [0, n)] \]

\[ V \xrightarrow{\\text{na.}\ell, m \leftarrow (t_m, \perp), \parallel} \ldots \parallel \text{na.}\ell + (n-1), m \leftarrow (t_m, \perp), \parallel} \]

\[ m_s = [(t_m, \perp) \mid m \in [0, n]] \]

**Definition 3.22** (ORC11 View-based Reductions).

\[ \xrightarrow{\ell \notin \text{unalloc}(M)} M \mid V \xrightarrow{\ell} M' \mid V' \]
the write, and the fresh action id \( r \) for the read, following Definition 3.20.

- **OM-write** says that a write \( \mathcal{W}(\ell, v) \) is only possible if \( \ell \) is alive in \( M \), and \( M \) is updated to \( M' \) with a new message \( (t, v, V_w) \) for \( \ell \), where \( t \) is a fresh timestamp in \( M \) for \( \ell \), and \( V_w \) is computed following Definition 3.21, which also defines how \( V' \) 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 \( \ell \) in \( M \). This allows “holes” in the set of used timestamps, so that writes to \( \ell \) 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 \( \ell \) seen by the current thread-view, so as to guarantee that \( \text{mo}_\ell \) agrees with the current thread’s po.

- **OM-update** combines the effects of both **OM-read** and **OM-write**, saying that an update \( \mathcal{U}(\ell, v_r, v_w) \) reads from an existing write message \( (t_r, v_r, V_r) \) and updates the memory \( M \) with a new write message \( (t_w, v_w, V_w) \). The new write message’s timestamp \( t_w \) must be fresh for \( \ell \) in \( M \), 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 \( \text{mo}_\ell \), 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 \( V' \) is computed from \( V \) 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 \( V'.\text{acq} \) into the new current component \( V'.\text{cur} \), 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 \( V.\text{cur} \) into the new release components \( V'.\text{rel} \) (the per-location release views) and \( V'.\text{frel} \) (the release-fence view).

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

- Finally, **OM-free** says that a deallocation \( D(\ell, n) \) requires that \( \ell \) is indeed the base location of a block whose size is \( n \), and all locations in the block are alive (\( \forall m \in [0, n), \ell + m \notin \text{unalloc}(M) \)).
deallocation inserts \( n \) write messages \( ms \) into the global memory \( M \), all with the deallocated memory value \( \star \), and the maximal timestamps \( (\forall m \in [0, n), t \in \text{dom}(M(\ell + m)), t < t_m) \).

**Property 3.23** (Wellformedness of View-based Reductions). The pair \( M \mid V \) is wellformed if \( V \) is wellformed (Property 3.9) and \( M \) is wellformed (Property 3.13) and \( V \) is closed in \( M \) (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 \( (V \subseteq V') \).

### 3.4 The Data-Race Detector

The goal of the race detector, as the name suggests, 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 reached 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”.

The aim of the race detector is to make sure locally that conflicting 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 \( M \) and the race detector’s state \( N \in \text{View} \), in combination of the executing thread \( \pi \)’s thread-view \( V \). In practice, only the current component \( V\text{.cur} \in \text{View} \) will be used, because that view encodes what have happened before the thread \( \pi \)’s program counter \( PC_\pi \), and recall that races are due to the lack of hb edges between conflicting accesses.

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

The race detector checks for data-race freedom (DRF) for each memory access on an \( \ell \) that \( \pi \) is going to perform. If it is not data-race free, then the execution gets stuck. Otherwise, the race detector state \( N \) 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 \( N' \) for the race detector.

\(^{17}\)There may be several ways for an execution to run into UBs, i.e., to get stuck (e.g., performing computations on poison \( \star \), 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.
\[
\begin{align*}
\text{DRF-read-na} & : N(\ell).w \leq \mathcal{V}.\text{cur}(\ell).w \quad \forall \ell \in \text{dom}(\mathcal{M}(\ell)), t \leq \mathcal{V}.\text{cur}(\ell).w \quad N(\ell).aw \subseteq \mathcal{V}.\text{cur}(\ell).aw \\
\mathcal{M}, N, \mathcal{V} & \vdash \text{RaceFree}(\mathcal{R}^\text{na}(\ell, v)) \\
\text{DRF-write-na} & : N(\ell) \subseteq \mathcal{V}.\text{cur}(\ell) \quad \forall \ell \in \text{dom}(\mathcal{M}(\ell)), t \leq \mathcal{V}.\text{cur}(\ell).w \\
\mathcal{M}, N, \mathcal{V} & \vdash \text{RaceFree}(\mathcal{W}^\text{na}(\ell, v)) \\
\text{DRF-read-at} & : r\text{lx} \subseteq o \quad N(\ell).w \leq \mathcal{V}.\text{cur}(\ell).w \\
\mathcal{M}, N, \mathcal{V} & \vdash \text{RaceFree}(\mathcal{R}^o(\ell, v)) \\
\text{DRF-write-at} & : r\text{lx} \subseteq o \quad N(\ell).w \leq \mathcal{V}.\text{cur}(\ell).w \quad N(\ell).nr \subseteq \mathcal{V}.\text{cur}(\ell).nr \\
\mathcal{M}, N, \mathcal{V} & \vdash \text{RaceFree}(\mathcal{W}^o(\ell, v)) \\
\text{DRF-update} & : \mathcal{M}, N, \mathcal{V} \vdash \text{RaceFree}(\mathcal{R}^\text{nu}(\ell, v)) \quad \mathcal{M}, N, \mathcal{V} \vdash \text{RaceFree}(\mathcal{W}^\text{nu}(\ell, v)) \\
\mathcal{M}, N, \mathcal{V} & \vdash \text{RaceFree}(\mathcal{U}^\text{nu}(\ell, v, v_w)) \\
\text{DRF-alloc} & : \mathcal{M}, N, \mathcal{V} \vdash \text{RaceFree}(\mathcal{A}(\ell, n)) \\
\text{DRF-dealloc} & : \forall i \in \langle n \rangle, t' \in \text{dom}(\mathcal{M}(\ell + i)), t' \leq \mathcal{V}.\text{cur}(\ell).w \quad \forall i \in \langle n \rangle, N(\ell + i) \subseteq \mathcal{V}.\text{cur}(\ell + i) \\
\mathcal{M}, N, \mathcal{V} & \vdash \text{RaceFree}(\mathcal{D}(\ell, n)) \\
\text{DRF-fence} & : \mathcal{M}, N, \mathcal{V} \vdash \text{RaceFree}(\mathcal{F}^o)
\end{align*}
\]

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

**Definition 3.25** (DRF Pre-conditions). \[ \mathcal{M}, N, \mathcal{V} \vdash \text{RaceFree}(\epsilon) \] \[ \mathcal{M}, N, \mathcal{V} \vdash \text{RaceFree}(\epsilon) \] says that the memory access \( \epsilon \) is data-race free when executed with the global state \( (\mathcal{M}, N) \) 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 \( \ell \) is data-race free if the thread has observed all writes to \( \ell \), atomic and non-atomic, tracked globally by \( \mathcal{M} \) and \( \mathcal{N} \). Note that the conditions concerning \( \mathcal{V}.\text{cur}(\ell).w \) by themselves are not sufficient: they only maintain that \( \mathcal{V} \) has observed the max-atomic maximum non-atomic write, which is only sufficient to guarantee observations of all non-atomic writes which must happen sequentially. The condition \( N(\ell).aw \subseteq \mathcal{V}.\text{cur}(\ell).aw \) 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 \( \ell \) is data-race free if the thread has observed all memory accesses to \( \ell \).
• **DRF-read-at** says that an atomic read from $\ell$ is data-race free if the thread has observed the latest non-atomic write to $\ell$. 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 $\ell$ 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).**

The relation $\mathcal{N} \xrightarrow{\varepsilon,r,m_s} \mathcal{N}'$ defines how a race-free event $\varepsilon$ for $\ell$ updates the global race detector state from $\mathcal{N}$ to $\mathcal{N}'$, using the optional action id $r$ if $\varepsilon$ is a read, and the list of write messages $m_s$ if $\varepsilon$ 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 $\ell$’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 $\ell$’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 $\ell$’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 $\ell$’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.

\[\text{The racing-ghost notation is due to Jan-Oliver Kaiser.}\]
\begin{itemize}
  \item DRF-Post-fence says that fences do not affect the race detector state.
\end{itemize}

Lemma 3.27. The DRF post-conditions (Definition 3.26) only grow the data-race view, i.e., $\mathcal{N} \subseteq \mathcal{N}'$. 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.\textsuperscript{19} developed for iGPS, in order to address the extra effects of relaxed accesses. To explain the necessity of this extension, we

\textsuperscript{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:

\[
\ell_x := 0; \\
* \ell_x || \ell_x := 37
\]

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 \( \ell_x \) is executed after the write to \( \ell_x \), but it does not detect a race in the interleaving where \( \ell_x \) 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:

\[
\ell_x := 0; \ell_y := 0; \\
* \ell_x; \ell_y := \text{rlx } 1 | \text{while } (\text{rlx } \ell_y == 0); \ell_x := 37
\]

Here, the non-atomic read in the left thread is guaranteed to be executed before the non-atomic write to \( \ell_x \) 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 \( \ell_x \) 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 $\ell_x$ by the left thread, when executed, will add a fresh read event $r_{na}$ into $N$’s global set of non-atomic reads for $\ell_x$ ($N(\ell_x.nr)$). The non-atomic write of $\ell_x$ 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 $\ell_x$ will be declared racy by the ORC11 race detector.

3.6 The Correspondence between RC11 and ORC11

To show a formal correspondence between ORC11 and RC11, we exploit the fact that ORC11 is very close to the “promise-free” fragment of Kang et al., extended with non-atomics and a race detector. Kang et al. already proved a formal correspondence between their promise-free fragment and C11. Building on their result, we show on paper that any racy execution in RC11 can be replayed as a racy execution in ORC11. The proof is relatively straightforward since ORC11 explicitly tracks read and write events.

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 $\lambda_{Rust}$ language and explain how to combine it with ORC11 to achieve our target language that will be used to instantiate Iris.
The Relaxed $\lambda_{\text{Rust}}$ Language

The relaxed $\lambda_{\text{Rust}}$ language retrofits the original RustBelt’s $\lambda_{\text{Rust}}$ on top of the relaxed memory semantics of ORC11. In this chapter, we briefly review $\lambda_{\text{Rust}}$ and formally define how to “plug” it in with ORC11 to obtain our target language.

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 $\lambda_{\text{Rust}}$ calculus only captures central features of the Rust language, and the original semantics assume a SC memory model, and we extend the operational semantics to cover relaxed memory accesses. Nevertheless, the reasoning principles we develop in this dissertation are not restricted to $\lambda_{\text{Rust}}$ or Rust, and can be applied to other languages that employ the RC11 memory model (or stronger memory models).

4.1 Language Syntax

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

- values that can be poison ($\bot$); 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 ($\pi$) 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 ($e(e)$) where the arguments are a list expressions; or a (switch) case block (case $e$ of $\tau$) 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; or an explicit allocation or deallocation instruction.

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

---

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

2 with a race detector that intuitively implements reader-writer locks on nonatomic locations.

3 see §3.2.

4 see Definition 2.2.
\[
v \in \text{Val} ::= \# | \ell | z | \text{rec } f(\overline{\tau}) := e
\]
\[
e \in \text{Expr} ::= v \mid x \mid e.e \\
| e + e \mid e - e \mid e \leq e \mid e == e \\
| e(\overline{\tau}) \\
| \text{case } e \text{ of } \overline{\tau} \\
| \text{fork } \{ e \} \\
| *^o e \mid e_1 := o \ e_2 \mid \text{CAS}^{\overline{\tau} \cdot \overline{o} \cdot \overline{w}}(e_0, e_1, e_2) \mid \text{fence}_o \\
| \text{alloc}(e) \mid \text{free}(e_1, e_2)
\]
\[
K \in \text{ECtx} ::= \bullet \\
| K.e \mid v.K \\
| K + e \mid v + K \mid K - e \mid v - K \mid K \leq e \mid v \leq K \mid K == e \mid v == K \\
| K(\overline{\tau}) \mid v(\overline{\tau} \cdot [K] \cdot \overline{\tau}) \\
| \text{case } K \text{ of } \overline{\tau} \\
| *^o K \mid K := o \ v \mid v := o \ K \\
| \text{CAS}^{\overline{\tau} \cdot \overline{o} \cdot \overline{w}}(K, e_1, e_2) \mid \text{CAS}^{\overline{\tau} \cdot \overline{o} \cdot \overline{w}}(v_0, K, e_2) \mid \text{CAS}^{\overline{\tau} \cdot \overline{o} \cdot \overline{w}}(v_0, v_1, K) \\
| \text{alloc}(K) \mid \text{free}(K, e_2) \mid \text{free}(e_1, K)
\]

Figure 4.1: The relaxed λRust 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 (λRust Syntactic Sugars). Several syntactic sugars are taken
as-is from the original RustBelt, given in Figure 4.2. Specifically:

- Non-recursive functions \( (\lambda[\overline{\tau}]. e) \) simply ignore the recursive function
  argument. \textbf{let} bindings are used to declare local variables in λRust, 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 \textbf{let} bindings.
\[ \lambda[\overline{e}]. \overline{e} := \text{rec}_{\overline{\lambda}}((\overline{\lambda}[\overline{x}]. \overline{e}')(\overline{\epsilon})) \]

\[
\text{let } \overline{x} = \overline{e} \text{ in } \overline{e'} := (\lambda[\overline{x}]. \overline{e}')(\overline{\epsilon})
\]

\[
e' \cdot \overline{e} := \text{let } \overline{e} = \overline{e} \text{ in } \overline{e'}
\]

\[
\text{false } := \overline{0}
\]

\[
\text{true } := \overline{1}
\]

\[
\text{if } \overline{e}_0 \text{ then } \overline{e}_1 \text{ else } \overline{e}_2 := \text{case } \overline{e}_0 \text{ of } [\overline{e}_2, \overline{e}_1]
\]

\[
\ast \overline{e} := \ast_{\overline{\text{na}}} \overline{e}
\]

\[
e := \text{let } \overline{e}_1 = \overline{e}_2 := \overline{na}_{\overline{e}_2}
\]

\[
\text{new } := \lambda[\overline{\text{size}}]. \text{if } \overline{\text{size}} == 0 \text{ then } \overline{(42, 1337)} \text{ else } \overline{\text{alloc}}(\overline{\text{size}})
\]

\[
\text{delete } := \lambda[\overline{\text{size}}, \overline{\text{ptr}}]. \text{if } \overline{\text{size}} == 0 \text{ then } \overline{\text{free}}(\overline{\text{size}}, \overline{\text{ptr}}) \text{ else free}(\overline{\text{size}}, \overline{\text{ptr}})
\]

\[
\text{memcpy } := \text{rec}_{\overline{\text{memcpy}}}([\overline{\text{dst}}, \overline{\text{len}}, \overline{\text{src}}]) :=
\]

\[
\text{if } \overline{\text{len}} \leq 0 \text{ then } \overline{\text{else}}
\]

\[
\overline{\text{dst}}.0 := \overline{\text{src}}.0;
\]

\[
\text{memcpy}([\overline{\text{dst}}.1, \overline{\text{len}} - 1, \overline{\text{src}}.1])
\]

\[
e := \overline{n} \cdot \overline{e}_2 := \text{memcpy}(\overline{e}_1, \overline{n}, \overline{e}_2)
\]

\[
e := \text{let } \overline{e}_1 := \overline{e}_2 := \overline{i}; \overline{e}_1.0 := \overline{i}; \overline{e}_1.1 := \overline{e}_2
\]

\[
\text{skip } := \text{let } \overline{x} = \overline{\text{in}} \overline{\text{skip}}
\]

\[
\text{newlft } := \overline{\text{if}}
\]

\[
\text{endlft } := \text{skip}
\]

\[
\text{skip } := \text{let } \overline{x} = \overline{\text{in}} \overline{\text{skip}}
\]

\[
\text{newlft } := \overline{\text{if}}
\]

\[
\text{endlft } := \text{skip}
\]

• We use 0 and 1 as boolean values \text{false} and \text{true}, respectively. This allows us to define \text{if}-branching using \text{case}: if \overline{e}_0 \text{ is false}, then the expression with index 0 in the list \([\overline{e}_2, \overline{e}_1]\), which is \overline{e}_2 is picked to be evaluated next; and if \overline{e}_0 \text{ is true}, then the expression with index 1, which is \overline{e}_1, is picked.

• We suppress the access modes for reads and writes if they are \text{na}.

• \text{new} and \text{delete}, unlike \text{alloc} and \text{free}, never get stuck when the provided block size is 0. \text{new} simply just returns some location in that case.

• \text{memcpy} copies \overline{\text{len}} cells from \overline{\text{src}} to \overline{\text{dst}} using the path operator. The “assign with length” notation \((\overline{e}_1 := \overline{n} \cdot \overline{e}_2)\) uses \text{memcpy}.

• There is no language primitive to define compound data structures. Instead, they can be implemented in memory using pointer arith-
letcont \( k(\pi) := e \) in \( e' \) := let \( k = (\text{rec} \ k(\pi) := e) \) in \( e' \)

jump \( k(\pi) := k(\pi) \)

funrec \( f(\pi) \) ret \( k := e := \text{rec} \ f([k] + \pi) := e \)

call \( f(\pi) \) ret \( k := f([k] + \pi) \)

Figure 4.3: CPS notations for \( \lambda_{Rust} \).

Continuations (meta-variable \( k \)) can be defined with letcont \( k(\pi) := e \) in \( e' \) where \( \pi \) is a list of binders that will be instantiated with the arguments \( \pi \) when a continuation is called with jump \( k(\pi) \).

CPS functions can be declared with funrec \( f(\pi) \) ret \( k := e \) where \( f \) is the binder for a recursive function, \( \pi \) 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 call \( f(\pi) \) ret \( k \) where \( \pi \) 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, which we do not need to worry about until Chapter 12 (Part II).

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).

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

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

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

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

**Definition 4.7 (Value Equality).**

\[
\mathcal{M} \vdash v_1 = v_2
\]

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

\[
\begin{align*}
\mathcal{M} \vdash z = z \\
\mathcal{M} \vdash \ell = \ell \\
\ell_1 \in \text{unalloc}(\mathcal{M}) \lor \ell_2 \in \text{unalloc}(\mathcal{M}) \\
\mathcal{M} \vdash \ell_1 = \ell_2
\end{align*}
\]

**Definition 4.8 (Value Inequality).**

\[
\vdash v_1 \neq v_2
\]

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

\[
\begin{align*}
z_1 \neq z_2 \\
\ell_1 \neq \ell_2 \\
\ell \neq 0 \\
\ell 
eq \ell
\end{align*}
\]

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).**

\[
\vdash v_1 =^? v_2
\]

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

\[
\begin{align*}
z_1 =^? z_2 \\
\ell_1 =^? \ell_2 \\
\ell =^? 0 \\
0 =^? \ell
\end{align*}
\]

**Definition 4.10 ($\lambda_{\text{Rust}}$ Expression Reductions).**

\[
\mathcal{M}, \mathcal{V} \vdash e \xrightarrow{\epsilon^?} e_1', e_2'
\]

The expression reduction relation $\mathcal{M}, \mathcal{V} \vdash e \xrightarrow{\epsilon^?} e_1', e_2'$ 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 $\epsilon^?$ and an optional expression $e_2'$ that will be running concurrently in a newly forked thread. Only memory operations will generate a memory event $\epsilon$, and only $\text{fork} \{ e_2' \}$ 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.
\[
\begin{align*}
\text{OE-CTX} & : e \vdash e' \quad e'^{\ell}
\hline
\mathcal{M}, \mathcal{V} \vdash K[e] \to K[e'] \quad \mathcal{M}, \mathcal{V} \vdash \ell.n \to \ell + \ell n
\end{align*}
\]

\begin{align*}
\text{OE-TRUE} & : z_1 \leq z_2 \quad \mathcal{M}, \mathcal{V} \vdash z_1 \leq z_2 \to \text{true} \\
\text{OE-FALSE} & : z_1 > z_2 \quad \mathcal{M}, \mathcal{V} \vdash z_1 > z_2 \to \text{false} \\
\text{OE-EQ} & : z_1 + z_2 = z' \quad \mathcal{M}, \mathcal{V} \vdash z_1 + z_2 = z' \\
\text{OE-SUB} & : z_1 - z_2 = z' \quad \mathcal{M}, \mathcal{V} \vdash z_1 - z_2 = z'
\end{align*}

\begin{align*}
\text{OE-LE} & : \lambda \quad \mathcal{M}, \mathcal{V} \vdash \lambda \to f\neg \quad \mathcal{M}, \mathcal{V} \vdash v_1 = v_2 \to \text{true} \\
\text{OE-APPL} & : \mathcal{M}, \mathcal{V} \vdash (\text{rec } f(\pi) := e)(\pi) \to e[\text{rec } f(\pi) := e/f, \pi/\pi]
\end{align*}

\begin{align*}
\text{OE-CASE} & : 0 \leq i < |\pi| \quad \mathcal{M}, \mathcal{V} \vdash \text{case } i \text{ of } \pi \to \pi_i \\
\text{OE-FORK} & : \mathcal{M}, \mathcal{V} \vdash \text{fork } \{ e \} \to \bullet, e \\
\text{OE-READ} & : \mathcal{M}, \mathcal{V} \vdash ^* \ell \xrightarrow{\kappa'(\ell,v)} v \\
\text{OE-FREE} & : \mathcal{M}, \mathcal{V} \vdash \ell := v \xrightarrow{\omega' \ell, v} \bullet
\end{align*}

\begin{align*}
\text{OE-CAS-FAIL} & : \text{rli} \subseteq o_f, o_r, o_{ow} \quad \forall \omega \in \text{Readable}(\ell, \mathcal{M}, \mathcal{V}). \exists v', \omega \equiv v' \land v_1 = v' \land v_1 \neq v_r \\
\mathcal{M}, \mathcal{V} \vdash \text{CAS}^{\text{rli}, o_f, o_{ow}}(\ell, v_1, v_2) \xrightarrow{\kappa'\ell(v_r, v_2)} \text{false}
\end{align*}

\begin{align*}
\text{OE-CAS-SUCC} & : \text{rli} \subseteq o_f, o_r, o_{ow} \quad \forall \omega \in \text{Readable}(\ell, \mathcal{M}, \mathcal{V}). \exists v', \omega \equiv v' \land v_1 = v' \land v_1 = v_r \\
\mathcal{M}, \mathcal{V} \vdash \text{CAS}^{\text{rli}, o_f, o_{ow}}(\ell, v_1, v_2) \xrightarrow{\omega'\ell, v_r, v_2} \text{true}
\end{align*}

\begin{align*}
\text{OE-FENCE} & : \mathcal{M}, \mathcal{V} \vdash \text{fence}_{\ell} \xrightarrow{v} \bullet \\
\text{OE-ALLOC} & : n > 0 \quad \mathcal{M}, \mathcal{V} \vdash \text{alloc}(n) \xrightarrow{n} \ell \\
\text{OE-FREE} & : n > 0 \quad \mathcal{M}, \mathcal{V} \vdash \text{free}(n, \ell) \xrightarrow{n} \bullet
\end{align*}

\textbf{Figure 4.4:} Relaxed $\lambda_{\text{Rust}}$ expression semantics.

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

- **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 ($\bullet$), 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 $\tau_i$ will

\(^8\)see Definition 4.7 and Definition 4.8.
be picked to reduce. Note that no expression in the list $\pi$ is reduced before the \texttt{case} is reduced.

- \texttt{OE-app} says that function application reduces once all arguments have been evaluated to a list of values $\pi$.\footnote{Recall that by the Definition 4.2 for evaluation contexts, the arguments are evaluated left-to-right.} It is also required that the list of binders and the list of arguments have the same length. Then, the reduction \textit{substitutes} the arguments for the binders, including the recursive function binder $f$, in the function’s body.

- \texttt{OE-fork} says that $\text{fork\{}e\text{\}}$ 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).

- \texttt{OE-read} says that a read simply reduces to the value $v$ that comes with the read event that $R^o(\ell,v)$ 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).

- \texttt{OE-write} says that a write reduces to poison and is tied to the corresponding write event.

- A compare-and-swap instruction $\text{CAS}^{o_f,o_r,o_w}(\ell,v_1,v_2)$ takes three \textit{atomic} access modes: $o_f$ is the order that will be used when the \texttt{CAS} fails, in which case it acts like a read with the mode $o_f$; otherwise, if the \texttt{CAS} succeeds, then it acts as both a read with the mode $o_r$ and a write with the mode $o_w$. The \texttt{CAS} \textit{atomically} (i) reads the location $\ell$, (ii) compares the value read $v_r$ with $v_1$, and (iii) if the values are equal, writes $v_2$ to $\ell$. \texttt{OE-CAS-fail} and \texttt{OE-CAS-succ} therefore both require that for any memory value $\omega$ \textit{readable}\footnote{\textit{i.e.}, those values that are not yet overshadowed by $V.cur(\ell)$—see Definition 4.6.} by the \texttt{CAS}, its injected value $v'$ must be comparable\footnote{see Definition 4.9.} with $v_1$.

  - In case of failure, \texttt{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 \texttt{false} and is tied to the read event $R^o_i(\ell,v_r)$.

  - In case of success, \texttt{OE-CAS-succ} says that it must be that $v_1$ is equal to the read value $v_r$, and the reduction reduces to \texttt{true} and is tied to the update event $U^{o_r,o_w}(\ell,v_r,v_2)$.

Note again that this means that the \texttt{CAS} can non-deterministically fail or succeed if $v_1$ and what $\ell$ stores can be unallocated location values.

- \texttt{OE-fence} says that a fence also reduces to poison and is tied to the corresponding fence event.

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

\textit{Note 4.11.} In the expression reductions, the global memory $M$ is used in equality comparison (\texttt{OE-EQ-true}). Other than that, it is only used
The combined 1-thread semantics of ORC11 machine semantics and \( \lambda_{\text{Rust}} \) expression semantics is given in Figure 4.5. The configuration \((M, N) | (e, V)\) is called a 1-thread configuration which includes the thread's executing expression \(e\), the thread-view \(V\), and the global state \(M\) and \(N\). The pair \((e, V)\) 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 \(\varepsilon\) and an optional pair of expression and thread-view \((e_f, V_f)\) 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 \(V_f\), derived from \(V\), is picked for the newly forked thread. The definition of \(\text{ForkView}(V)\) encodes our choice for \textit{fork}'s behaviors with respect to synchronization: the forked (child) thread should be synchronized with the forking thread, but a \textit{fork} does not act as a release fence for the forked thread, so its release views are empty (\(\emptyset\)).

- **OC-MEM** allows for a memory step that simultaneously (i) reduces the expression \(e\) in one step to \(e'\) \((M, V \vdash e \overset{\varepsilon}{\Rightarrow} e', \bot, \text{Definition 4.10})\) with a memory event \(\varepsilon\), and (ii) makes a view-based machine step \((M | V \overset{e', r, ms}{\Rightarrow} M' | V', \text{Definition 3.22})\) and a race detector step \((N \overset{e', r, ms}{\Rightarrow} N')\,\text{Definition 3.26}) with the same memory event \(\varepsilon\), 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 \(\varepsilon'\), it must be that \((M, N, V) \vdash \text{RaceFree}(\varepsilon')\) (Definition 3.25).
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 $T$ is a partial, finite map from thread-ids to pairs of expressions and thread-views, i.e.,

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

**Definition 4.14 (Threadpool Reductions).**

$$\varsigma \mid T \rightarrow \varsigma' \mid T'$$

The thread-pool semantics is given in Figure 4.6. It defines the reduction of a thread-pool configuration $\varsigma \mid T$ 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 $\pi$ in the thread-pool and make a 1-thread step using the 1-thread configuration $\varsigma \mid (e, V)$ for the global state and thread $\pi$. The results of the 1-thread step are then used to update the thread-pool configuration. In case thread $\pi$ forks a new thread with $(e_f, V_f)$, then a fresh thread-id $\rho \notin \text{dom}(T)$ is picked to insert the newly forked thread into the thread-pool.

**Chapter Summary.** This chapter presents the $\lambda_{Rust}$ 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.
More Background: Iris, A Framework for Concurrent Separation Logics

In this chapter, we give a quick review of the concurrent separation logic framework Iris. Readers who are familiar with Iris can skip this review, and jump to Chapter 6 for the instantiation of Iris with our relaxed \( \lambda_{\text{Rust}} + \text{ORC11} \) 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 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) that supports interactive resource reasoning and that can be instantiated with any BI logics.

An excerpt of Iris grammar is given in Figure 5.1. Propositions in Iris belong to the type \( \text{iProp} \), which has a step-indexing model on resources.

Concept 5.1 (Iris Base Logic). The Iris base logic supports the common logical connectives (\( \text{False}, \text{True}, \Rightarrow, \land, \lor \)). \( \text{iProp} \) propositions are to be interpreted with resources in mind, so the “classical” conjunction \( P \land Q \) should be read as \( P \) and \( Q \) hold relying on the same resources. The base logic logic allows embedding pure facts \( \phi \) which exist at the meta-level and which naturally do not occupy resources.

- It is a BI logic: the separating conjunction \( P \star Q \) says that \( P \) and \( Q \) hold on disjoint resources, and the wand implication \( P \rightarrow Q \) 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 (\( \mu \)) because the quantified variable \( x \) can also be an \( \text{iProp} \). 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 until the program takes a step (and thus decrement the step-index). The step-indexing model guarantees the logic’s soundness in the presence of recursive higher-order quantifications, impredicative invariants, and higher-order ghost state.

1Jung 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].

2O’Hearn and Pym, “The logic of bunched implications” [OP99]; Ish-tiaq and O’Hearn, “BI as an Assertion Language for Mutable Data Structures” [IO01].

3derived from the base logic


5[Jun+18b], §4.
$P \in \text{iProp} ::= \begin{array}{l}
(* \text{base logic} *) \\
| \phi | \text{False} | \text{True} | P \Rightarrow Q | P \wedge Q | P \vee Q | P \ast Q | P \rightarrow Q | \exists x. P | \forall x. P | \mu x. P \\
| \not P | \not\not P | [\not a : M]^{\gamma} | \not\Rightarrow P | \ldots
\end{array}
(* \text{program logic} *)
| [\not P]^{N} | \not\not P | \not\Rightarrow P

**Figure 5.1:** An excerpt of Iris grammar.

- The **persistent** modality $\square P$ says that $P$ is known to hold without some exclusive resource, so that $P$ can be freely duplicated, i.e., $P \Rightarrow P \ast P$.
- The proposition $[\not a : M]^{\gamma}$ asserts the ownership of an element $a$ of a resource algebra $M$ for the ghost location $\gamma$. In case the resource algebra $M$ is clear in context, we simply write $[\not a]^{\gamma}$.
- The **basic update** modality $\Rightarrow P$ hides away some **ghost updates** that can be performed to achieve $P$.

**Concept 5.2** (Iris Program Logic). To support program verification for a language, several constructions can be derived from the base logic:

- $[\not P]^{N}$ asserts the existence of some global **invariant** that holds the resource $P$, under some **invariant name** that is in the namespace $N$. Namespaces provide some hierarchy to sets of invariant names.
- The **fancy update** $\not\not P \Rightarrow \not\not P$ 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** $E_1$ and $E_2$ are sets of invariant names that identify the invariants that hold (unopened) before and after the update, respectively.
- The **weakest pre-condition** $\text{wp}_e e \{v. Q\}$ assert the resources needed for $e$ to safely execute and maintain the invariants in $E$ 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.

**5.1 Basic Rules**

**Figure 5.2** 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 $P \vdash Q$ stands for $P \Rightarrow Q$ and $Q \vdash P$.

The rules in **Figure 5.2** 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
we can always get $P$ from $\square P$; or \textsc{later-intro} says that if we have $P$ now, we also have $P$ in the next step; or \textsc{lob} 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.

5.2 Ghost State and Resource Algebras

The ghost ownership assertion \textit{(see page 57)} 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 \textit{partial commutative monoids} (PCMs).

\textbf{Definition 5.3} (Resource Algebras). A resource algebra (RA) is a tuple 
\[ (M, \text{valid} : M \to \text{Prop}, | - | : M \to M^\tau, (\cdot) : M \times M \to M) \]
where the type $M$ has a commutative, associative \textit{composition} $(\cdot)$; a \textit{core} function $(| - |)$ that computes a optional \textit{per-element} unit (in $M^\tau$) for each element $a \in M$; and a \textit{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 $\varepsilon$, an RA can have multiple per-element units $|a|$, and some elements may not have a unit, \textit{i.e.,} $|a| = \bot$.

The following properties must hold for an RA.

\begin{align*}
\forall a, b. a \cdot b &= b \cdot a & (\text{RA-comm}) \\
\forall a, b, c. (a \cdot b) \cdot c &= a \cdot (b \cdot c) & (\text{RA-assoc}) \\
\forall a. |a| \in M \Rightarrow |a| \cdot a &= a & (\text{RA-core-id}) \\
\forall a. |a| \in M \Rightarrow |a| &= a & (\text{RA-core-idemp}) \\
\forall a. |a| \in M \land a \preceq b \Rightarrow |b| \in M \land |a| \preceq |b| & (\text{RA-core-mono}) \\
\forall a, b. \text{valid}(a \cdot b) \Rightarrow \text{valid}(a) & (\text{RA-valid-op})
\end{align*}

where
\begin{align*}
a \cdot \bot &= \bot \cdot a = a \\
\forall a, b. a \preceq b :&= \exists c \in M. b = a \cdot c & (\text{RA-incl})
\end{align*}
More Background: Iris, A Framework for Concurrent Separation Logics

**Figure 5.3:** Basic rules of Iris ghost ownership and basic updates.

- **Ghost-alloc**
  \[
  \text{valid}(a) \Rightarrow \exists \gamma . \text{˙} | \gamma = a
  \]

- **Ghost-valid**
  \[
  \begin{align*}
  & \overline{\gamma \gamma} \Rightarrow \text{valid}(a) \\
  & \gamma \cdot \gamma \Leftrightarrow \overline{\gamma \gamma} \cdot \overline{\gamma \gamma}
  \end{align*}
  \]

- **Ghost-op**
  \[
  \begin{align*}
  & a \leadsto b \Rightarrow \exists \gamma . \text{γ} = b \\
  & \overline{\gamma \gamma} \Rightarrow \exists b \in B . \overline{\gamma \gamma} = b
  \end{align*}
  \]

- **Ghost-update-gen**
  \[
  \begin{align*}
  & P \vdash Q \Rightarrow P \vdash \text{˙} | Q \\
  & \text{Budp-mono} \quad P \vdash \text{˙} | P \\
  & \text{Budp-intro} \quad P \vdash P \Rightarrow P \\
  & \text{Budp-trans} \quad Q \cdot \text{˙} | P \vdash \text{˙} | (Q \cdot P)
  \end{align*}
  \]

- **Ghost-update**
  \[
  \begin{align*}
  & a \leadsto b \Rightarrow \exists \gamma . \text{γ} = b \\
  & \overline{\gamma \gamma} \Rightarrow \overline{\gamma \gamma}
  \end{align*}
  \]

\[P \Rightarrow Q := \Box (P \rightarrow \text{˙} | Q)\]

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 \(\epsilon\) satisfying:

\[
\text{valid}(\epsilon) \quad \forall a \in M . \epsilon \cdot a = a \quad |\epsilon| = \epsilon
\]

The rules for ghost ownership are given in Figure 5.3. Ghost-alloc lets us allocate a fresh ghost location \(\gamma\) 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 \(\gamma\), and this update is hidden in the basic update modality \(\Rightarrow\). 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 5.4 (Basic Viewshifts).**

Intuitively, basic viewshift \(P \Rightarrow Q\) says that the resources owned by \(P\) can be turned into the resources owned by \(Q\) using some ghost updates.

\[P \Rightarrow Q := \Box (P \rightarrow \text{˙} | Q)\]

**Ghost-update-gen** and **Ghost-update** allow us to update some ghost state, using the basic viewshift. **Ghost-update-gen** allows for a 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 5.5 (Frame-preserving Ghost Updates).**

When doing a ghost update for \(\overline{\gamma \gamma}\), one must remember that one only has a part of the ghost location \(\gamma\): with separation (i.e., using Ghost-op), other parts of \(\gamma\) 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 \(a \leadsto b\) 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 ⇝ b is derived from the more general definition a ⇝ B which says that a can be frame-preservingly updated to any element in B:

\[ a \mapsto B ::= \forall c^3 \in M^3. \text{valid}(a \cdot c^3) \Rightarrow \exists b \in B. \text{valid}(b \cdot c^3) \]  

(RA-FRAME-UPD-GEN)

\[ a \mapsto b ::= a \mapsto \{ b \} \]  

(RA-FRAME-UPD)

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

Figure 5.3 also provides some structural rules for basic updates.

5.3 Invariants and Fancy Updates

Invariants can be seen as logical, global spaces where resources can be stored for concurrent accesses. 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.

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 §5.7. Here, we look at the vanilla rules of Iris invariants.

The proposition \( I^N \) asserts the existence of I in the global invariant space with some invariant name ι in the namespace N. Several rules for Iris invariants are given in Figure 5.4, which rely on fancy updates, which in turn generalize basic updates with masks. Intuitively, \( \mathcal{E}_1 \models \mathcal{E}_2 \) 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.

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 \( N \)—meaning that the invariants in \( N \) are not opened yet, then we can open all invariants in \( N \) with the fancy update \( \mathcal{E}_1 \models \mathcal{E}_2 \setminus N \) (so after that only the invariants in \( \mathcal{E}_2 \setminus N \) are enabled).

- Furthermore, if we know that \( I^N \), i.e., I is stored in one of those invariants in \( N \) 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 \( I^N \) itself.

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

\[ \text{HD: explain atomic} \]
Finally, we also get a “closing” update, \( \triangleright I \stackrel{\mathcal{N}}{\Rightarrow} \triangleright \), 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 \( \mathcal{N}_1 \neq \mathcal{N}_2 \subseteq \mathcal{E} \). Then we can apply \( \text{INV-ALLOC} \) twice, first with \( I_1 \mathcal{N}_1 \) and \( \mathcal{E} \), then with \( I_2 \mathcal{N}_2 \) and \( \mathcal{E} \setminus \mathcal{N}_2 \), to get access to \( \triangleright I_1 \triangleright I_2 \).

\( \text{INV-ALLOC} \) allows us to allocate an invariant \( I \) with some fresh invariant name \( \iota \) picked non-deterministically from \( \mathcal{N} \). Note that we only need to provide \( I \) under a later, and that \( \triangleright \iota \) 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 5.4** also provides some structural rules for fancy updates. Importantly, \( \text{FUPD-BUPD} \) says that a fancy update includes a basic update.

**Notation 5.6** (Fancy Updates and Wand Viewshifts). \( \triangleright \iota \) denotes non-mask-changing fancy updates \( \mathcal{E} \triangleright \mathcal{E} \), and \( P \mathcal{E} \triangleright \mathcal{E} Q \) denotes wand viewshifts, a combination of wand implication and fancy update:

\[
P \mathcal{E} \triangleright \mathcal{E} ^{E} Q := P \rightarrow \mathcal{E} \triangleright \mathcal{E} ^{E} Q
\]

\( P \triangleright \mathcal{E} Q \) denotes non-mask-changing wand viewshifts.

Plain viewshifts are the persistent version of wand viewshifts:

\[
P \mathcal{E} \triangleright \mathcal{E} ^{E} Q := \Box (P \rightarrow \mathcal{E} \triangleright \mathcal{E} ^{E} Q)
\]

\[
P \triangleright \mathcal{E} Q := \Box (P \rightarrow \triangleright \mathcal{E} Q)
\]
5.4 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 λRust language in Chapter 6.) Intuitively, the proposition wpE 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 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 5.7 (Iris Hoare Triples). Hoare triples in Iris are defined in terms of weakest pre-conditions:

\{P\} e {v, Q}_E := □(P \rightarrow \text{wp}_E e \{v, Q\})

The persistent modality (□) 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 E, and if e terminates to v then Q holds.

5.5 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 5.8 (Iris Adequacy). If ⊬ wp_τ e \{v, φ(v)\} is derivable in the Iris program logic for Λ where φ(v) is a pure (meta-level) fact, then the following holds.

∀π, T, σ. ([π ↦ e], σ_{init}) \rightarrow^* (T, σ) \Rightarrow

∀v. T(π) = v ⇒ φ(v)  \quad \text{(IRIS-ADEQUACY-VAL)}

∧ ∀p, e_p. T(p) = e_p ⇒ e_p is a value ∨ red(e_p, σ)  \quad \text{(IRIS-ADEQUACY-NO-STUCK)}

That is, if we start running e with the initial global configuration ([π ↦ e], σ_{init})—where the threadpool only has a single thread π with
the expression $e$ and the initial global state $\sigma_{\text{init}}$ is typically empty—then for any configuration reachable $(T, \sigma)$ through the reflexive, transitive closure of the threadpool reduction $\rightarrow_{\Lambda}$ for $\Lambda$, (\textsc{iris-adequacy-val}) if thread $\pi$ has reduces to a value $v$, the pure fact $\phi(v)$ must hold, and (\textsc{iris-adequacy-no-stuck}) for any thread $\rho$ with the expression $e_\rho$, it is not stuck: either $e_\rho$ is a value, or $e_\rho$ is still reducible on $\sigma$ ($\text{red}(e_\rho, \sigma)$).

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 $\vdash \wp e \{v. \phi(v)\}$ is the same as deriving a Hoare triple with a trivial pre-condition, i.e., $\vdash \{\text{True}\} e \{v. Q\} \top$.

5.6 Some Common Rules for WPs and Hoare Triples

On the other hand, during program verification, in order to derive $\wp e \{v. Q\}$, 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 $\Lambda$, we need to extend the logic with WP rules for the primitive instructions of $\Lambda$. We will see those primitive rules for our relaxed $\lambda$Rust + ORC11 language in Chapter 6. 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 5.5. Note that when reading the WP rules, by reading them backward—as typically when we use a rule of form $\wp e \{v. \phi(v)\} \vdash \wp e \{\phi\}$ to turn the verification goal $\wp e \{\phi\}$ into the goal $\wp e \{v. \phi(v)\}$, 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 $\Phi$ and $\Psi$ for predicates on values ($\text{Val} \rightarrow \text{iProp}$) 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 $\Psi$ we may want to prove a WP with the stronger post-condition $\Phi$. It also additionally provides monotonicity for masks: if we can prove a WP relying on fewer invariants (with the smaller mask $E_1$), then that WP also works with more invariants (with the bigger mask $E_2$).\footnote{So that from a $\wp e \{\Phi\}$ one can, for example, obtain $\wp e \{\Phi\}$ with the top ($\top$) mask, and subsequently apply adequacy.} The well-known \textsc{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 $K$. \textsc{hoare-bind} is derivable from WP-bind.
Weakest Pre-conditions and Invariants

The weakest pre-conditions and Hoare triples. 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 \( \Phi \). 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 \( \rho \) (with the expression \( e \)) with a trivial post-condition. The trivial post-condition signifies that forked threads are “detached” by the default.\(^{12}\) 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.

5.7 Weakest Pre-conditions and Invariants

Figure 5.6 provides some rules for the interaction between WPs (Hoare triples) and invariants and fancy updates.
WP-FUPD
\[ \vdash_{E} wp_{E} e \{ v \vdash_{E} \Phi(v) \} \vdash wp_{E} e \{ \Phi \} \]

WP-atomic
\[ \vdash_{E}^{\Phi} wp_{E}^{\Phi} e \{ v \vdash_{E}^{\Phi} \Phi(v) \} \vdash wp_{E}^{\Phi} e \{ \Phi \} \]

WP-INV
\[ \vdash I \vdash wp_{E \setminus N} e \{ v \vdash I \Phi(v) \} \quad \text{atomic(e)} \quad N \subseteq E \]
\[ \vdash \{ P \} e \{ v \vdash Q \} \]

Hoare-vs
\[ P \vdash_{E} P' \quad \{ P' \} e \{ v \vdash Q \} \quad \forall v. Q' \vdash_{E} Q \]
\[ \{ P \} e \{ v \vdash Q \} \]

Hoare-inv
\[ \vdash I \Phi \vdash wp_{E \setminus N} e \{ v \vdash I \Phi(v) \} \quad \text{atomic(e)} \quad N \subseteq E \]
\[ \vdash \{ P \} e \{ v \vdash Q \} \]

where \( P \vdash_{E} Q := \square(P \rightarrow \vdash_{E} Q) \) (see Notation 5.6).

Figure 5.6: Some rules for Iris weakest pre-conditions and invariants.

- **WP-FUPD** says that we can perform fancy updates around an expression if our goal is a WP. Combining this rule with **FUPD-MONO** and **FUPD-FRAME** (Figure 5.4), around a WP we can eliminate any hypotheses with fancy updates in our proof context if they have masks smaller than \( E \). 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 \( wp_{E} e \{ \Phi \} \), we can perform a mask-changing fancy update from the mask \( E_{1} \) to the \( E_{2} \), and prove a WP for the mask \( E_{2} \) whose post-condition must perform a reverse mask-changing update from \( E_{2} \) back to \( E_{1} \) 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 \( E_{1} = E \), and \( E_{2} = E \setminus N \), we can use **WP-atomic** and then **INV-ACC** to open the invariant \( \{ P \} \) around a goal of \( wp_{E} e \{ \Phi \} \), and use \( \vdash I \) for the execution of the atomic expression \( e \), if we can re-establish \( \vdash I \) after the step. **HOARE-INV** is easily derivable from **WP-INV**.

5.8 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 5.7.

**Property 5.9** (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 as in Timeless-sep. Some typical timeless propositions are pure (metalevel) facts (Timeless-pure), and ownership of a ghost element whose RA is discrete (Timeless-discrete), i.e., the equivalence relation is not step-indexed.  

\[ \text{Property 5.10 (Persistent Propositions).} \] 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).

5.9 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.\(^{14}\) 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.

\[ \text{Definition 5.11 (The Authoritative Resource Algebra).} \] The authoritative RA,\(^{15}\) denoted \( \text{AUTH}(M) \), 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 5.8.

\(^{13}\)A step-indexing family of equivalence relations is needed for higher-order ghost state.

\(^{14}\)Jensen and Birkedal, “Fictional Separation Logic” [JB12].

\(^{15}\)Jung et al., “Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning” [Jun+15].
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 5.12 (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 \( \bullet a_1 \circ b_1 \) can frame-preservingly updated to \( \bullet a_2 \circ b_2 \) if \( (a_1, b_1) \) can be locally updated to \( (a_2, b_2) \).

**Definition 5.13 (Local Updates).**

A pair \( (a_1, b_1) \) can be locally updated to a pair \( (a_2, b_2) \), if for any (optional) frame \( a_i^7 \) completing \( b_1 \) to \( a_1 \), \( a_i^7 \) also complete \( b_2 \) to \( a_2 \):

\[
(a_1, b_1) \rightsquigarrow (a_2, b_2) :=
\forall a_i^7 \cdot \text{valid}(a_1) \land a_1 = b_1 \cdot a_i^7 \Rightarrow \text{valid}(a_2) \land a_2 = b_2 \cdot a_i^7
\]

In the case of AUTH-UPDATE, this means that the frame \( a_i^7 \) 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 \( a_i^7 \).
The Physical State Interpretation 67

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

5.10 The Physical State Interpretation

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

**Definition 5.14** (Iris WP, simplified).

\[
\begin{align*}
\text{wp}_S^e \{ \Phi \} &:= \\
&= e \in \text{Val} \land \exists \Phi(e) \\
&\lor \left( e \notin \text{Val} \land \forall \sigma. S(\sigma) \rightarrow \\
&\quad e \xrightarrow{\sigma} (\text{red}(e, \sigma) \land \forall e', \sigma', e_\ell. (e, \sigma) \rightarrow (e', \sigma', e_\ell) \rightarrow \\
&\quad \lor \forall \sigma' S(\sigma') \land \text{wp}_S^{e'} \{ \Phi \} \land \text{wp}_S^\top \{ \text{v. True} \}) \right)
\end{align*}
\]

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 \( \sigma \) before a step, (i) \( e \) must be safe to take a 1-thread reduction step \( (\rightarrow_t) \) in \( \sigma \), i.e., \( e \) is reducible in \( \sigma \) \( (\text{red}(e, \sigma)) \), and (ii) for any resulting configuration \( (e', \sigma', e_\ell) \) of such a 1-thread reduction step, the state interpretation \( S(\sigma') \) for the physical state \( \sigma' \) after the step must hold, and the weakest pre-conditions hold recursively for \( e' \) and the forked expression \( e_\ell \). 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 \( \sigma \in \text{State} \) 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 \( \sigma \). Specifically, we will use \( S \) to keep the current state \( \sigma \) in sync with our authoritative ghost copy \( \bullet \sigma \), and give out the fragmentary element \( o \sigma \)—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 \( \text{State} \) 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).

5.11 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., \( \text{State} := \text{Loc} \xrightarrow{\text{fin}} \text{Val} \).\(^{16}\) Let us call this language

\(^{16}\)This example is rephrased from [Jun+18b] and [Kai+17].
\( \lambda_{HEAP} \). What we want is to derive the small-footprint rules for reads and writes, using the local points-to assertion.

\[
\begin{align*}
\text{HEAP-READ} & \quad \{ \ell \mapsto v \} \{ w, w = v \} \top \quad \{ \ell \mapsto v \} \ell := v' \{ (), \ell \mapsto v' \} \top \\
\text{HEAP-WRITE} & \quad \{ \ell \mapsto v \} \end{align*}
\]

To do so, we pick a suitable RA to split a heap \( \sigma \) into multiple singleton heaps of the form \( [\ell \mapsto v] \), 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, valid\( (\sigma \cdot \sigma') = \text{dom}(\sigma) \cap \text{dom}(\sigma') = \emptyset \). The state interpretation and the points-to assertion are then defined as:

\[
S(\sigma) := \bullet \sigma : \text{AUTH}(\text{SHEAP}) \gamma_{HEAP} \\
\ell \mapsto v := \otimes [\ell \mapsto v] : \text{AUTH}(\text{SHEAP}) \gamma_{HEAP}
\]

That is, the state interpretation \( S(\sigma) \) is the ghost ownership of the authoritative heap \( \bullet \sigma \) at the ghost location \( \gamma_{HEAP} \), and the points-to assertion \( \ell \mapsto v \) is the ghost ownership of the fragmentary singleton heap \( \circ \{ \ell \mapsto v \} \) at the same ghost location \( \gamma_{HEAP} \). The ghost location \( \gamma_{HEAP} \) is allocated before the program runs (a proof that indeed needs to be done in Adequacy—see §5.5), and is needed to establish the agreement between \( \ell \mapsto v \) and the current physical state \( \sigma \), indirectly through \( S \). To see this in action, let us look at the proofs of \text{HEAP-READ} and \text{HEAP-WRITE}. Both proofs first proceed by unfolding the definitions of Hoare triples (§5.4, Definition 5.7), and WP (Definition 5.14).

**Proof sketch of \text{HEAP-READ}**. After introducing the assumptions and clearing the fancy update (using \text{FUPD-INTRO-MASK} and \text{FUPD-MONO}),\(^{17}\) we arrive at the following goal.

<table>
<thead>
<tr>
<th>Context:</th>
<th>Goal:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ell \mapsto v ) ( (\equiv \bullet \sigma : \text{AUTH}(\text{SHEAP}) \gamma_{HEAP}) ) ( \equiv S(\sigma) ) ( (\equiv \bullet \sigma : \text{AUTH}(\text{SHEAP}) \gamma_{HEAP}) ) ( \text{red}(\ell, \sigma) \land \forall e', \sigma', e_1, \ldots )</td>
<td></td>
</tr>
</tbody>
</table>

We first need to show that \( e \) is reducible on \( \sigma \).

With \text{GHOST-OP} and \text{GHOST-VALID} (see §5.2), from our assumptions, we have valid\( (\bullet \sigma \cdot \circ \{ \ell \mapsto v \}) \), and then by \text{AUTH-BOTH-VALID}, we have \( [\ell \mapsto v] \leq \sigma \) and valid\( (\sigma) \), so \( \sigma = [\ell \mapsto v] \sqcup \sigma_\ell \) for some \( \sigma_\ell \).

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

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

\[
\ell \mapsto v * S(\sigma) \quad \forall e', \sigma', e_1, (\bullet \ell, \sigma) \rightarrow_\ell (e', \sigma', e_1) \Rightarrow \sigma^T_{\top} \ldots
\]

Looking at the operational semantics of \( (\bullet \ell, \sigma) \rightarrow_\ell (e', \sigma', e_1) \), we learn that \( e' = \sigma(\ell) \land \sigma' = \sigma \land e_1 = \bot \), so our goal is

\[
\ell \mapsto v * S(\sigma) \quad \sigma^\top_{\top} S(\sigma) \ast \text{wp}_\mathbb{S}^\ell \sigma(\ell) \{ w, w = v \ast \ell \mapsto v \}
\]

After clearing the later and fancy update modalities and simplifying (using \text{LATER-INTRO} and \text{FUPD-MONO} again),

\[
\ell \mapsto v * S(\sigma) 
\]

Since we already know \( \sigma(\ell) = v \), we are done. \( \square \)
Note that to apply \texttt{GHOST-OP} in the above proof, it is important that 
\( S(\sigma) \) and \( \ell \mapsto v \) use the same ghost location \( \gamma_{\text{HEAP}} \).

**Proof sketch of \texttt{HEAP-WRITE}**. Similar to the proof of \texttt{HEAP-WRITE}, by owning \( \ell \mapsto v \) we know that \( \sigma(\ell) = v \) where \( \sigma \) is the current physical state, so we can prove \( \text{red}(\ell := v', \sigma) \). We then introduce all assumptions and clear the fancy update and the later from the goal. However, we additionally need to use \texttt{FUPD-TRANS} to keep a fancy update \( \models \top \): 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.

<table>
<thead>
<tr>
<th>Context:</th>
<th>Goal:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ell \mapsto v * S(\sigma) )</td>
<td>( \models \top S(\sigma') * \wp^S_e { (\ell \mapsto v') } \ldots )</td>
</tr>
</tbody>
</table>

From the operational semantics of \( \rightarrow_t \), we learn that
\( e' = () \land \sigma' = \sigma[\ell \leftarrow v'] \land e_f = \perp \), so our goal is

\[
\ell \mapsto v * S(\sigma) \models \top S(\sigma') * \wp^S_e (\{ (\ell \mapsto v') \})
\]

After simplifying, we arrive at

\[
\ell \mapsto v * S(\sigma) \models \top S(\sigma') * \ell \mapsto v'
\]

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

\[
\bullet \sigma \circ [\ell \leftarrow v] \rightsquigarrow \bullet \sigma' \circ [\ell \leftarrow v']
\]

Applying \texttt{AUTH-UPDATE}, we have to show

\[
(\sigma, [\ell \leftarrow v]) \rightsquigarrow (\sigma', [\ell \leftarrow v'])
\]

This is easy. We know that the frame \( \sigma_f \) that completes \( \sigma \) with \( [\ell \leftarrow v] \) is disjoint from \( [\ell \leftarrow v] : \sigma = \sigma_f \uplus [\ell \leftarrow v] \). Thus we can show

\[
(\sigma_f \uplus [\ell \leftarrow v], [\ell \leftarrow v]) \rightsquigarrow (\sigma_f \uplus [\ell \leftarrow v'], [\ell \leftarrow v'])
\]

easily by looking at the definition of local updates (Definition 5.13). We are done because \( \sigma' = \sigma[\ell \leftarrow v'] = \sigma_f \uplus [\ell \leftarrow v'] \). \( \square \)
A Base Logic for RMC in Iris

In this chapter, we demonstrate how to instantiate the Iris framework with our \( \lambda_{\text{Rust}} + \text{ORC11} \) 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 5. 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. §5.11 already gives an instantiation example for a language with simple heaps, but it worths articulating the process, more specifically for our \( \lambda_{\text{Rust}} + \text{ORC11} \) semantics:

1. We instantiate Iris with the 1-thread reductions (Definition 4.12) of \( \lambda_{\text{Rust}} + \text{ORC11} \). Since we are in a relaxed memory semantics with views, the resulting base logic will have to expose them. §6.1 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. §6.2 discusses the design (interfaces) of those new assertions.

3. §6.3 presents the desired small-footprint, primitive rules that use the new local assertions.

4. §6.4 presents the resource algebras needed to give a model for the new local assertions and proves their properties.

5. Finally, §6.5 defines the state interpretation \( S \) for our language, and §6.6 provides proofs of some primitive rules as well as adequacy.

6.1 Thread-local Configurations as Expressions

The Iris framework requires as input a language with a reduction relation \((e, \sigma) \rightarrow_t (e', \sigma', e_f)\)—which we call a 1-thread reduction—where \( e \) is the expression of the thread being evaluated and \( \sigma \) is the physical machine
threads, so e state. The resulting RMC base logic

\[
\text{BL-WP-BIND}\quad \wp_E (e, \nu) \{ (v, \nu_v) \}, \wp_E (K[v], \nu_v) \{ \Phi \} \vdash \wp_E (K[e], \nu_v) \{ \Phi \}
\]

\[
\text{BL-WP-FRAME}\quad P \ast \wp_E (e, \nu) \{ \Phi \} \vdash \wp_E (e, \nu) \{ (v, \nu_v). P \ast \Phi (v, \nu_v) \}
\]

\[
\text{BL-WP-VAL}\quad \vdash \wp_E (e, \nu) \{ \Phi \}\quad \text{BL-WP-LAM}\quad \vdash \wp_E (\lambda x.e, \nu) \{ \Phi \} \vdash \wp_E ((\lambda x.e)v, \nu) \{ \Phi \}
\]

\[
\text{BL-WP-FORK}\quad \vdash \Phi (f, \nu) \ast \wp_\rightarrow (e, \text{ForkView}(\nu)) \{ \_ \text{True} \} \vdash \wp_E (\text{fork} \{ e \}, \nu) \{ \Phi \}
\]

\[
\text{BL-WP-PLUS}\quad \vdash \wp_E (n \text{+} m, \nu) \{ \Phi \} \vdash \wp_E (n + m, \nu) \{ \Phi \}\quad \text{BL-WP-IF}\quad \vdash \wp_E ((b) ? e_1 : e_2, \nu) \{ \Phi \} \vdash \wp_E (\text{if} \ b \ \text{then} \ e_1 \ \text{else} \ e_2, \nu) \{ \Phi \}
\]

Figure 6.1: Pure primitive WPs in the RMC base logic

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

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 \[ (e, \nu) \xrightarrow{\epsilon \cdot (e_f, \nu_f)}' \xrightarrow{\xi} (e', \nu') \] (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 \((e, \nu)\)—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.

\[
\wp_E (e, \nu) \{ (v, \nu_v). Q \}
\]

\[
\{ P \} (e, \nu) \{ (v, \nu_v). Q \}
\]

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

Figure 6.1 presents a few pure WP rules that do not involve memory.
They are the expected WP rules from §5.6, but adapted with an arbitrary thread-view that mostly stays unchanged. Most notably, WP-\textsc{bind} propagates the thread-view after the expression has finished to the evaluation context, and BL-WP-\textsc{fork} picks the correct thread-view for the forked thread. Rules for binary operations are in the same form as BL-WP-\textsc{plus}, and the rule for \texttt{case} is similar to BL-WP-\textsc{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.

6.2 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 $\ell$, 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 \textsc{heap-read} and \textsc{heap-write} (§5.11).

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 $\ell$ being accessed, (ii) the thread-view $V$ of the executing thread, and (iii) the global race detector view for $\ell$ ($\mathcal{N}(\ell)$). Most importantly, the safety and the result of an access depends on the relations between the thread-view $V$ 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 §6.4, we will present the RAs needed to fictionally separate the machine state and define these local assertions within the logic.

\textbf{Definition 6.1 (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 $\mathbb{I} \mathbb{P} \mathbb{R} \mathbb{P} \mathbb{G} \mathbb{R}$.\footnote{see Property 3.12, §3.3}

- The seen thread-view observation $\text{Seen}(V)$ is a persistent knowledge that some thread’s thread-view $V$ is closed in the global memory.\footnote{Boyland, “Checking interference with fractional permissions” [Boy03].}
  This assertion is needed to guarantee that a memory access is grounded in the global memory.

- The history ownership assertion $\text{Hist}_q(\ell, h)$ is a fractional ownership\footnote{Boyland, “Checking interference with fractional permissions” [Boy03].} of the write messages $h \in \text{History}$ of the location $\ell$ in the global memory, where $\text{History} := \text{Time} \times \{ \text{val} : \text{Val}, \text{view} : \text{View} \}$. The fraction $q \in (0, 1]$ denotes shared or full ownership of $\ell$’s history. The allocated assertion $\text{Local}_k(\ell, h, V)$ says that the simple view $V$
A Base Logic for RMC in Iris

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 $\text{Read}^\text{na}_q(\ell, \alpha)$ is the fractional ownership of a subset $\alpha$ of ℓ’s non-atomic reads in the global race detector state. That is, $\alpha \subseteq \mathcal{N}(\ell)_{\text{na}}$ if $\mathcal{N}$ is the global race detector state A related persistent knowledge is the non-atomic read observation $\text{Local}^\text{na}_q(\ell, \alpha, V)$, which asserts that the simple view $V$ has observed a subset $\alpha$ of ℓ’s non-atomic reads. Both assertions are needed to perform race-free non-atomic reads on ℓ.

- The **atomic read** assertion $\text{Read}^\text{rlx}_q(\ell, \alpha)$ is the fractional ownership of a subset $\alpha$ of ℓ’s atomic reads in the global race detector state. That is, $\alpha \subseteq \mathcal{N}(\ell)_{\text{rlx}}$. The persistent knowledge $\text{Local}^\text{rlx}_q(\ell, \alpha, V)$ asserts that $V$ has observed a subset $\alpha$ of ℓ’s atomic reads. Both assertions are needed to perform race-free atomic reads on ℓ.

- The **atomic write** assertion $\text{Write}^\text{rlx}_q(\ell, \alpha)$ is the fractional ownership of a subset $\alpha$ of ℓ’s atomic writes in the global race detector state. That is, $\alpha \subseteq \mathcal{N}(\ell)_{\text{rlx}}$. The persistent knowledge $\text{Local}^\text{rlx}_q(\ell, \alpha, V)$ asserts that $V$ has observed a subset $\alpha$ 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 $\uparrow^\text{b}_q \ell$ 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 6.2.

**Property 6.2** (Seen Thread-view Observations). The seen thread-view observation is timeless and persistent ($\text{BL-SEEN-TIMELESS}$ and $\text{BL-SEEN-PERS}$). As $\text{Seen}(V)$ is only a snapshot of some thread’s thread-view at some point, it can be joined with others ($\text{BL-SEEN-JOIN}$), or can be forked to get old snapshots ($\text{BL-SEEN-DOWNCLOSED}$). With $\text{Seen}(V)$, we know that the thread-view $V$ 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 §6.6.

**Property 6.3** (History Ownership). The assertion $\text{Hist}_q(\ell, h)$ is timeless ($\text{BL-HIST-TIMELESS}$) and fractional ($\text{BL-HIST-FRAC-VALID}$ and $\text{BL-HIST-FRAC}$). Owning a fraction of the assertion is sufficient to know the history of ℓ, as implied by $\text{BL-HIST-AGREE}$. A change to the history requires the full fraction, written as $\text{Hist}(\ell, h)$ without the fraction $q = 1$, which is exclusive, as in $\text{BL-HIST-EXCL}$. ($\text{BL-HIST-EXCL}$ is derivable from $\text{BL-HIST-EXCL-VALID}$ and $\text{BL-HIST-EXCL-EXCL}$.)

$\text{BL-HIST-DROP-SINGLETON}$ allows us to truncate the current history to just a singleton of the latest write. This is a convenient abstraction for ℓ’s
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 \( \text{Hist}_q(\ell, h) \) 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 6.4 (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 \( \text{Write}^{\ell}_{q}(\ell, \alpha) \) maintain agreement on \( \ell \)'s set of atomic writes in the race detector (\( \text{BL-ATW-frac} \)), and the full fraction is required to update the set. A fraction \( q \) of the read assertions \( \text{Read}^{na}_{q}(\ell, \alpha) \) or \( \text{Read}^{rlx}_{q}(\ell, \alpha) \) on the other hand does not maintain agreement. Instead, a fraction only maintains that the set \( \alpha \) 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 (\( \text{mo}^{\ell} \)) 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 \( \text{BL-NAR-join} \) or \( \text{BL-ATR-join} \).

**Property 6.5** (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.

\[
\text{Local}^{l}_{h}(\ell, h, V) := \exists t \in \text{dom}(h). t \subseteq V(\ell).w
\]

\[
\text{Local}^{rlx}_{q}(\ell, \alpha, V) := \alpha \subseteq V(\ell).aw
\]

\[
\text{Local}^{na}_{q}(\ell, \alpha, V) := \alpha \subseteq V(\ell).nr
\]

\[
\text{Local}^{rlx}_{q}(\ell, \alpha, V) := \alpha \subseteq V(\ell).ar
\]

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 \( \text{BL-ALLOC-MONO} \)). View monotonicity is an important property that we will rely on heavily (see Chapter 7).

The local observations for reads can be joined together using \( \text{BL-NAR-JOIN} \) and \( \text{BL-ATR-JOIN} \).

**Property 6.6** (Block Ownership). The block ownership assertion is also timeless and fractional. \( \text{BL-BLOCK-JOIN} \) 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.

### 6.3 Primitive Memory Rules

We now see how the local assertions are meant to be used in our primitive memory rules, given in Figures 6.3 to 6.6. 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 (\( \text{BL-HOARE-WRITE-NA} \)), but with the block ownership assertions. For the sake of simplicity, we will present them in a cleaner form in Chapter 8 (see \( \text{NA-ALLOC} \) and \( \text{NA-DEALLOC} \)).

#### 6.3.1 Rules for Fences

Figure 6.3 presents the simplest memory-related primitive rules, for release and acquire fences. Both \( \text{BL-HOARE-REL-FENCE} \) and \( \text{BL-HOARE-ACQ-FENCE} \) requires \( \text{Seen}(\mathcal{V}) \) as the pre-condition for a fence running with the
thread-view \( V \). Their post-conditions say that the fence instructions will return poison \( \bot \) with a new thread-view \( V' \supseteq V \). The effects of the fences are approximated by the properties of \( V' \). In case of an acquire fence, the current component of \( V' \) is updated to include its acquire component, exactly reflecting \( \text{OM-ACQ-FENCE} \) (\S3.3). In case of a release fence, the release-fence component of \( V' \) is updated to include its current component. This only approximates \( \text{OM-REL-FENCE} \) (also \S3.3) because it hides away (in the relation \( V' \supseteq V \)) the changes to the per-location release views \( V'. \text{rel} \). We made this abstraction to keep the rule simple, as we have never needed such detailed information on \( V'. \text{rel} \).

### 6.3.2 Rules for Reads and Writes

The rules for reads and writes are given in Figure 6.4 and Figure 6.5.

**NON-ATOMIC READS.** To guarantee data-race freedom, \( \text{BL-\text{HOARE-READ-NA}} \) requires a pre-condition that implies \( \text{DRF-READ-NA} \) (\S3.4). That is, the pre-condition ensures that the executing thread has observed all writes (non-atomic and atomic) to \( \ell \). The pre-condition thus includes:

- \( \text{Seen}(V) \), 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 \( \text{Hist}_q(\ell, [t \leftarrow (v, V')] ) \) where \( (t, v, V') \) is \( \ell \)'s latest write; and
- the knowledge \( \text{Local}_q(\ell, [t \leftarrow (v, V')], V.\text{cur}) \) that the current thread-view \( V.\text{cur} \) has observed not only \( \ell \)'s allocation but also its latest write; and
- a fraction \( q \) of the race detector’s atomic writes set \( \text{Write}^q_{\text{r}lx}(\ell, \alpha_w) \) and the knowledge \( \text{Local}^q_{\text{r}lx}(\ell, \alpha_w, V.\text{cur}) \) that the current thread-view has observed all atomic writes \( \alpha_w \); and
- a fraction \( q \) of the race detector’s non-atomic reads set \( \text{Read}^q_{\text{na}}(\ell, \alpha_r) \), needed to extend the read set \( \alpha_r \) with the read to be performed.

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 \( \ell, r \) representing this read, and the thread arrives at a new thread-view \( V' \) represented by \( \text{Seen}(V') \). We can make an abstraction here on how \( V' \) is related to \( V \) (like in the rule \( \text{BL-\text{HOARE-REL-FENCE}} \)), but the higher-level rules require a detailed relation between \( V' \) and \( V \), so we simply keep the “raw” relation \( V \xrightarrow{\text{\alpha_n, \ell, r, T, \bot}} V' \) from the operational semantics. \(^7\) Note that we do know that \( V \subseteq V' \).

\(^{5}\)see OE-\text{FENCE}, \S4.2

\(^{6}\)Recall that observing the latest write (the previous part of the pre-condition) 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.

\(^{7}\)see OM-\text{POST-READ-TVIEW}, \S3.3
### BL-HOARE-READ-NA
\[
\text{Local}_\ell([t \leftarrow (v, V')], V, \text{cur}) \quad \text{Local}_\ell^{\text{rlx}}(\ell, \alpha_w, V, \text{cur}) \quad \text{Local}_\ell^{\text{na}}(\ell, \alpha_r, V_r)
\]
\[
\{ \text{Seen}(V) \ast \text{Hist}_\ell([t \leftarrow (v, V')]) \ast \text{Write}_\ell^{\text{rlx}}(\ell, \alpha_w) \ast \text{Read}_\ell^{\text{na}}(\ell, \alpha_r) \}
\]
\[
\{ w, V' \}, w = v \ast \exists r, V \xrightarrow{\text{r, } \ell, t, \perp} V' \ast \text{Local}_\ell(\ell, \alpha_r \cup \{ r \}, V_r \cup V', \text{cur}) \ast \\
\text{Seen}(V) \ast \text{Hist}_\ell([t \leftarrow (v, V')]) \ast \text{Write}_\ell^{\text{rlx}}(\ell, \alpha_w) \ast \text{Read}_\ell^{\text{na}}(\ell, \alpha_r \cup \{ r \}) \}
\]

### BL-HOARE-WRITE-NA
\[
\text{Local}_\ell([t \leftarrow (v, V')], V, \text{cur}) \quad \text{Local}_\ell^{\text{na}}(\ell, \alpha_1, V, \text{cur}) \quad \text{Local}_\ell^{\text{rlx}}(\ell, \alpha_2, V, \text{cur}) \quad \text{Local}_\ell^{\text{rlx}}(\ell, \alpha_w, V, \text{cur})
\]
\[
\{ \text{Seen}(V) \ast \text{Hist}_\ell([t \leftarrow (v, V')]) \ast \text{Write}_\ell^{\text{rlx}}(\ell, \alpha_w) \ast \text{Read}_\ell^{\text{na}}(\ell, \alpha_1) \ast \text{Read}_\ell^{\text{rlx}}(\ell, \alpha_2) \}
\]
\[
(\ell := \text{na} w', V')
\]
\[
\{ w', V' \}, \exists t', V \xrightarrow{\text{x, } \ell, t', \perp} V' \ast \text{Local}_\ell(\ell, [t' \leftarrow (v', \perp)], V', \text{cur}) \ast \\
\text{Seen}(V) \ast \text{Hist}_\ell([t' \leftarrow (v', \perp)]) \ast \text{Write}_\ell^{\text{rlx}}(\ell, \alpha_w) \ast \text{Read}_\ell^{\text{na}}(\ell, \alpha_1) \ast \text{Read}_\ell^{\text{rlx}}(\ell, \alpha_2) \}
\]

---

**Figure 6.4:** The base logic’s primitive Hoare rules for non-atomic reads and writes

---

Last but not least, we have an additional assertion `Local(\ell, \alpha_r, V_r)` in the pre-condition that is updated to `Local(\ell, \alpha_r \cup \{ r \}, V_r \cup V', \text{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 \cup V' \text{cur}` after the step) that has observed the subset `\alpha_r` of all non-atomic reads performed so far with the fraction `Read(\ell, \alpha_r)`. The view `V_r` will only be needed later when a thread has collected the full fraction `Read(\ell, \alpha_r)` where `\alpha_r` is then the complete set of `\ell`’s non-atomic reads. At that point, race-free operations would require that the executing thread has observed all reads in `\alpha_r`, which pins down to the thread’s thread-view `V` including `V_r`, i.e., `V_r \subseteq V \text{cur}`. We will see concretely how this view is used in Chapter 8.

#### 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 `\ell`.\(^8\) The pre-condition therefore requires full ownership (the fraction `q = 1`) of `\ell`’s current singleton history, and of the 3 sets of the race detector’s state, and the knowledge that the current thread-view `V \text{cur}` has observed `\ell`’s allocation, latest write, and those sets of all reads and all atomic writes to `\ell`.\(^9\)

The post-condition keeps most ownership unchanged, and only updates the singleton history ownership to `Hist([t' \leftarrow (v', \perp)])`, 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 `V'` computed from `V' (V \xrightarrow{\text{x, } \ell, t', \perp} V')`,\(^10\) and knows that the new current view `V' \text{cur}` has observed the new write (`Local(\ell, [t' \leftarrow (v', \perp)], V' \text{cur})`).

#### Atomic reads

The rule BL-HOARE-READ-AT for atomic reads is rather simple: it requires as pre-condition fractional ownership `q` of `\ell`’s current history, and of an atomic read subset of the race detector’s state `Read(\ell, \alpha)` to add the read to be performed. An atomic read only needs to avoid race with non-atomic writes,\(^11\) so `Local(\ell, h, V \text{cur})` is

---

\(^8\)see DRF-WRITE-NA, §3.4

\(^9\)Recall that a fraction of `Read(\ell, \alpha_1)` or `Read(\ell, \alpha_2)` only says that `\alpha_1` or `\alpha_2` is only a subset of the global set—we need a full fraction to be guaranteed that `\alpha_1` or `\alpha_2` is the global set.

\(^10\)see OM-POST-WRITE-TVIEW, §3.3

\(^11\)see DRF-READ-AT, §3.4
BL-HOARE-read-at
\[
\text{r1x} \subseteq o \quad \text{Local}_q(\ell, h, \text{V.cur}) \quad \text{Local}_q^{\text{r1x}}(\ell, \alpha, V_r)
\]
\[
\{ \text{Seen}(\text{V}) \ast \text{Hist}_q(\ell, h) \ast \text{Read}_q^{\text{r1x}}(\ell, \alpha) \}
\]
\[
(\alpha \ast \ell, \text{V})
\]
\[
\{ (v, V'). \exists \ell, V', r. h(t) = (v, V') \ast \text{V} \overset{\text{R.o.e.t.V'}}{\text{\rightarrow}} \ast \text{V}' \ast \text{Local}_q^{\text{r1x}}(\ell, \alpha \cup \{r\}, V_r \cup V'.\text{cur}) \ast \\
\quad \text{Seen}(\text{V}') \ast \text{Hist}_q(\ell, h) \ast \text{Read}_q^{\text{r1x}}(\ell, \alpha \cup \{r\}) \}
\]
\[
\text{\varepsilon}
\]

BL-HOARE-write-at
\[
\text{r1x} \subseteq o \quad \text{Local}_q(\ell, h, \text{V.cur}) \quad \text{Local}_q^{\text{na}}(\ell, \alpha_r, \text{V.cur}) \quad \text{Local}_q^{\text{r1x}}(\ell, \alpha_w, V_w)
\]
\[
\{ \text{Seen}(\text{V}) \ast \text{Hist}(\ell, h) \ast \text{Write}_q^{\text{r1x}}(\ell, \alpha_w) \ast \text{Read}_q^{\text{na}}(\ell, \alpha_r) \}
\]
\[
(\ell := v', \text{V})
\]
\[
\{ (v', V'). \exists \ell' \notin \text{dom}(h), V', \text{V} \overset{\text{R.o.e.t.V'}}{\text{\rightarrow}} \ast \text{V}' \ast \text{Local}_q^{\text{r1x}}(\ell, \alpha_w \cup \{t'\}, V_w \cup V'.\text{cur}) \ast \\
\quad \text{Seen}(\text{V}') \ast \text{Hist}(\ell, h[t'[\leftarrow (v', V')]) \ast \text{Write}_q^{\text{r1x}}(\ell, \alpha_w \cup \{t'\}) \ast \text{Read}_q^{\text{na}}(\ell, \alpha_r) \}
\]
\[
\text{\varepsilon}
\]

ATOMIC WRITES. An atomic write must not race with non-atomic accesses, both reads and writes, so \text{BL-HOARE-write-at} requires as pre-condition full fractions of the current history ownership \text{Hist}(\ell, h) and of the non-atomic reads set \text{Read}_q^{\text{na}}(\ell, \alpha_r), as well as the knowledge \text{Local}_q(\ell, h, \text{V.cur}) and \text{Local}_q^{\text{na}}(\ell, \alpha_r, \text{V.cur}) that the current thread-view \text{V.cur} has observed all non-atomic writes and reads to \ell. Additionally, the full ownership \text{Write}_q^{\text{r1x}}(\ell, \alpha_w) of the race detector’s atomic write set for \ell is needed to extend the set \alpha_w with the write to be performed.

The post-condition keeps the non-atomic reads set unchanged, extends the atomic writes set with the new timestamp \(t'\) (fresh in \text{h}), and updates the history and the atomic writes set to insert the new write \((h[t'\leftarrow (v', V')])\) in the \text{\alpha_w}\cup\{t'\}). The new thread-view \text{V}'. is computed from \text{V} accordingly.

The view \text{V}_w in \text{Local}_q^{\text{r1x}}(\ell, \alpha_w, V_w) plays the same role as the view \text{V} in \text{BL-HOARE-read-na} and \text{BL-HOARE-read-at}. It is the view that has observed all atomic writes \alpha_w done so far using \text{Write}_q^{\text{r1x}}(\ell, \alpha_w). We will

\[12\text{see OM-post-read-view, §3.3}\]

\[13\text{see DRF-write-at, §3.4}\]

\[14\text{see OM-post-write-view, §3.3}\]
BL-HOARE-CAS
\[
\begin{align*}
\text{rlx} & \sqsubseteq o_f, o_r, o_w & \text{Local}_h(\ell, h, V.\text{cur}) & \text{Local}_v^m(\ell, \alpha_1, V.\text{cur}) \\
\text{Local}_{\text{q}}^{\text{rlx}}(\ell, \alpha_2, V_r) & \text{Local}_{\text{v}}^{\text{rlx}}(\ell, \alpha_w, V_w) & \forall v_0 \in \text{Readable}(h, V), \top v_0 \neq v_r
\end{align*}
\]

\[\begin{aligned}
\text{P}_{\text{cmp}} \star \Box ((v_r = \ell_r) \land (P_{\text{cmp}} \Rightarrow \Phi_{\text{cmp}}(\ell_r) : \text{True}))
\end{aligned}\]

\[\begin{aligned}
\text{CAS}^{o_f, o_r, o_w} (\ell, v_1, v_2, V)
\end{aligned}\]

\[\begin{aligned}
\text{b, } V^\prime. & \text{ P}_{\text{cmp}} \star \exists h', t, t', v', V', r, V_r. V \sqsubseteq V^\prime \sqsubseteq V + h(t') = (v', V'') \star \\
& \text{Local}_{\text{q}}^{\text{rlx}}(\ell, b) \cup \{t\} : \alpha_w, V_w \sqcup V'.\text{cur} \star \text{Local}_{\text{q}}^{\text{rlx}}(\ell, \alpha_2 \cup \{r\}, V_r \sqcup V'.\text{cur}) \star \\
& \text{Seen}(V^\prime) \star \text{Hist}(\ell, h') \star \text{Write}^{\text{rlx}}(\ell, b) \cup \{t\} : \alpha_w \star \text{Read}^m(\ell, \alpha_1) \star \text{Read}^m_q(\ell, \alpha_2 \cup \{r\}) \star \\
& \text{b} = \text{false} \land v_r = v' \neq h' = h \star V \xrightarrow{R_{o_f, \ell, t, V', r}} V_x \star \\
& \forall \text{b} = \text{true} \land v_r = v' \land t \notin \text{dom}(h) \land t = t' = 1 \star V \sqsubseteq V' \star h' = h[t \leftarrow (v_w, V)] \star \\
& V \xrightarrow{R_{o_r, \ell, t, V', r}} V_x \sqcup \alpha_w, V, V_r \xrightarrow{\forall} V' \star \\
& \varepsilon
\end{aligned}\]

where \(\Phi_{\text{cmp}}(\ell_r) := (\exists_E \exists q_r, h_r. \triangleright \text{Hist}_{q_r}(\ell_r, h_r)) \land (\forall \ell' \in \text{Readable}(h, V) \setminus \{\ell_r\}. \exists_E \exists q', h'. \triangleright \text{Hist}_{q'}(\ell', h'))\)

Figure 6.6: The base logic’s primitive Hoare rule for CASes

also see concretely how \(V_w\) is used in Chapter 9.

6.3.3 A Rule for CASes

We present a Hoare rule BL-HOARE-CAS for CASes in Figure 6.6. 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).\(^\text{15}\) so the pre-condition requires the full fractions of the history ownership \(\text{Hist}(\ell, h)\), and of the non-atomic reads set \(\text{Read}^m(\ell, \alpha_1)\), and the knowledge \(\text{Local}_h(\ell, h, V.\text{cur})\) and \(\text{Local}_v^m(\ell, \alpha_1, V.\text{cur})\) that \(V.\text{cur}\) has observed \(\ell\)'s allocation and all of its non-atomic reads and writes.

Second, the pre-condition needs the full fraction of the atomic writes set \(\text{Write}^{\text{rlx}}(\ell, \alpha_w)\) and a fraction of the atomic reads set \(\text{Read}^m_q(\ell, \alpha_2)\) 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 \(\alpha_w\) and \(\alpha_2\) respectively, and play the similar role to those in the read and write rules, which we will see in Chapter 9.

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 \((\ell', v', V')\) in \(b\) will be read, and the atomic reads set \(\alpha_2\) will be extended with a new action \(R_{o_f}\) for that read. The non-atomic reads set remains unchanged.

\[\begin{aligned}
\text{In case the CAS fails, i.e., } b = \text{false}, \text{ we know that the value read } v' \text{ is not equal to the expected value } v_r \land v' \neq v_r.\text{ Also that the history } h \text{ and the atomic writes set } \alpha_w \text{ also remain unchanged, and that the thread-view effect is similar to that of a read with the mode } o_f \\
\end{aligned}\]

\[\begin{aligned}
(\forall \ell'. v', V'. r \rightarrow V.	ext{cur})\]
6.3.4 A Stronger WP Rule for CASes

We present the rule \( \text{BL-WP-CAS} \) for shaped WP. For example, if \( \Phi \) triples and WPs are usually written with a universally quantified post-triples are derived from WPs. In practice (in Coq), Iris rules for Hoare Notation 6.7 (Iris WP-style Rules) in Coq, but also important to make our CAS rule stronger. \( \Phi \) quantified as ion that is common with Iris WPs, where the post-condition is universally locale. Intuitively, the persistent implication \( \Box \) be used is encoded in the predicate \( \Phi \). 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. 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 \( v_h \) that the CAS may read: \( \forall v_h \in \text{Readable}(h, V). \top \vdash v_h \equiv v_r. \) The set of readable values \( \text{Readable}(h, V) \) is lifted for histories from Definition 4.6 for the global memory. The comparison is safe if the values are comparable (Definition 4.9).

\[ \text{Determinate Pointer Comparison.} \] If the comparison is between locations, \( i.e. \), if \( v_r \) is a non-null location \( \ell_r \), the pre-condition of \( \text{BL-Hoare-CAS} \) requires some extra resources \( P_{\text{cmp}} \) to learn that compared locations are alive, and thus to guarantee deterministic comparison.20 Furthermore, \( P_{\text{cmp}} \) will only be used to derive facts and will not be consumed, so it is returned as-is in the post-condition. How \( P_{\text{cmp}} \) will be used is encoded in the predicate \( \Phi_{\text{cmp}}(\ell_r) \) which employs a classical conjunction. Intuitively, the persistent implication \( \Box(P_{\text{cmp}} \Rightarrow \Phi_{\text{cmp}}(\ell_r)) \) requires that the resources in \( P_{\text{cmp}} \) simultaneously support two goals:

1. using \( P_{\text{cmp}} \) and potentially opening some invariant with the fancy update \( \equiv_{\ell_r} \), one gets some fraction of the ownership \( \text{Hist}_{\ell_r}(\ell_r, h_i) \) of the expected value \( v_r \), which is sufficient to learn that \( \ell_r \) is alive.

2. for any location \( \ell' \) readable from \( h \) that is not \( \ell_r \), using \( P_{\text{cmp}} \) and potentially opening some invariant, one also learns that \( \ell' \) is alive.

\[ \top \vdash (P \Rightarrow \forall v. \{Q \Rightarrow P_{\text{cmp}} \Rightarrow \Phi_{\text{cmp}}(\ell_r)\}) \Rightarrow \text{wp e} \{\Phi\} \]

- In case the CAS succeeds, \( i.e. \), \( b = \text{false} \), 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 \( V' \) and the write message view \( V \) are computed from the old thread-view \( V \) and the read message view \( V'' \) accordingly.18 The atomic writes set \( \alpha_w \) is also extended with the new timestamp \( t \).

\[ \top \vdash (P \Rightarrow \forall v. \{Q \Rightarrow P_{\text{cmp}} \Rightarrow \Phi_{\text{cmp}}(\ell_r)\}) \Rightarrow \text{wp e} \{\Phi\} \]

\[ \top \vdash (P \Rightarrow \forall v. \{Q \Rightarrow P_{\text{cmp}} \Rightarrow \Phi_{\text{cmp}}(\ell_r)\}) \Rightarrow \text{wp e} \{\Phi\} \]

\[ \top \vdash (P \Rightarrow \forall v. \{Q \Rightarrow P_{\text{cmp}} \Rightarrow \Phi_{\text{cmp}}(\ell_r)\}) \Rightarrow \text{wp e} \{\Phi\} \]
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 form of rule is more applicable to a goal of form $\text{wp } K[e] \{\Psi\}$: we first apply the bind rule $\text{WP-BIND}$ to focus on the expression $e$ and push the continuation into the post-condition (i.e., $\text{wp } e \{v. \text{wp } K[v] \{\Psi\}\}$) which will then be used to instantiate the predicate $\Phi$ of the rule.

We state our CAS rule in the following form:

$$
\frac{R \quad R'}{\vdash \text{wp } (e, V) \{\Phi\}}
$$

where $P$ is the pre-condition, $Q$ is the post-condition, and $R$ and $R'$ are extra premises. The wand implication $\triangleright \forall v, V_v. Q \rightarrow \Phi(v, V_v)$ is for the post-condition and is the right-most premise. A rule of this form can be read as the following Hoare rule:

$$
\frac{R \quad R'}{\vdash \{P\} (e, V) \{(v, V_v), Q\}}
$$

Our CAS rule $\text{BL-WP-CAS}$, however, is strengthened by moving the later inside and adding fancy updates to the post-condition—a combination called wand step viewshifts.

**Notation 6.8 (Wand Step Viewshifts).**

The wand step viewshift is a balanced (potentially mask-changing) view-shift that has a later in between.

$$
P \equiv E' Q ::= P \mapsto E' \triangleright E' \triangleright E Q
$$

We can now look at $\text{BL-WP-CAS}$ in Figure 6.7. The rule can be applied with an arbitrary post-condition $\Psi$ which typically is the continuation after executing the CAS. The rule says that the client can go on proving $\Psi$ assuming the return value $(b, V')$ (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 6.7) of the Hoare rule $\text{BL-HOARE-CAS}$, $\text{BL-HOARE-CAS}$ are strengthened in several ways.

- The client of the rule does not need to specify and provide $P_{\text{cmp}}$ in the pre-condition. Instead, the client can pick $P_{\text{cmp}}$ 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) $P_{\text{cmp}}$ and how it is to be used ($\Phi$) if the CAS succeeds ($b = \text{true}$).

- Note that the client however does not know that “$v_r = v'$ in case $b = \text{true}$” before picking and proving $P_{\text{cmp}}$, because $P_{\text{cmp}}$ is needed to achieve that deterministic comparison result.
\[ \begin{align*}
& \text{BL-WP-CAS} \\
& \forall b, \forall t' = t, \forall v_0 \in \text{Readable} (b, V), \\
& \quad \vdash v_0 \equiv v_r \\
& \quad \text{where } \Phi_{\text{cmp}} (P_{\text{cmp}}, \ell_r, v') := \land \ P_{\text{cmp}} \mapsto E, \equiv_{E_r} \equiv_{q_r, h_r} \triangleright \text{Hist}_{q_r} (\ell_r, h_r) \\
& \quad \land v'' = v' \not= \ell_r, \ P_{\text{cmp}} \mapsto E \mapsto E' \mapsto E'' \triangleright E_0 (\ell') \equiv q', h'. \triangleright \text{Hist}_{q'} (\ell', h')
\end{align*} \]

- After proving \( P_{\text{cmp}} \) and \( \Phi \), the client does get the deterministic comparison result and \( P_{\text{cmp}} \) back. Recall that \( P_{\text{cmp}} \) 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 \( \triangleright E' \) allows the client to make a mask-changing viewshift from the mask \( E \) to the mask \( E' \) to open invariants (\( E' \) 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 \( E \), all in order to prove the continuation \( \Psi (b, V') \). Note that if the client uses \( \text{BL-HOARE-CAS} \), 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 \( \Phi_{\text{cmp}} \), i.e., the proof that \( P_{\text{cmp}} \) implies that the compared locations are alive. In particular, the mask \( E_r \) and the function \( E_0 \) from locations to masks are also of the client’s choice. The client only needs to show that expected value \( \ell_r \) and the read value \( v' \) are alive. Interestingly, the client can do so by opening invariants (of the client’s choice) without closing them.
6.4 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 6.9 (Lattice RA for Seen Thread-view Observations).** The lattice RA $\text{Lat}(A)$ 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, $\text{Lat}(A) ::= (\{A, \lambda_\_\_\_\_\_\_\_, \text{id}, \sqcup\})$. If the lattice $A$ has a bottom element $\bot \subseteq a$ for any $a \in A$, then $\bot$ is the unit of $\text{Lat}(A)$. Note that RA inclusion $\subseteq$ then coincides with the lattice order $\sqsubseteq$. Most importantly, the RA has the following properties.

\[
\forall a, b. \text{valid}(\bullet a \cdot b) \Rightarrow b \sqsubseteq a \quad \text{(AUTH-LAT-VALID)}
\]

\[
\forall a, b. b \sqsubseteq a \Rightarrow \bullet b \rightsquigarrow \bullet a \cdot a \quad \text{(AUTH-LAT-UPDATE)}
\]

For $\text{Seen}(V)$, we use the RA $\text{SEENR} = \text{AUTH(\text{Lat(View)})}$. It is an optimization that we only track a simple view with $\text{Lat(View)}$ and not a thread-view with $\text{Lat(ThreadView)}$. Recall that the role of $\text{Seen}(V)$ is to guarantee that $V$ is closed in the global memory, which can be done instead by just guaranteeing that $V.\text{acq}$ is closed in the global memory, because $V.\text{acq}$ includes all other components of a wellformed $V$. 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 $\circ V'.\text{acq}$, thanks to AUTH-LAT-VALID, $V'.\text{acq} \subseteq V_{up}$. 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 6.10 (Fractional Agreement Map RA for History).** We use the RA $\text{HISTR} = \text{AUTH(\text{MAP}($\text{Loc}$, $\text{FRAC} \times \text{AG($\text{History'}$)}$))}$ for $\text{Hist}_{\ell}(\ell, h)$.

The agree RA $\text{AG}$ only provides valid composition between elements that are the same, and the fractional RA $\text{FRAC}$ 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 $\text{MAP}$ 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 $\text{MAP}$ with $\text{FRAC}$ and $\text{AG}$ gives per-location agreement between fractions of history ownership, and with the full fraction we can change the history. We use the option type $\text{History'}$ to support deallocation: when a location is deallocated, then its history will be None. We use $\text{AUTH}$ to have the authoritative element be the complete memory, and the fragmentary elements of singleton maps will be used to define $\text{Hist}_{\ell}(\ell, h)$. More concretely, we have the following properties.

\[
\forall m, \ell, q, h. \text{valid}(\bullet m \cdot o [\ell \leftarrow (q, \text{ag}(h))]) \Rightarrow m(\ell) = (1, \text{ag}(h))
\]

\[
\forall \ell, q, h, q', h'. \text{valid}(o [\ell \leftarrow (q, \text{ag}(h))] \cdot o [\ell \leftarrow (q', \text{ag}(h'))]) \Rightarrow h = h'
\]

\[
\forall m, \ell, h. \bullet m \cdot o [\ell \leftarrow (1, \text{ag}(h))] \rightsquigarrow \bullet m[\ell \leftarrow (1, \text{ag}(h))] \cdot o [\ell \leftarrow (1, \text{ag}(h'))]
\]
The injection $ag$ is the constructor of the RA $Ag$.

**Definition 6.11** (Fractional Map RA for Atomic Writes Sets). We use the RA $\text{WRITER} = \text{AUTH}(\text{MAP}(\text{Loc, FRAC} \times Ag(\text{ActIds})))$ 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 6.12** (Fractional Set Lattice RA for Reads Sets). We use a slightly different RA $\text{READR} = \text{AUTH}(\text{MAP}(\text{Loc, FRAC} \times \text{LAT}(\text{ActIds})))$. That is, we use $\text{LAT}(\text{ActIds})$ in place of $Ag(\text{ActIds})$. As such, we do not have agreement between the fractions of read sets. Instead, a fraction is sufficient to grow the set: we can update the element $\ell \leftarrow (q, \alpha)$ (together with the authoritative element) to $\ell \leftarrow (q, \alpha')$ where $\alpha \subseteq \alpha'$ without requiring $q = 1$. Note that because we use the lattice RA $\text{LAT}$, the sets can only grow. More concretely, we have the following properties.

$$\forall m, \ell, q, \alpha. \text{valid}(\bullet m \cdot \ell \leftarrow (q, \alpha)) \Rightarrow \exists \alpha'. m(\ell) = (1, \alpha') \land \alpha \subseteq \alpha'$$

$$\forall m, \ell, q, \alpha, \alpha', \alpha''. m(\ell) = (1, \alpha') \Rightarrow \alpha \subseteq \alpha'' \Rightarrow$$

$$\bullet m \cdot \ell \leftarrow (1, \alpha) \Rightarrow \bullet m[\ell \leftarrow (q, \alpha', \alpha'') \cdot \ell \leftarrow (q, \alpha'')]$$

**Definition 6.13** (Fractional Block RA for Block Ownership). We use the RA $\text{BLOCKR} = \text{AUTH}(\text{MAP}(\mathbb{N}^+, \text{FRAC} \times \text{MAP}(\mathbb{Z}, \text{EX}(1))))$. 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 us 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 $\text{EX}$, such per-location block ownership is unique.

6.5 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 $\gamma_{\text{SEEEN}}, \gamma_{\text{HIST}}, \gamma_{\text{NAR}}, \gamma_{\text{ATR}}, \gamma_{\text{ATR}},$ and $\gamma_{\text{BLK}}$. These ghost locations will need to be allocated before any program runs (in the adequacy proof, see Theorem 6.19).

**Definition 6.14** (Ghost State Model of Local Assertions). We define our local assertions purely as ghost ownership of fragmentary elements.

- $\text{Seen}(\mathcal{V}) := \text{acq} \bowtie \text{SEEN}^{\gamma_{\text{SEEEN}}}$
- $\text{Hist}_{q}(\ell, h) := \ell \leftarrow (q, \gamma_{\text{AG}}(\text{Some}(h))) \cdot \text{HIST}^{\gamma_{\text{HIST}}}$
- $\text{Write}_{q}^{\text{RLX}}(\ell, \alpha) := \ell \leftarrow (q, \alpha) \cdot \text{WRITER}^{\gamma_{\text{WRIT}}}$
- $\text{Read}_{q}^{\text{R}}(\ell, \alpha) := \ell \leftarrow (q, \alpha) \cdot \text{READR}^{\gamma_{\text{READ}}}$
- $\text{Read}_{q}^{\text{RLX}}(\ell, \alpha) := \ell \leftarrow (q, \alpha) \cdot \text{READR}^{\gamma_{\text{READ}}}$
- $\downarrow_{q}^{n} := \ell \leftarrow n \cdot \text{BLOCKR}^{\gamma_{\text{BLK}}}$

The injection $ex$ is the constructor of the RA $\text{EX}$. 
Definition 6.15 (Ghost Ownership for the Global State). The ghost ownership \( \text{GlobalGhost} \) 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 \text{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 \(V_{cut}\).

\[
\text{GlobalGhost}(\mathcal{M}, \mathcal{N}, V_{up}, V_{cut}) := \\
\begin{align*}
\bullet \quad V_{up} & : \text{SEEN}^* \\
\bullet \quad \ell & \leftarrow (1, \text{ag}(\text{trunc}(\mathcal{M}(\ell), V_{cut}(\ell)), w)) | \ell \in \text{dom}(\mathcal{M}) : \text{HIST}^* \\
\bullet \quad \ell & \leftarrow (1, \text{ag}(\mathcal{N}(\ell).aw)) | \ell \in \text{dom}(\mathcal{N}) : \text{WRITE}^* \\
\bullet \quad \ell & \leftarrow (1, \mathcal{N}(\ell).nr) | \ell \in \text{dom}(\mathcal{N}) : \text{READR}^* \\
\bullet \quad \ell & \leftarrow (1, \mathcal{N}(\ell).ar) | \ell \in \text{dom}(\mathcal{N}) : \text{READA}^* \\
\bullet \quad i & \leftarrow (1, [n \leftarrow \text{ex}() | (i,n) \in \text{dom}(\mathcal{M})]) | i,0 \in \text{dom}(\mathcal{M}) : \text{BLOCK}^*
\end{align*}
\]

where

\[
\text{trunc}(h, t_0) := \begin{cases} 
\text{None} & \text{if } h \text{ is deallocated} \\
\text{Some}([t \leftarrow h(t) | t_0 \leq t \in \text{dom}(h)]) & \text{if } h \text{ is alive}
\end{cases}
\]

So \(\text{GlobalGhost}(\mathcal{M}, \mathcal{N}, V_{up}, V_{cut})\) 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 \(V_{cut}\); 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 \(\mathcal{M}\).

We use the map insert notation, e.g., \([\ell \leftarrow (1, \mathcal{N}(\ell).nr) | \ell \in \text{dom}(\mathcal{N})]\) 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 6.16 (Agreements between the Global Ghost State and Local Assertions). The global ghost state ownership \(\text{GlobalGhost}\) and the local assertions satisfy several agreement properties given in Figure 6.8. They are all derived from validity of the corresponding RAs.

Lemma 6.17 (Updates of the Global Ghost State and Local Assertions). \(\text{GlobalGhost}\) can be updated together the local assertions following the rules in Figure 6.9. They are all derived from frame-preserving updates of the corresponding RAs, and the properties of \(\text{trunc}\).

Most notably, \text{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 \(\ell\) up to its latest timestamp \(t\), which is the input to \(\text{trunc}\). The rule is used to prove
BL-GHOST-SEEEN
\[ \text{GlobalGhost}(\mathcal{M}, \mathcal{N}, V_{up}, V_{cut}) \ast \text{Seen}(V) \vdash V_{acq} \subseteq V_{up} \land V \in \mathcal{M} \]

BL-GHOST-HIST
\[ \text{GlobalGhost}(\mathcal{M}, \mathcal{N}, V_{up}, V_{cut}) \ast \text{Hist}_q(\ell, h) \vdash \text{trunc}(\mathcal{M}(\ell), V_{cut}(\ell)) = h \land \ell \notin \text{unalloc}(\mathcal{M}) \]

BL-GHOST-AT-READ
\[ \text{GlobalGhost}(\mathcal{M}, \mathcal{N}, V_{up}, V_{cut}) \ast \text{Write}_q(\ell, \alpha) \vdash N(\ell).aw = \alpha \]

BL-GHOST-NA-READ
\[ \text{GlobalGhost}(\mathcal{M}, \mathcal{N}, V_{up}, V_{cut}) \ast \text{Read}_q(\ell, \alpha) \vdash \alpha \subseteq N(\ell).nr \land (q = 1 \Rightarrow N(\ell).nr = \alpha) \]

BL-GHOST-BLOCK-FULL
\[ \text{GlobalGhost}(\mathcal{M}, \mathcal{N}, V_{up}, V_{cut}) \ast \uparrow n \ell \vdash 0 < n \land \forall m. (\ell + m) \in \text{dom}(\mathcal{M}) \Leftrightarrow m \in [0, n) \]

Figure 6.8: Several agreements between the global ghost state and local assertions

\[
\begin{aligned}
V_{up}' \subseteq V_{up}' & \quad V_{cur}' \subseteq V_{up}' & \quad V_{cut}' \in \mathcal{M} \\
\text{GlobalGhost}(\mathcal{M}, \mathcal{N}, V_{up}, V_{cut}) & \Rightarrow \text{GlobalGhost}(\mathcal{M}, \mathcal{N}, V_{up}', V_{cut}) \ast \text{Seen}(V) \\
\end{aligned}
\]

BL-GHOST-UPDATE-SEEEN
\[
\begin{aligned}
t > \max(\text{dom}(h)) \geq V_{cut}(\ell).w \\
\mathcal{M}' = \mathcal{M}[\ell \leftarrow \mathcal{M}(\ell)[t \leftarrow (v, V)] \\
\mathcal{N}' = \mathcal{N}[\ell \leftarrow \{N(\ell)[w := t]\}] \\
V_{cut}' = V_{cut}[\ell \leftarrow \{V_{cut}(\ell)[w := t]\}] \\
\end{aligned}
\]

\[ \text{GlobalGhost}(\mathcal{M}, \mathcal{N}, V_{up}, V_{cut}) \ast \text{Hist}(\ell, h) \Rightarrow \text{GlobalGhost}(\mathcal{M}', \mathcal{N}', V_{up}', V_{cut}) \ast \text{Hist}(\ell, [t \leftarrow (v, V)]) \]

BL-GHOST-UPDATE-NA-READ
\[
\begin{aligned}
t \geq V_{cut}(\ell).w & \quad \mathcal{M}' = \mathcal{M}[\ell \leftarrow \mathcal{M}(\ell)[t \leftarrow (v, V)] \\
\mathcal{N}' = \mathcal{N}[\ell \leftarrow \{N(\ell)[aw := N(\ell).aw \cup \{t\}]\}] \\
V_{cut}' = V_{cut}[\ell \leftarrow \{V_{cut}(\ell)[aw := V_{cut}(\ell).aw \cup \{t\}]\}] \\
\end{aligned}
\]

\[ \text{GlobalGhost}(\mathcal{M}, \mathcal{N}, V_{up}, V_{cut}) \ast \text{Hist}(\ell, h) \ast \text{Write}_q(\ell, \alpha) \Rightarrow \text{GlobalGhost}(\mathcal{M}', \mathcal{N}', V_{up}', V_{cut}) \ast \text{Hist}(\ell, h[t \leftarrow (v, V)]) \ast \text{Write}_q(\ell, \alpha \cup \{t\}) \]

BL-GHOST-UPDATE-NA-READ
\[
\begin{aligned}
\mathcal{N}' = \mathcal{N}[\ell \leftarrow \{N(\ell)[ar := N(\ell).nr \cup \{r\}]\}] \\
\end{aligned}
\]

\[ \text{GlobalGhost}(\mathcal{M}, \mathcal{N}, V_{up}, V_{cut}) \ast \text{Read}_q(\ell, \alpha) \Rightarrow \text{GlobalGhost}(\mathcal{M}, \mathcal{N}', V_{up}, V_{cut}) \ast \text{Read}_q(\ell, \alpha \cup \{r\}) \]

BL-GHOST-UPDATE-AT-READ
\[
\begin{aligned}
\mathcal{N}' = \mathcal{N}[\ell \leftarrow \{N(\ell)[ar := N(\ell).nr \cup \{r\}]\}] \\
\end{aligned}
\]

\[ \text{GlobalGhost}(\mathcal{M}, \mathcal{N}, V_{up}, V_{cut}) \ast \text{Read}_q(\ell, \alpha) \Rightarrow \text{GlobalGhost}(\mathcal{M}, \mathcal{N}', V_{up}, V_{cut}) \ast \text{Read}_q(\ell, \alpha \cup \{r\}) \]

Figure 6.9: 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 6.18 (State Interpretation for the Base Logic).**

\[
\text{GlobalInv} (\varsigma) ::= \exists V_{up}, V_{cut}. \text{GlobalGhost}(\varsigma, \mathcal{M}, \varsigma, \mathcal{N}, V_{up}, V_{cut}) \ast \\
\varsigma \text{ is wellformed} \ast V_{up} \in \varsigma, \mathcal{M} \ast \varsigma, \mathcal{N} \subseteq V_{cut}
\]

\[
I_{\text{HIST}} ::= \exists \varsigma. \mathcal{M} \upharpoonright \text{view}(\varsigma) \ast \text{GlobalInv} (\varsigma) \upharpoonright \mathcal{N}_{\text{hist}}
\]

\[
S(\varsigma) ::= \mathbf{\text{ex}}(\varsigma) \upharpoonright \text{AUTH} \text{Ex(GlobalState)} \upharpoonright \mathcal{I}_{\text{HIST}}
\]

The global physical state $\varsigma$, together with the logical states $V_{up}$ and $V_{cut}$, need to satisfy some basic properties, as written in the global invariant $\text{GlobalInv}(\varsigma)$: $\varsigma$ 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 $S = \text{Globalinv}$. But recall that $S$ is only accessible with a weakest pre-condition (Definition 5.14), so if we let $S = \text{GlobalInv}$, 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 $\text{GlobalGhost}$, 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” of the logic from a history $h$ to a singleton, without changing the physical memory for $\ell$. To support such rules with not just WPs but also viewshifts, we put $\text{GlobalInv}$ inside an invariant with the fixed namespace $\mathcal{N}_{\text{hist}}$, and employ extra ghost state to maintain that the state $\varsigma$ existentially quantified in the invariant $I_{\text{HIST}}$ is always exactly the parameter $\varsigma$ of $S(\varsigma)$, which is the actual physical state. The RA $\text{AUTH} \text{Ex(GlobalState)}$ ensures that the states agree: valid($\mathbf{\text{ex}}(\varsigma) \circ \text{ex}(\varsigma')$) $\Rightarrow \varsigma = \varsigma'$.

Note that we also need another global ghost location $\gamma_{\text{state}}$, and for every viewshift $\triangleright_{\mathcal{E}}$ that wants to access $\text{GlobalInv}$, we implicitly assume that $\mathcal{N}_{\text{hist}} \subseteq \mathcal{E}$ and the invariant $I_{\text{HIST}}$ is known.

### 6.6 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 (§6.2).** Assuming $\mathcal{N}_{\text{hist}} \subseteq \mathcal{E}$, we use INV-ACC (§5.3) to open $I_{\text{HIST}}$. Note that since the contents of $I_{\text{HIST}}$ are all timeless, the later we get after opening $I_{\text{HIST}}$ can be stripped off right away. We then use BL-Ghost-Update-Hist-drop-singleton to truncate the history with a new cut view $V'_{cut} \supseteq V_{cut} \supseteq \varsigma, \mathcal{N}$. 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. \(\Box\)
Proof sketch of BL-HOARE-ACQ-FENCE (§6.3.1). We start by unfolding the definition of Hoare triples (Definition 5.7 or Notation 6.7), and then of WPs (Definition 5.14). Our goal then looks as follows.

Context: Goal:
See(P) + S(ς)
\[ \xi \not\supset (\text{red}(\text{fence}_{\text{acq}}), \varsigma) \ast \forall(e', V') \ldots \]

Since we have a fancy update \( \xi \not\supset \) in the goal, we can open \( \mathcal{I}_{\text{Hist}} \):
\[ \not\exists (\xi 
\not\supset \) * (GlobalInv(ς) * (\ldots \not\supset \xi \ldots)) \]

We have the following assumptions:
1. (ς \cdot \text{acq})
2. (ς \cdot \text{acq, cur})
3. (ς \cdot \text{acq, cur})
4. (ς \cdot \text{acq, cur})

The obligation \( \text{red}(\text{fence}_{\text{acq}}, V), \varsigma) \) is easily discharged.

For the remaining goal, after introducing assumptions, we know
\[ (i) \varsigma \mid (\text{fence}_{\text{acq}}, V)^{\text{acq}, 1} \varsigma \mid (\varsigma, V') \]

and the goal is
\[ \not\exists (S(ς) \ast V' \subseteq V \ast \text{Seen}(V') \ast V'.cur = V'.acq) \]

Using BL-\text{GHOS}T-\text{SEE}N, we know \( V.acq \subseteq V.upp \), so we can use BL-\text{GHOS}T-\text{UPDATE-SEE}N to get \( \text{Seen}(V') \) without updating \( V.upp \).

We can then use the closing wand viewshift \( (\ldots \not\supset \) to close the invariant without updating \( \varsigma \). The goal is now:
\[ S(ς) \ast \text{Seen}(V') \]

This is easily done by looking at the reduction (i) for acquire fences.

Proof sketch of BL-HOARE-WRITE-AT (§6.3.2). We have the following assumptions:
1. rlx \( \subseteq \) o,
2. Local(ς, h, V.cur),
3. Local(ς, h, V.cur),
4. Local(ς, h, V.cur).

After unfolding Hoare triple and WP definitions, we have the following goal.

Context: Goal:
See(P) + Hist(ς, h) + Write(ς, \alpha, V) + Read(ς, \alpha, V)
\[ \xi \not\supset (\forall(e', V'), \varsigma) \ast \forall \ldots \]

We unfold \( S \) and open the invariant \( \mathcal{I}_{\text{Hist}} \):
\[ \not\exists (\xi \not\supset \) * (GlobalInv(ς) * (\ldots \not\supset \xi \ldots)) \]

We first show safety:
\[ \text{red}(\forall(e': V', \varsigma)) \]

By BL-\text{GHOS}T-\text{HIST} and BL-\text{GHOS}T-\text{NA-READ}, we have
\[ \text{trunc}(\varsigma, M(ς), V.cut(ς)) = h \land h \not\in \text{unalloc}(\varsigma, M) \land \varsigma.N(ς).nr = \alpha. \]

Combining these with (2), (3), and \( \varsigma.N \subseteq V.cut \), we have
\[ \varsigma.N(ς).w \leq V.cut(ς).w \land \varsigma.N(ς).nr \subseteq V.cut(ς).nr. \]

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
\[ (i) \varsigma \mid (\ell := o, V', V) \]

and the goal is
\[ \not\exists (S'(ς) \ast \exists v' \not\in \text{dom}(h), V'.V \not\supset (\forall(o, \ell, v', V') \ldots) \]

Local(ς, \alpha, V) + \text{Hist}(ς, h(ς)) \ast \text{Seen}(\varsigma) \ast \text{Write}(ς, \alpha, V) \ast \text{Read}(ς, \alpha, V) \ast
By looking at the reduction (i), we can get the new timestamp $t'$ and the write message view $V'$, and the fact $V = \bullet_{a,w}, t', \perp, V'$. Additionally, we have $\varsigma.\mathcal{N}(t).aw = \alpha_w$. So we can pick at new cut view $V'_{\text{cut}}$, and use BL-\text{GHOST-UPDATE-AT-WRITE} to update both the history ownership to Hist($\ell, h|t' \leftarrow (v', V')$) and the atomic write ownership to Write$^{\text{rwx}}(\ell, \alpha_w \cup \{t'\})$. By using (4), we also discharge the local observation Local$^{\text{rwx}}(\ell, \alpha_w \cup \{t'\}, V_w \cup V'.\text{cur})$. Using BL-\text{GHOST-UPDATE-SEEN}, we also get Seen($V'$). The non-atomic read ownership Read$^{\text{na}}(\ell, \alpha_r)$ is returned as-is.

We eventually end up with the following goal.

$$\text{GlobalInv}(\varsigma') \ast (\ldots \boxdot \bowtie \ldots)$$

This is easily done by first updating the ghost ownership $[\bullet_{\text{ex}}(\varsigma)]^{\text{TEST}} \ast [\odot_{\text{ex}}(\varsigma)]^{\text{TEST}}$ to that of $\varsigma'$, and then use the closing wand viewshift to close the invariant $I_{\text{HIST}}$.

Finally, we show adequacy for our base logic, which is slightly different from Theorem 5.8 in that we do not fix the initial state.

**Theorem 6.19 (Base Logic Adequacy).** Assuming that the state $\varsigma$ is wellformed and a thread-view $V$ is closed in $\varsigma.\mathcal{M}$ ($V \in \varsigma.\mathcal{M}$), if $\vdash \wp_{\mathcal{T}}(e, V)\{\langle v, \_ \rangle. \phi(v)\}$ is derivable in the base logic logic for $\lambda_{\text{Rust}} + \text{ORC11}$ where $\phi(e)$ is a pure (meta-level) fact, then the following holds.

$$\forall \pi, T', \varsigma'. (\langle \pi \mapsto (e, V) \rangle, \varsigma) \to^* (T', \varsigma') \Rightarrow$$

$$\forall \rho. T(\pi) = v \Rightarrow \phi(v)$$

(BL-\text{ADEQUACY-VAL})

$$\land (\forall \rho, e, V_p, T(\rho) = (e, V_p) \Rightarrow (e, V_p, \text{cur}))$$

(BL-\text{ADEQUACY-NO-STUCK})

**Proof.** The proof follows from a Iris-provided adequacy theorem (that also implies Theorem 5.8). All we need to do is to allocate the various global ghost locations with the correct RAs, and establish the global invariant $I_{\text{HIST}}$ of the state interpretation $S(\varsigma)$, which requires $\varsigma$’s well-formedness and that $V \in \varsigma.\mathcal{M}$. 

---

**Chapter Summary.** In this chapter, we demonstrated the instantiation of Iris with the $\lambda_{\text{Rust}} + \text{ORC11}$ 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 8 and Chapter 9, we will provide more abstractions for the local assertions.
7

vProp: View-monotone Predicates

Following iGPS\textsuperscript{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 $v\text{Prop}$, which stands for view propositions. We call this surface logic of $v\text{Prop}$ 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.

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.\textsuperscript{2} In this chapter, we establish some of such core rules for iRC11.

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 RB\textsubscript{xx} work (Part II), the Cosmo logic,\textsuperscript{3} and the Compass specifications (Part III), chronologically. §7.5 will introduce several modalities to restore explicit view reasoning in the logic of $v\text{Prop}$.

7.1 View-monotone Predicates

We define $v\text{Prop}$ as the type of view-monotone predicates over $i\text{Prop}$.

**Definition 7.1 ($v\text{Prop}$).**

\[
v\text{Prop} ::= \text{View} \xrightarrow{\text{mono}} i\text{Prop}
\]

satisfying $\forall P : v\text{Prop}, \forall V, V'. V \subseteq V' \Rightarrow P(V) \Rightarrow P(V')$ \ ($v\text{PROP-MONO}$)

An assertion $P : v\text{Prop}$ is to be interpreted as some resource that holds at a simple view. This view usually is the current component $V.\text{cur}$ of a thread $\pi$’s thread-view $V$ in case $P$ is owned locally by the thread $\pi$; 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.

\textsuperscript{1}Kaiser et al., “Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris” [Kai+17].


\textsuperscript{3}Mével et al., “Cosmo: a concurrent separation logic for multicore OCaml” [MJP20].
Another choice is to define $vProp$ as a predicate of thread-views ($\text{ThreadView} \xrightarrow{\text{mon}} iProp$), 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 $V$, 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 7.2** (Model of $vProp$ propositions). The function $J \cdot K$ provides a model of $vProp$ propositions into predicates from views to $iProp$, embedding proofs that the predicates are monotone.\(^4\)

\[ J[\phi] := \lambda_v \phi \]
\[ J[\text{False}] := \lambda_v \text{False} \]
\[ J[\text{True}] := \lambda_v \text{True} \]
\[ J[P \rightarrow Q] := \lambda V. \forall V' \sqsupseteq V. \left[ (P)(V') \implies [Q](V') \right] \]
\[ J[P \land Q] := \lambda V. \left[ (P)(V) \land [Q](V) \right] \]
\[ J[P \lor Q] := \lambda V. \left[ (P)(V) \lor [Q](V) \right] \]
\[ J[P \ast Q] := \lambda V. \left[ (P)(V) \ast [Q](V) \right] \]
\[ J[P \rightarrow Q] := \lambda V. \forall V' \sqsupseteq V. \left[ (P)(V') \rightarrow [Q](V') \right] \]
\[ J[\exists x. P] := \lambda V. \exists x. \left[ (P)(V) \right] \]
\[ J[\forall x. P] := \lambda V. \forall x. \left[ (P)(V) \right] \]
\[ J[\text{\Box} P] := \lambda V. \left[ (P)(V) \right] \]
\[ J[\text{\Diamond} P] := \lambda V. \left[ (P)(V) \right] \]
\[ J[\text{\Box}^\gamma P] := \lambda_v \left[ (P)(V) \right] \]
\[ J[\text{\Diamond}^\gamma P] := \lambda_v \left[ (P)(V) \right] \]
\[ J[\text{\Box}^\gamma_\delta P] := \lambda_v \left[ (P)(V) \right] \]

Note that, $J[P \ast Q]$, for example, is view-monotone assuming $P$ and $Q$ are $vProp$ and thus view-monotone. On the other hand, $J[P \rightarrow Q]$ needs to be monotonized explicitly. The model of ghost state ownership $J[\text{\Box}^\gamma_\delta P]$ interestingly simply ignores the input view.

**Lemma 7.3** (Properties of $iRC_{11}$ connectives). The properties in Figure 5.2, Figure 5.3, Figure 5.4, and Figure 5.7 are preserved for $iRC_{11}$ connectives by the encoding of Definition 7.2.
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 8, atomic points-to in Chapter 9, and invariants in Chapter 10. In the remaining, we discuss the models of iRC11 (vProp) weakest pre-conditions and several RMC-specific modalities.

7.2 Model of iRC11 Weakest Pre-conditions

iRC11 WPxs are a vProp proposition that is built upon the base logic WPxs, 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 (§7.5) 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 TVIEWR = AUTH(LAT(ThreadView))), and use ghost locations as thread-ids.

Definition 7.4 (iRC11 Weakest Pre-conditions).

\[ [wp_E e \pi \{ \Phi \}] := \lambda V. \forall V. \forall V'. \forall V. cur \subseteq V. cur \rightarrow [\mathbf{\text{\#}}. TVIEWR]^V \rightarrow \text{Seen}(V) \rightarrow wp_E (e, V) \{ (v, V'), [\mathbf{\text{\#}}. TVIEWR]^V \times [\Phi(v)] \{ V'. cur \} \} \]

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 whose current component \( V. cur \) is in the upward closure of the input view \( V \).
- It threads through the authoritative ghost ownership \( [\mathbf{\text{\#}}. TVIEWR]^V \) of the thread-view being executed with the expression \( e \), in the pre- and post-conditions. This also allows for creating snapshots (fragmentary ownership) \( [\mathbf{\text{\#}}. TVIEWR]^V \) for the lower bound of the executing thread \( \pi \)'s thread-view, which in turn will be used to define release and acquire modalities.
- By hiding thread-views, it also hides the assertion \( \text{Seen}(V) \). Accordingly, it also provides \( \text{Seen}(V) \) as assumption to the base logic WP. It does not require \( \text{Seen}(V') \) 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 7.5 (iRC11 Hoare triples). iRC11 Hoare triples are defined similarly as before.

\[ \{ P \} e \pi \{ v. Q \} \triangleq \square (P \Rightarrow wp_E e \pi \{ v. Q \}) \]
If we interpret this definition in iProp, we will arrive at the following.

\[\{P\} e \in \pi \{v. Q\} \vDash\]

\[\lambda V. \Box \forall V', V. V \subseteq V' \subseteq V. cur \Rightarrow \{P\}(V) \Rightarrow \boxed{\bullet V'}^\pi + \text{Seen}(V)
\]

\[\Rightarrow \wp_{e, e} \{v, V, \boxed{\bullet V'}^\pi + \{Q\}(V'. cur)\}\]

As one can see, generally both the pre-condition \(P\) and post-condition \(Q\) are interpreted at the current components \(V. cur\) and \(V'. cur\), respectively, of the executing thread’s thread-view.

**Lemma 7.6** (Properties of iRC11 WPs and Hoare triples). The properties in Figure 5.5 and Figure 5.6, except those concerning invariants which are not yet defined, also hold for iRC11 WPs and Hoare triples.

**Theorem 7.7** (iRC11 Adequacy). If \(\forall \pi. \vdash \wp_{e, e} e \in \pi \{v. \phi(v)\}\) is derivable in the iRC11 logic for \(\lambda_{Rust} + \text{ORC11}\) where \(\phi(v)\) is a pure (meta-level) fact, then the following holds.

\[\forall \pi, T', \varsigma'. (\{\pi \vdash (e, \varsigma_{\text{init}})\}, \varsigma_{\text{init}}) \vdash^* (T', \varsigma') \Rightarrow
\]

\[\forall e, T(\pi) = v \Rightarrow \phi(v)
\]

(ADEQUACY-VAL)

\[\land \forall \rho, e_{\rho}, V_{\rho}. T(\rho) = (e_{\rho}, V_{\rho}) \Rightarrow (e_{\rho} \text{ is a value} \lor \text{red}((e_{\rho}, V_{\rho}), \varsigma'))
\]

(ADEQUACY-NO-STUCK)

where \(\varsigma_{\text{init}} = (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)\) and \(\varsigma_{\text{init}} = (\emptyset, \emptyset)\).

**Proof sketch.** The proof follows from the base logic adequacy (Theorem 6.19). Note that the initial thread-view \(\varsigma_{\text{init}}\) and state \(\varsigma_{\text{init}}\) are wellformed and \(\varsigma_{\text{init}} \in \varsigma_{\text{init}}\). We then need to allocate the ghost state \(\boxed{\bullet V}^\pi\) for some \(\pi\) and \(V\), and get \(\text{Seen}(V)\), so that we can instantiate and apply our assumption \(\forall \pi. \vdash \wp_{e, e} e \in \pi \{v. \phi(v)\}\) and finish the proof. □

### 7.3 Fence Modalities

To model the effects of relaxed accesses and fences, iRC11 inherits two modalities from FSL\(^5\)—the release modality \(\Delta\) and the acquire modality \(\nabla\)—which allow us to talk about ownership of resources at a thread’s release-fence or acquire views. The assertion \(\Delta_{\rho} P\) represents ownership of \(P\) at thread \(\rho\)’s release-fence view, while the assertion \(\nabla_{\varpi} P\) represents ownership of \(P\) at thread \(\varpi\)’s acquire view.

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 2.1d, Figure 2.2d). We have some resource described by the proposition \(P\) that we want to transfer from the left-hand thread \(\pi\) to the right-hand thread \(\rho\). However, when the “producer” thread \(\pi\) performs its relaxed write, the message view of that write is drawn from \(\pi\)’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 \(\pi\) is sending holds under its release-fence view—that is what is denoted by \(\Delta_{\rho} P\). Dually, when the “consumer” thread \(\rho\) performs its relaxed read, the message view it reads will only be joined into its acquire

view, not its current view. Hence, we need a way of insisting (in the
postcondition of the relaxed read) that $\rho$ 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 9.

Of course, we need a way of actually introducing $\Delta_\pi P$ and eliminating
$\nabla_\rho P$. These steps are achieved by rules \textsc{Hoare-Rel-fence} and \textsc{Hoare-Acq-fence} (Figure 7.1), which allow one to transfer any proposition into the release modality at the point of a \textit{re}1 fence, or out of the acquire modality at the point of an \textit{acq} fence, because those are the points where the current and release-fence/acquire views get synchronized.

\textsc{Hoare-Rel-fence} and \textsc{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 ($\S 7.3.1$).
This can be seen clearly in the model of iRC11 ghost ownership, in Definition 7.2.

6 This can be seen clearly in the model of iRC11 ghost ownership, in Definition 7.2.

**RelMod-Ghost, AcqMod-Ghost, Ghost-RelMod, and Ghost-AcqMod**
together state that the ghost ownership assertion $\Pi^\gamma$ can move freely
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 $\Pi^\gamma$ is view-agnostic
and thus does not care at which view it is interpreted, it is equivalent to
$\Delta^\gamma$ or $\nabla^\gamma$. As a result, $\Pi^\gamma$ can be transferred from one thread
to another without the need for physical synchronization—in particular,
without the need for release/acquire fences.

The remaining rules in Figure 7.1 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.

### 7.3.1 Model of the Fence Modalities

We rely on the extra ghost state of the RA TVIEWR that we have added to
iRC11 WPs (Definition 7.4) to get access to the executing thread’s hidden
thread-view, so that we can give a model to our fence modalities.

**Definition 7.8** (Model of the Fence Modalities).

\[
\begin{align*}
\Delta \Pi & ::= \lambda V. \exists V^\gamma. \Pi^\gamma \cdot [\Pi](V.frel) & \text{(RelMod-Model)} \\
\nabla \Pi & ::= \lambda V. \exists V^\gamma. \Pi^\gamma \cdot [\Pi](V.acq) & \text{(AcqMod-Model)}
\end{align*}
\]

Note that due to validity of TVIEWR, from $\bullet V^\gamma_0 \vdash V^\gamma_1$, we know
that $V \subseteq V'$. Consequently, if we own $\Delta \Pi$ in a goal of a WP for the
thread $\pi$, we know that $\Pi$ holds at the view $V.frel$ where $V \subseteq V'$ and $V'$
is $\pi$’s actual thread-view, and thus by view-monotonicity, $\Pi$ also holds at $V'.frel$. In fact, let us sketch the proofs of some of the rules in Figure 7.1.

**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 7.5), we have the following goal.

\[
\begin{align*}
\text{Context:} & \quad V \subseteq V'.cur \land [\Pi](V) \cdot [\bullet V^\gamma] \cdot \text{Seen}(V) \\
\text{Goal:} & \quad \text{wp}_c(e, V) \left\{(\bullet, V') \cdot [\bullet V^\gamma] \cdot \left[\Delta \Pi \right](V'.cur)\right\}
\end{align*}
\]

We then apply **WP-Mono** (§5.6) and **BL-Hoare-Rel-Fence** (§6.3.1).

The goal, after unfolding the model of the release modality, is now:

\[
\begin{align*}
\text{Goal:} & \quad V \subseteq V'.cur \land V' \subseteq V' \land V'.frel = V'.cur \\
& \quad [\Pi](V) \cdot [\bullet V^\gamma] \cdot \text{Seen}(V') \cdot \left[\bullet V^\gamma_0\right] \cdot \exists V_0. \left[\bullet V^\gamma_0\right] \cdot \left[\Pi\right](V_0.frel)
\end{align*}
\]

We update the ghost thread-view of $\pi$ using **Auth-Lat-Update** (§6.4).

We are then left with:

\[
\begin{align*}
\text{Goal:} & \quad V \subseteq V'.cur \\
& \quad [\Pi](V) \cdot [\bullet V^\gamma] \cdot \exists V_0. \left[\bullet V^\gamma_0\right] \cdot \left[\Pi\right](V_0.frel)
\end{align*}
\]

And then:

\[
\begin{align*}
\text{Goal:} & \quad [\Pi](V') \cdot [\Pi](V'.frel)
\end{align*}
\]
But we know that $V \subseteq V_.cur \subseteq V'_.cur = V'_.frel$. 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 $[P](V_0_.acq) \vdash [P](V'_.cur)$. But by auth-lat-valid and bl-hoare-acq-fence we know that $V_0_.acq \subseteq V'_.acq = V'_.cur$. Again this is done by vprop-mono.

**Proof of RelMod-Ghost**. After unfolding, we arrive at the following base-logic goal: $\exists V. \circ V_.π^∗ a_γ \vdash a_γ$. This is easily done.

**Proof of Ghost-RelMod**. After unfolding, we arrive at the following base-logic goal: $\circ V_.π^∗ a_γ \vdash \exists V. \circ V_.π^∗ a_γ$. We only need to pick any $V$ for which we own $\circ V_.π$, because the ghost ownership does not depend on $V$. With a basic update, we can get ownership of an RA's unit element. In case of TViewR, the unit element is $⊙ ∅$ for the empty thread-view $∅$. Therefore we can easily get $\circ ∅ π$ and we are done.

**Proof of RelMod-Forall**. After unfolding, we arrive at the following base-logic goal: $\exists V. \circ V_.π^∗ ∀ x. J_π K_π(V_.frel) \vdash ∀ x. \exists V. \circ V_.π^∗ J_π K_π(V_.frel)$. 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 $∀∃$ assumption to a $∃∀$ goal.

### 7.4 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 7.9** (Objective Propositions). A proposition $P : vProp$ is objective if its interpretation does not depend any view.\(^7\)

$$\text{objective}(P) ::= ∀ V, V'. [P](V) \vdash [P](V')$$

**Definition 7.10** (The Objective Modality). The objective modality carries the proof that some resource $P$ holds at any view.

$$\llbracket (\text{obj} \ P) \rrbracket ::= \lambda a. ∀ V. [P](V)$$

Figure 7.2 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 ($∀$) in its model. ObjMod-intro allows one to put objective propositions under the objective modality, so that one can store the meta-level objectivity
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 RB21 work (Part II), then developed more formally by the Cosmo logic,⁸ and then used extensively in Compass (Part III).

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.

⁸Mével et al., “Cosmo: a concurrent separation logic for multicore OCaml” [MJP20].
Definition 7.12 (The View-Seen Observations).

The view-seen observation \( \square V \) asserts that the implicit view that is being used to interpret resources is at least \( V \).

\[
\llbracket \square V \rrbracket ::= \lambda V'. V \subseteq V'
\]

The view-seen observation is similar to the seen thread-view observation (Definition 6.9), 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 7.13 (The View-At Modality).

The view-at modality \( \Diamond V P \) asserts that \( P \) holds explicitly (at least) at the view \( V \).

\[
\llbracket \Diamond V P \rrbracket ::= \lambda \_ [P](V)
\]

When we progress through a proof—with or without a program execution (i.e., a WP)—in the \( \mathcal{C} \) 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 \( \Diamond V P \) 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 7.14 (The View-Join Modality).

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.

\[
\llbracket \sqcup V P \rrbracket ::= \lambda V'. [P](V' \sqcup V)
\]

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 7.3 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).
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 \( V \) to observe a view \( V_1 \): it is simply that \( V_1 \subseteq V \).

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 (\( \forall V \)). As such, we can send \( @_V P \) and \( \forall V \) 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 7.15** (Alternative Model for Fence Modalities). The fence modalities are in fact defined in VProp using the view-at modality.

\[
\Delta P := \exists V. \forall^* \pi \ast \varnothing V P \\
\nabla P := \exists V. \forall^* \pi \ast \varnothing V P
\]

After unfolding into the base logic, this is exactly the same as Definition 7.8.

We note that the release modality and the view-at modality can interact through the following rule.

\[
\text{RelMod-VA-revert} \\
\forall V. \{ \forall^* \pi \ast \Delta P \sqsubseteq V \} \in \{ \Phi \} e \\
\{ \Delta P \} \in \{ \Phi \} e
\]

That is, with a goal in the form of a WP for the thread \( \pi \), we can turn the assumption \( \Delta P \) into \( \forall^* \pi \ast \Delta \sqsubseteq V \) for some view \( V \).

### 7.6 The Subjective Modality

Finally, we introduce the subjective modality, a derivation from the view-at modality.

**Definition 7.16** (The Subjective Modality).

\[
\langle \text{subj} \rangle P := \exists V. \forall^* \pi P
\]

That is, the subjective modality asserts that \( P : \forall \text{Prop} \) 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 7.4, 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 \( \forall \text{Prop} \) is the view lattice.
### 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.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{SUBJMod-VA}$</td>
<td>$\exists V. @V P \vdash (\text{subj}) P$</td>
</tr>
<tr>
<td>$\text{SUBJMod-ELIM-OBJ}$</td>
<td>$\text{objective}(P) \vdash (\text{subj}) P \vdash P$</td>
</tr>
<tr>
<td>$\text{SUBJMod-AND}$</td>
<td>$(\text{subj})(P \land Q) \vdash (\text{subj}) P \land (\text{subj}) Q$</td>
</tr>
<tr>
<td>$\text{SUBJMod-OR}$</td>
<td>$(\text{subj})(P \lor Q) \vdash (\text{subj})(P \lor Q)$</td>
</tr>
<tr>
<td>$\text{SUBJMod-FORALL}$</td>
<td>$(\text{subj}) \forall x. P \vdash \forall x. (\text{subj}) P$</td>
</tr>
<tr>
<td>$\text{SUBJMod-EXIST}$</td>
<td>$\exists x. (\text{subj}) P \vdash (\text{subj}) \exists x. P$</td>
</tr>
</tbody>
</table>
| $\text{SUBJMod-SEP}$ | $(\text{subj})(P 
 Q) \vdash (\text{subj}) P * (\text{subj}) Q$ |
| $\text{SUBJMod-LATER}$ | $\triangleright (\text{subj}) P \vdash (\text{subj}) \triangleright P$ |
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 \( \ell \), which allows for safe, non-racy operations on \( \ell \). For concurrent reads, the assertion can be equipped with fractional permission, i.e., \( \ell \mapsto_{q} v \). 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 \( \ell \mapsto_{q} v \) 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 6. In Chapter 9, we will also discuss iRC11’s ability to switch between non-atomic and atomic points-to assertions.

8.1 The Interface of Non-Atomic Points-To

The interface of iRC11 non-atomic points-to is rather standard, as given in Figure 8.1. \textsc{Na-frac}, \textsc{Na-frac-valid}, and \textsc{Na-frac-agree} together say that non-atomic points-to is fractional, and \textsc{Na-excl} says that full ownership of a non-atomic points-to is exclusive. \textsc{Na-read} allows us to perform non-racy reads using any access mode \( o \) with a fraction \( \ell \mapsto_{q} v \), and we are guaranteed the return value is \( v \). \textsc{Na-write} allows us to perform non-racy writes also using any access mode \( o \) with the full fraction \( \ell \mapsto _{\cdot} \), and we know afterwards \( \ell \) 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, \textsc{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 \((\bigstar_{m \in [0,n]} \ell + m \mapsto \bigstar)\) for all locations of the newly allocated block, whose base location is \( \ell \). Conversely, \textsc{Na-dealloc} consumes the block ownership and the points-to ownership of all locations. We strengthen \textsc{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 \( \ell \), but is not synchronized with that write, i.e., it has not observed the write’s message.

\(^{1}\)see Definition 6.1
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.

### 8.2 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 6.1) to vProp as follows.

**Definition 8.1** (Lifting Local Assertions to vProp).

$$[\text{Hist}(\ell, h)] := \lambda_. \text{Hist}(\ell, h)$$

$$[\text{Write}^{\text{rlx}}(\ell, \alpha)] := \lambda_. \text{Write}^{\text{rlx}}(\ell, \alpha)$$

$$[\text{Read}^{\text{na}}(\ell, \alpha)] := \lambda_. \text{Read}^{\text{na}}(\ell, \alpha)$$

$$[\text{Local}^{\text{rlx}}(\ell, h)] := \lambda V. \text{Local}^{\text{rlx}}(\ell, h, V)$$

$$[\text{Local}^{\text{na}}(\ell, h)] := \lambda V. \text{Local}^{\text{na}}(\ell, h, V)$$

$$[\text{Local}^{\text{na}}(\ell, \alpha, V_{na})] := \lambda V. \text{Local}^{\text{na}}(\ell, \alpha, V_{na}) \land V_{na} \subseteq V$$

The lifting is straightforward. Recall that the various local ownership for parts of the race-detector state are purely ghost state, 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 6.5 that the local observations are all view-monotone.

**Remark 8.2** (The non-atomic view $V_{na}$). Note that unlike the rest of iRC11 local observations, we do not hide the view $V_{na}$ of the non-atomic local observation $\text{Local}^{\text{na}}(\ell, \alpha, V_{na})$. Instead, we require that the implicit interpreting view $V$ includes $V_{na}$. The view $V_{na}$ 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 \( \ell \) 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 \( \ell \) can happen (logically) much later than the most recent physical non-atomic operation to \( \ell \). 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 9.

In this chapter, we can simply ignore this view.

**Definition 8.3** (Model of \( \ell \mapsto v \)). We define a primitive non-atomic points-to \( \ell \mapsto v \) which represents fractional ownership of \( \ell \) with a history \( h \), and then use it to define the unsynchronized non-atomic points-to \( \ell \mapsto v ? \) and the actual points-to \( \ell \mapsto v \).

\[
\ell \mapsto_{na} h \ ::= \exists \alpha_w, \alpha_1, \alpha_2. \text{Local}_h(\ell, h) \ast \text{Local}_r^r(\ell, \alpha_w) \ast \\
\text{Local}_r^r(\ell, \alpha_2) \ast (\exists V_{na}. \text{Local}^na_r(\ell, \alpha_1, V_{na})) \ast \\
\text{Hist}_q(\ell, h) \ast \text{Write}^r(\ell, \alpha_w) \ast \\
\text{Read}^na_r(\ell, \alpha_1) \ast \text{Read}^r(\ell, \alpha_2)
\]

\[
\ell \mapsto _? \ ::= \exists t, v, V^?. \ell \mapsto_{na} [t \leftarrow (v, V^?)]
\]

\[
\ell \mapsto v \ ::= \exists t, V^?. \ell \mapsto_{na} [t \leftarrow (v, V^?)] \ast \exists V^?
\]

A fraction \( q \) of the primitive non-atomic points-to for \( \ell \) contains the corresponding fractions for \( \ell \)'s history ownership of \( h \) and the parts of race-detector state. By \( \text{Local}_h(\ell, h) \), the owner of \( \ell \mapsto_{na} h \) has also observed the allocation of \( \ell \). The sets \( \alpha_w \), \( \alpha_1 \), and \( \alpha_2 \) 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 \( \ell \mapsto_{na} h \).

The unsynchronized non-atomic points-to \( \ell \mapsto ? \) then simply requires that the history be a singleton \( [t \leftarrow (v, V^?)] \) for \( \ell \)'s latest write event \( (t, v, V^?) \). The non-atomic points-to \( \ell \mapsto v \) 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 6.2. 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 $iRC_{11}$ (Definition 7.4) and of the base logic
(Definition 5.14), and work directly with the state interpretation (Definition 6.18), 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 \( \ell \)—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 \( \dag \ell \), 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 \( \hat{\bullet} \), which is lifted to the poison value \( \bullet \) in $iRC_{11}$.

Proof sketch of $\text{NA-DEALLOC}$. After unfolding the WP definitions of both $iRC_{11}$ 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 $\text{BL-\textit{HOST}}-\text{BLOCK-\textit{FULL}}$ (Figure 6.8, §6.5) 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 $\text{OM-FREE}$ (§3.3) and $\text{DRF-DEALLOC}$ (§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 $\text{None}$.\( \square \)

Proof sketch of $\text{NA-READ}$. We only need to unfold the definitions of the non-atomic points-to (Definition 8.3) and the $iRC_{11}$ Hoare triples and WPs (Definition 7.5 and Definition 7.4), and perform the proof in the base logic. Recall that by Definition 7.5, all of our resources are interpreted at the current component $V.\text{\textit{cur}}$ of the thread-view $V$.

- In case $o = \text{na}$, we apply $\text{BL-HOARE-READ-NA}$ (§6.3.2). Note that $V_r$ is instantiated to $V_\text{na}$. In the post-condition we only need to use the post-condition of $\text{BL-HOARE-READ-NA}$ to address the only change by the read, which is the non-atomic reads set and its local observation.

- In case $o \sqsupseteq \text{rlx}$, we apply $\text{BL-HOARE-READ-AT}$ (§6.3.2). Note that the history in the non-atomic points-to is a singleton, and $V_r$ is instantiated to $V.\text{\textit{cur}}$, so the proof is straightforward.\( \square \)

Proof sketch of $\text{NA-WRITE}$. The proof is similar to that of $\text{NA-READ}$. We use $\text{BL-HOARE-WRITE-NA}$ in case $o = \text{na}$. Otherwise, if $o \sqsubseteq \text{rlx}$, we use $\text{BL-HOARE-WRITE-AT}$, and then use $\text{BL-HIST-DROP-SINGLETON}$ (Figure 6.2, §6.2) to shrink the history back to a singleton.\( \square \)
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 II). 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 III).

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 \(\ell \rightarrow_{at} (v, V)\) represents Cosmo’s ownership of an atomic location \(\ell\) 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.

In the presence of concurrent writes to the same location \(\ell\), iRC11 rules for handling \(\ell\)’s history are rather cumbersome and hard to use. In practice, if a client performs arbitrary concurrent writes to a location \(\ell\), then the concurrent protocol for \(\ell\) 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.\(^3\) Interesting algorithms tend to perform concurrent accesses with more reasonable protocols: 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 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 §9.1, 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 §9.2, we give the model of the atomic points-to assertion, built atop the local assertions of the base logic (§6.2).

---

\(^1\)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].

\(^2\)Mével et al., “Cosmo: a concurrent separation logic for multicore OCaml” [MJP20].

\(^3\)see OM-WRITE, §3.3
9.1 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).

\[ \theta \in \text{AtomicMode} ::= \text{con} \mid \text{sw} \mid \text{cas}. \]

The atomic points-to with the arbitrarily concurrent mode \( \text{con} \) 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 \( \text{sw} \), all writes using the atomic points-to must be made sequential (synchronized), but reads can be arbitrarily concurrent. In the CAS-only mode \( \text{cas} \), the atomic points-to only supports CASes to write, and reads can be arbitrarily concurrent.

**Definition 9.1 (Atomic Points-To Assertion).** The atomic points-to assertion has the form \( \ell \mapsto_{\theta} h \) where (1) \( \theta \) is one of the 3 atomic modes; and (2) \( \emptyset \neq h \in \text{History} \) is \( \ell \)'s current history which contains write events still visible to accessing threads; and (3) \( \gamma \) 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 \( \ell \mapsto_{\theta} h ::= \exists t. \ell \mapsto_{\theta}^{t_x} h \) to ignore the exclusive single-writer timestamp.

**Definition 9.2 (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 \( \ell \sqsubseteq_{\text{sn}} h \) asserts the observation of all \( \ell \)'s write events in the non-empty history \( h \). This observation is the minimum requirement to perform an atomic read on \( \ell \).
- The history-sync observation \( \ell \sqsubseteq_{\text{sy}} h \) asserts not only the observation of \( \ell \)'s write events of in \( h \), but also the observation of those writes’ message views.
- The single-writer ownership \( \ell \sqsubseteq_{\text{sw}} h \) asserts the exclusive permission to write (the single-writer) to \( \ell \), and the history-sync observation of \( h \) (i.e., \( \ell \sqsubseteq_{\text{sy}} h \)). The single-writer ownership guarantees that \( h \) is the current history of \( \ell \).
- The fractional CAS ownership \( \ell \sqsubseteq_{t_x,q}^{\text{cas}} h \) asserts the shared permission to CAS to \( \ell \), and the history-seen observation of \( h \) (i.e., \( \ell \sqsubseteq_{\text{sn}} h \)). A fraction \( q \) of the CAS ownership only guarantees that \( h \) is the sub history of \( \ell \)'s current history. The timestamp \( t_x \) is of the latest exclusive single-writer write to \( \ell \). As usual, we write \( \ell \sqsubseteq_{\text{cas}} h \) to hide this timestamp, and write \( \ell \sqsubseteq_{\text{cas}} h \) for the full ownership where \( q = 1 \).
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 the atomic mode $\theta$.

Figure 9.1: Basic properties of assertions related to the atomic points-to

Property 9.3 (Basic Properties ofAssertions Related to Atomic Points-To). Figure 9.1 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-split 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 $\theta$ is not the concurrent mode 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 $\theta = \text{cas}$. 

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT-sw-excl</td>
<td>$\ell \models_{\text{sw}} h \land \ell \not\models_{\text{sn}} h'$ $\vdash$ False</td>
</tr>
<tr>
<td>AT-sw-agree</td>
<td>$\ell \models_{\text{w}} h \land \ell \not\models_{\text{sn}} h'$ $\vdash$ $\theta = \text{sw} \land h = h'$</td>
</tr>
<tr>
<td>AT-cas-frac-agree</td>
<td>$\ell \not\models_{\text{cas}} h \land \ell \not\models_{\text{cas}} t' + q'$ $\land \theta \not= \text{con} \land t = t' \land q' \subseteq h$</td>
</tr>
<tr>
<td>AT-cas-split</td>
<td>$\ell \not\models_{\text{cas}} h \land \ell \not\models_{\text{cas}} t' + q'$ $\vdash$ $\ell \not\models_{\text{cas}} h \land \ell \not\models_{\text{cas}} h'$</td>
</tr>
<tr>
<td>AT-sy-unfold</td>
<td>$h(t) = (v, V)$ $\ell \models_{\text{sy}} h \vdash \exists V \exists \ell \leftarrow \emptyset \left[w := t\right]$</td>
</tr>
<tr>
<td>AT-sy-joint</td>
<td>$\ell \models_{\text{sn}} h \land \ell \not\models_{\text{sy}} h' \vdash \ell \models_{\text{sy}} (h \cup h')$</td>
</tr>
</tbody>
</table>
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 \( ℓ \supseteq_{\text{cas}} h \) 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 7.12). 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 9.4 (Conversions Between Non-Atomic and Atomic Points-To).**
The top three rules of Figure 9.2 present the rules for converting the non-atomic points-to ownership to the atomic one. The bottom of Figure 9.2 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 \( V (\supseteq V)\).

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 7.13); 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 \([t ← (v, V)]\) 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 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 (§6.2), which
The Interface of the Atomic Points-To Assertion

$$\text{NA-AT-sw}$$

$$\ell \mapsto v \Rightarrow \exists \gamma, t, V. \exists V \ni \ell \mapsto \gamma_v^t [t \leftarrow (v, V)] * \ell \mapsto \gamma_v^t [t \leftarrow (v, V)]$$

$$\text{NA-AT-sw-view}$$

$$\ell \mapsto v * P \Rightarrow \exists \gamma, t, V. \exists V \ni @V (P * \ell \mapsto \gamma_v^t [t \leftarrow (v, V)] * \ell \mapsto \gamma_v^t [t \leftarrow (v, V)])$$

$$\text{AT-NA}$$

$$\ell \mapsto \gamma_v^t h \Rightarrow \exists t_0, v, V. \exists V \ni \ell \mapsto v * h(t) = (v, V) * t = \max(\text{dom}(h))$$

$$\text{AT-con-sw}$$

$$\ell \mapsto \gamma_v^t h \Rightarrow \exists t = \max(\text{dom}(h)), \ell \mapsto \gamma_v^t h * \ell \mapsto \gamma_v^t h$$

$$\text{AT-cas-sw}$$

$$\ell \mapsto \gamma_v^t h * \ell \mapsto \gamma_v^t h' \Rightarrow \exists t = \max(\text{dom}(h)), \ell \mapsto \gamma_v^t h * \ell \mapsto \gamma_v^t h$$

$$\text{AT-sw-con}$$

$$\ell \mapsto \gamma_v^t h * \ell \mapsto \gamma_v^t h' \vdash \ell \mapsto \gamma_v^t h$$

$$\text{AT-sw-cas}$$

$$\ell \mapsto \gamma_v^t h * \ell \mapsto \gamma_v^t h' \vdash \ell \mapsto \gamma_v^t h * \ell \mapsto \gamma_v^t h$$

$$\text{AT-con-cas}$$

$$\ell \mapsto \gamma_v^t h \vdash \ell \mapsto \gamma_v^t h * \ell \mapsto \gamma_v^t h$$

$$\Rightarrow \ell \mapsto \gamma_v^t h$$

$$\Rightarrow \ell \mapsto \gamma_v^t h$$

$$\Rightarrow \ell \mapsto \gamma_v^t h$$

$$\Rightarrow \ell \mapsto \gamma_v^t h$$

Figure 9.2: Conversions between the non-atomic and atomic points-to assertion

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 $\text{AT-sy}$ and $\text{AT-sy-unfold}$, we know that we have observed the latest write message view $V (\exists V)$.\(^4\)

**Cycles of Alternating Non-Atomic and Atomic Periods.** We note that we have made the ghost location $\gamma$ explicit in these rules, who signify its role. In the model of atomic points-to, $\gamma$ 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 $\gamma$ uniquely identifies an atomic access period of the location $\ell$. When we use the rule $\text{NA-AT-sw}$ (or $\text{NA-AT-sw-view}$) to switch from non-atomic to atomic points-to, we receive a fresh location $\gamma$ that identifies and enforces the atomic protocol for the current atomic period of $\ell$. As such, the atomic local ownership and observations (Definition 9.2) with the ghost location $\gamma$ are only meaningful when we still have access to the atomic points-to $\ell \mapsto \gamma_v^t h$ with the same $\gamma$. If we use $\text{AT-NA}$ to turn $\ell \mapsto \gamma_v^t h$ back to a non-atomic points-to, we can see that the atomic local ownership and observations with ghost location $\gamma$ are not needed, and
in fact $\gamma$ is simply forgotten, and afterwards the atomic local ownership and observations of $\gamma$ 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 $\gamma'$. This life-cycle can probably be understood better if we look at the visualization graph in Figure 9.2 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 $\gamma$. 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 9.5 (Conversions Between Modes of Atomic Points-To).** The rest of Figure 9.2 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, 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 (§8.1) which support atomic access modes.

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 10). Therefore, they only assume ownership of the atomic points-to under a view-at modality (Definition 7.13), i.e., $\@_V_h(\ell \mapsto_\theta h)$ where the view $V_h$ has no clear relation to what the executing thread has observed. In particular, the rules will have the following form.

$$\{\exists V \in \pi \forall v. \exists V', h'. \exists Q. \forall V_h(\ell \mapsto_\theta h') Q\}$$

That is, the view $V_h$ 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 $V_h$, except that it also contributes to $V_h$. Most importantly, the view $V_h$ will be used later to enable synchronized, safe switching back to non-atomic points-to (using AT-NA).

### 9.1.1 Atomic Read Rules

Several rules for atomic reads are given in Figure 9.3. AT-read-sn, AT-read-sync, AT-read-cas, and AT-read-sw allow reading with a history-seen observation, a history-sync observation, a fractional CAS ownership, and
We note that we will have observed a bigger snapshot history $h'$ in $\pi$ than what it has observed ($t, v, V$) due to $V \subseteq V'$ and $VS$-MONO ($\S$7.5). We note that $V' \subseteq V'$ if this is an acquire read, then we know that the thread has observed the read message view $V$, due to $V \subseteq V'$ and $VS$-MONO ($\S$7.5). We note that $V' \subseteq V'$ if this is a relaxed read, then the message view $V$ will be at least $V'$ for some view $V_0$. The post-condition says that the executing thread $\pi$ will read a message $(t, v, V)$ which is no earlier than what it has observed ($t \geq \max(\text{dom}(h_0))$), and afterwards the thread will have observed a bigger snapshot history $h'$ that contains the read message $(\ell \not\subseteq h')$. After the read, the current view of $\pi$ will be at least $V'$, and if this is an acquire read then we know that the thread has observed the read message view $V$, due to $V \subseteq V'$ and $VS$-MONO ($\S$7.5). We note that $V' \subseteq V'$ if this is a relaxed read, then the message view $V$ will

**AT-read-sn**

$\text{RLx} \subseteq o$

$$\{ \exists V_0 * \ell \not\subseteq h_0 * \not\subseteq V_0 (\ell \not\subseteq h_0) \}^{\ast \ell} \in \pi$$

$$(\forall h', t, V, V' \not\subseteq V_0 \subseteq h' \subseteq h \ast)$$

$h'(t) = (v, V) * t = \max(\text{dom}(h_0)) * (\forall o \not\subseteq \text{acq} ? V \not\subseteq V' : \not\subseteq V_0 (\ell \not\subseteq h_0))$

$\exists V' * \not\subseteq V_0 (\ell \not\subseteq h_0) * \not\subseteq V_0 (\ell \not\subseteq h_0)$

$\text{AT-read-sn-acq}$

$\{ \exists V_0 * \ell \not\subseteq h_0 * \not\subseteq V_0 (\ell \not\subseteq h_0) \}^{\ast \text{acq} \ell} \in \pi$

$$(\forall h', t, V, V' \not\subseteq V_0 \not\subseteq V_0 \subseteq h' \subseteq h \ast)$$

$h'(t) = (v, V) * t = \max(\text{dom}(h_0)) * (\forall o \not\subseteq \text{acq} ? V \not\subseteq V' : \not\subseteq V_0 (\ell \not\subseteq h_0))$

$\exists V' * \not\subseteq V_0 (\ell \not\subseteq h_0) * \not\subseteq V_0 (\ell \not\subseteq h_0)$

$\text{AT-read-cas}$

$\text{RLx} \subseteq o$

$$\{ \exists V_0 * \ell \not\subseteq h_0 * \not\subseteq V_0 (\ell \not\subseteq h_0) \}^{\ast \ell} \in \pi$$

$$(\forall h', t, V, V' \not\subseteq V_0 \subseteq h' \subseteq h \ast)$$

$h'(t) = (v, V) * t = \max(\text{dom}(h_0)) * (\forall o \not\subseteq \text{acq} ? V \not\subseteq V' : \not\subseteq V_0 (\ell \not\subseteq h_0))$

$\exists V' * \not\subseteq V_0 (\ell \not\subseteq h_0) * \not\subseteq V_0 (\ell \not\subseteq h_0)$

$\text{AT-read-sy}$

$\text{RLx} \subseteq o$

$$\{ \exists V_0 * \ell \not\subseteq h_0 * \not\subseteq V_0 (\ell \not\subseteq h_0) \}^{\ast \ell} \in \pi$$

$$(\forall h', t, V, V' \not\subseteq V_0 \subseteq h' \subseteq h \ast)$$

$h'(t) = (v, V) * t = \max(\text{dom}(h_0)) * (\forall o \not\subseteq \text{acq} ? V \not\subseteq V' : \not\subseteq V_0 (\ell \not\subseteq h_0))$

$\exists V' * \not\subseteq V_0 (\ell \not\subseteq h_0) * \not\subseteq V_0 (\ell \not\subseteq h_0)$

$\text{AT-read-sw}$

$\text{RLx} \subseteq o$

$$\{ \exists V_0 * \ell \not\subseteq h_0 * \not\subseteq V_0 (\ell \not\subseteq h_0) \}^{\ast \ell} \in \pi$$

$$(\forall h', t, V, V' \not\subseteq V_0 \subseteq h' \subseteq h \ast)$$

$h'(t) = (v, V) * t = \max(\text{dom}(h_0)) * (\forall o \not\subseteq \text{acq} ? V \not\subseteq V' : \not\subseteq V_0 (\ell \not\subseteq h_0))$

$\exists V' * \not\subseteq V_0 (\ell \not\subseteq h_0) * \not\subseteq V_0 (\ell \not\subseteq h_0)$

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 9.3 are derived from it. The rule assumes in the pre-condition a local history-seen observation $\ell \not\subseteq h_0$ for some snapshot history $h_0$ of $\ell$, and the shared atomic points-to $\ell \not\subseteq h_0$ of the current history $h$ at some view $V_0$. The pre-condition also includes a view-seen observation $\exists V_0$ for some view $V_0$. The post-condition says that the executing thread $\pi$ will read a message $(t, v, V)$ which is no earlier than what it has observed ($t \geq \max(\text{dom}(h_0))$), and afterwards the thread will have observed a bigger snapshot history $h'$ that contains the read message $(\ell \not\subseteq h')$. After the read, the current view of $\pi$ will be at least $V'$, and if this is an acquire read then we know that the thread has observed the read message view $V$, due to $V \subseteq V'$ and $VS$-MONO ($\S$7.5). We note that $V' \subseteq V'$ if this is a relaxed read, then the message view $V$ will

![Figure 9.3: iRC11 read rules with the atomic points-to assertion](image-url)
9.1.2 Atomic Write Rules

Several rules for atomic writes are given in Figure 9.4. **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 $\forall V_0$ for some view $V_0$, a history-seen observation $\ell \sqsubseteq_{sn} h_0$ for some snapshot history $h_0$ of $\ell$, and the atomic points-to in the concurrent mode $\ell \sqsubseteq_{con} h$, shared at some view $V_0$. The pre-condition additionally requires, in case the write is a relaxed write, a view-seen observation $\forall V_{rel}$ of some view $V_{rel}$ under the release modality, i.e., $\Delta_\ell (\forall V_{rel})$. This assertion ensures that the view $V_{rel}$ has been observed by the thread $\pi$ at its most recent release fence, so the message view of the relaxed write to be performed is guaranteed to include $V_{rel}$. In other words, $V_{rel}$ 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, $\pi$’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 $\ell \sqsubseteq_{con} h[t \leftarrow (v, V)]$ is returned, at the extended view $V_0 \sqcup V'$ where $V'$ is the thread $\pi$’s current view after the write. Note that the timestamp $t$ for the new write message must be fresh in $h$ ($t \notin \text{dom}(h)$), and must be mo-later than the events that the thread has observed for $\ell (\max(\text{dom}(h_0)) < t)$. 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 ($V_0 \not\sqsubset V$), because $V$ contains at least the new timestamp $t$ 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 $V_{rel}$ is a lower bound for $V$ ($V_{rel} \sqsubseteq V$). In any case the message view is no more than the thread’s current view after the
**AT-write-sn**

\[\exists t, V, V' \ni V_0 \supseteq V. \max(\text{dom}(h_0)) < t \notin \text{dom}(h) *\]

\[\{ o = \text{rlx} ? \} \Delta_\pi (\exists \text{Vrel} : \text{True}\} \] 

\[\ell := o \in \pi\]

**AT-write-cas**

\[\exists t, V, V' \ni V_0 \supseteq V. \max(\text{dom}(h_0)) < t \notin \text{dom}(h) *\]

\[\{ o = \text{rlx} ? \} \Delta_\pi (\exists \text{Vrel} : \text{True}\} \] 

\[\ell := o \in \pi\]

**AT-write-sw**

\[\exists t, V, V' \ni V_0 \supseteq V. \max(\text{dom}(h_0)) < t *\]

\[\{ o = \text{rlx} ? \} \Delta_\pi (\exists \text{Vrel} : \text{True}\} \] 

\[\ell := o \in \pi\]

**AT-write-sw-rlx**

\[\exists t, V, V' \ni V_0 \supseteq V. \max(\text{dom}(h_0)) < t *\]

\[\{ o = \text{rlx} ? \} \Delta_\pi (\exists \text{Vrel} : \text{True}\} \] 

\[\ell := o \in \pi\]

**AT-write-sw-rlx-simple**

\[\{ \ell \supseteq h @ V_0 (\ell \supseteq h) * P \} \ell := \text{rlx} \in \pi\]

**AT-write-sw-rel**

\[\exists t, V \ni V_0. \max(\text{dom}(h_0)) < t * \]

\[\{ o = \text{rlx} ? \} \Delta_\pi (\exists \text{Vrel} : \text{True}\} \] 

\[\ell := o \in \pi\]

---

Step \((V \subseteq V')\). Finally, after the step the thread extends its observation on \(\ell\)'s history by the write event it has just performed. That is, it owns \(\ell \supseteq h_0[t \leftarrow (v, V)]\), 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 \((\exists V_0 (\ell \supseteq h_0[t \leftarrow (v, V)])\).

**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-sy** 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 $V_{rel}$ 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 $\forall V P$. 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 10). 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 $\exists V W$, which it then can combine with $\forall V P$ (taken from the invariant) and apply the rule VA-elim (§7.5) to acquire the resource $P$ locally, and thus conclude the resource transfer.

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 (§7.5) 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.

9.1.3 Atomic CAS Rules

The general rule AT-cas-sn-gen in Figure 9.5 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 $\forall t_0, t_0 \geq \max (\text{dom}(h_0)). h(t_0) = (v_0, _) \Rightarrow \vdash v_0 = ^7 v_r$ is required to guarantee safe comparison between any readable value $v_0$ and the expected value $v_r$, and the resources concern $P_{\text{cmp}}$ are needed for deterministic pointer comparison. Please see also the explanation of the base logic rule BL-Hoare-CAS in §6.3.3.

In particular, if the expected value $v_r$ is a location value $\ell_r$, to guarantee deterministic pointer comparison, $P_{\text{cmp}}$ is required to simultaneously imply (with a basic update) that (1) some primitive non-atomic points-to ownership of $\ell_r$ at an arbitrary view, sufficient to show that $\ell_r$ is alive; and (2) for any location value $\ell'$ that the thread may read from $h$, $P_{\text{cmp}}$ is also sufficient to show that $\ell'$ 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, $P_{\text{cmp}}$ is returned unchanged, and we know that the thread’s current view after the step is $V'$.

In case of failure, i.e., $b = \text{false}$, 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 $V_h \cup V'$.
The Interface of the Atomic Points-To Assertion

\[ \text{AT-cas-sn-gen} \]

\[
\begin{align*}
&\text{rlx} \subseteq o_f, o_r, o_w \quad \forall (h_t, t_0) \geq \max(\text{dom}(h_0)), h(t_0) = (h_t, \_ ) \Rightarrow h_t \equiv v_r \\
&\exists V_0 \ni \ell \ni h_0 \ni @V_0(\ell \ni h_0) \ni (a = \text{rlx}) \Rightarrow (V \ni \Delta_v (\exists V_{\text{rel}})) \ni \text{True} \\
P_{\text{cmap}} \ni \Box ((v_r = \ell_r) \Rightarrow P_{\text{cmap}} \ni \Phi_{\text{cmap}}(\ell_r, h)) \ni \text{True}
\end{align*}
\]

\[ \text{CAS}^{v_r, o_r, o_w}(\ell, v_r, v_w) \ni \pi \ni \]

\[
\begin{align*}
&b \ni \exists \ell', v_r, V_r, V' \ni V_0, h_0, h', h_0 \ni h_0(\ell') = (v', V_r) \ni \max(\text{dom}(h_0)) \leq t' + 1 \\
P_{\text{cmap}} \ni \Box V' \ni @V(\ell \ni h_0) \ni @V_{\text{ VL}}(\ell \ni h_0) \ni (a = \text{rlx}) \ni (a = \text{rlx}) \\
&\ni \ni (b = \text{false} \ni v_r \neq v' \ni h' = h \ni (a_f \ni \text{acq}) \ni V_r \ni V' : \nabla_v (\exists V_r) \\
&\ni b = \text{true} \ni v_r = v' \ni \exists t \ni \text{dom}(h), V_w \ni V_r \ni V' \ni V_0, t = t' + 1 \\
&\ni \ni (\exists h' = h[t \ni (v_w, V_w)]) \ni h_0(\ell) = (v_w, V_w) \\
&\ni \ni (a = \text{rlx}) \ni V_w \ni V' : \nabla_v (\exists V_w) \ni (a = \text{rlx}) \ni V' \ni V_w : V_{\text{rel}} \ni V_w \ni \xi
\end{align*}
\]

where

\[ \Phi_{\text{cmap}}(\ell_r, h) := (\exists q_r, h_r, V_r, V \ni @V(\ell_r \ni h_r)) \ni (\forall t' \ni \text{dom}(h), \ell', h(t') = (t', \_ )) \ni \exists q_r', h', V_r \ni @V(\ell' \ni h') \]

\((@V_{\text{ VL}}(\ell \ni h_0)), \) and that the thread will have observed the new snapshot history \(h_0 \ni (@V(\ell \ni h_0))\). Furthermore, the read message view \(V_r\) is observed according to the failure read access mode \(a_f\). If \(a_f\) is at least an acquire mode, then \(V_r\) is included in the view \(V'\) after the step \((V_r \ni V')\). Otherwise, the observation of \(V_r\) is only available after the next acquire fence \((\nabla_v \ni V_r)\).

If the CAS succeeds, i.e., \(b = \text{true}\), 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 \((t \ni (v_w, V_w))\) where \(t\) is fresh in \(h\) \((t \ni \text{dom}(h))\). \(V_w\) 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 \(a_r\): if \(a_r\) is at least an acquire mode, then \(V_w \ni V'\), otherwise the thread only has \(\nabla_v (\exists V_w)\). If the write access mode \(a_w\) is at least release, then the write message view includes the thread’s current view after the step \((V' \ni V_w)\), otherwise \(V_{\text{rel}}\) 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., \(a_w \ni \text{rel} \ni o_r \ni \text{acq} \ni V' = V_w\).

Finally, we note that \text{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 \((\text{cas})\), we can apply \text{AT-con-cas} (Figure 9.2) to support CASes with the atomic points-to in the concurrent mode \(\text{con}\). With a fractional CAS ownership, we can also apply \text{AT-cas-sn-gen}, thanks to \text{AT-cas-sn}.

If the atomic points-to is in the single-writer mode \((\text{sw})\), then, together with the single-writer ownership \(\ell \ni h\), we can get to the atomic points-to in CAS-only mode thanks to \text{AT-sw-cas} (also in Figure 9.2), then apply

Figure 9.5: An IRC11 CAS rule with the atomic points-to assertion.
where $\Phi_{\text{cmp}}(\ell, h)$ is defined as in Figure 9.5.

**Figure 9.6.** An ATCAS rule with the atomic points-to in single-writer mode

**AT-CAS-SN-GEN**

$$\text{rlx} \subseteq o_f, o_r, o_w \quad \forall o, t_0 \geq \max(\text{dom}(h_0)), h(t_0) = (u_0, \_ , t) \Rightarrow \exists o' \equiv \gamma o_r$$

$$\forall V_0 * \ell \sqsupseteq h_0 * \forall V_c(\ell \sqsupseteq h) * \forall V_c(\ell \Rightarrow_{sv} h) * (o = \text{rlx}) ? \Delta \pi (\exists V_{rel} : \text{True})$$

$$P_{\text{cmp}} * \Box((v_r = \ell_r) ? (P_{\text{cmp}} \Rightarrow \Phi_{\text{cmp}}(\ell_r, h)) : \text{True})$$

**CAS**

$$b, \exists t', v', V_r, V' \sqsupseteq V_0, h, h', h_0 \sqsubseteq h' \sqsubseteq h' + h_0(t') = (v', V_r) * \max(\text{dom}(h_0)) \leq t' *$$

$$P_{\text{cmp}} * \exists V' * \forall V_c(\ell \sqsupseteq h) * \forall V_c(\ell \Rightarrow_{sv} h' * h \Rightarrow_{sv} h') *$$

$$\begin{cases} 
\text{false} & * v_r \not= v' * h' = h * (a_f \sqsubseteq \text{acq}) ? V_r \subseteq V' : \nabla_\pi (\exists V_r) \\
\text{true} & * v_r = v' * \exists t \not\in \text{dom}(h), V_w \sqsubseteq V_r \sqsubseteq V_r \sqsubseteq V_0, t = t' + 1 \\
* h' = h[t \leftarrow (v_w, V_w)] * h_0(t) = (v_w, V_w) \\
* \forall V_w \sqsubseteq V' : \nabla_\pi (\exists V_w) * (o_w \sqsubseteq \text{rel}) ? V_w \subseteq V_w : V_{rel} \subseteq V_w \end{cases}$$

The result is the CAS rule AT-CAS-SW-GEN for the single-writer mode, in Figure 9.6. 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 $V_0 \sqcup V'$, 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).

**9.2 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 (§6.4), in a similar way to the model of non-atomic points-to assertion (§8.2). 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 $\gamma$ of the assertion $\ell \Rightarrow_{\gamma} h$, which uniquely identifies an atomic period for $\ell$, will store this extra ghost state.

**Definition 9.6 (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.

$$\text{atHistR} ::= \text{Auth}\{\text{Map}(\text{Time}, \text{Ag}(\text{Val} \times \text{View}))\}$$

$$\text{naWriterR} ::= \text{Opt}(\text{Ag}(\text{View}))$$

$$\text{exWriterR} ::= \text{Auth}(\text{Opt}(\text{Frac} \times \text{Ag}(\text{Time})))$$

$$\text{atomicR} ::= \text{atHistR} \times \text{exWriterR} \times \text{naWriterR}$$
The RA \texttt{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 \texttt{NAWRITER} allows storing permanently a view which is meant to be a thread’s current view \( V_{\text{na}} \) 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 \( \gamma \) will have synchronized with \( V_{\text{na}} \).

The RA \texttt{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 \texttt{ATOMICR} for the atomic points-to is just a product of the 3 RAs above.

**Definition 9.7** (Ghost Ownership Abstraction for Atomic RAs). We define the following abstractions for ghost ownership of the atomic RAs in \( \nu\text{Prop} \). All are timeless and objective.

\[
\begin{align*}
\text{atLastNA}^\gamma(V_{\text{na}}) & := \{ (\varepsilon, \varepsilon, \text{ag}(V_{\text{na}})) \}^\gamma \\
\text{atExclTime}^\gamma(t_x) & := \{ \circ \text{ag}(t_x) \}^\gamma \\
\text{atReader}^\gamma(h) & := \{ \circ h, \varepsilon, \varepsilon \}^\gamma \\
\text{atWriter}^\gamma(h) & := \{ \bullet \frac{3}{4} h \circ h, \varepsilon, \varepsilon \}^\gamma \\
\text{atAuth}^\gamma(h, t_x, V_{\text{na}}) & := \{ \bullet \frac{3}{4} h \circ h, \bullet \{ \text{ag}(t_x), \text{ag}(V_{\text{na}}) \} \}^\gamma \\
\end{align*}
\]

- \text{atLastNA}^\gamma(V_{\text{na}}) \text{ is persistent, and records the view at the point of the switch from a non-atomic points-to to the atomic period identified by } \gamma.
- \text{atExclTime}^\gamma(t_x) \text{ is fractional and records the timestamp } t_x \text{ of the latest exclusive single-writer write in the atomic period identified by } \gamma. \text{ The full fraction } \text{atExclTime}^\gamma(t_x), \text{ which a single-writer would own, is the exclusive write permission needed to update } t_x.
- \text{atReader}^\gamma(h) \text{ is persistent and records a snapshot history } h, \text{ i.e., } h \text{ in the atomic period of } \gamma, \text{ } h \text{ is a lower bound of the current history.}
- \text{atWriter}^\gamma(h) \text{ is exclusive and records the current history } h. \text{ It is the (ghost) writer permission needed to perform a write (a change) to the history.}
- \text{atAuth}^\gamma(h, t_x, V_{\text{na}}) \text{ is the authoritative state of the atomic points-to protocol. The other ghost ownership abstractions defined above must agree or be included in this authoritative state. We note that we use a setup where the authoritative element is also fractional (} \bullet \frac{3}{4} \text{ in } \text{atAuth}^\gamma, \text{ or } \bullet \frac{1}{4} \text{ in } \text{atAuth}^\gamma). \text{ Fractions of the authoritative element enjoy agreement, and that is how we establish agreement between } \text{atWriter}^\gamma \text{ and } \text{atAuth}^\gamma.\]
Several properties of these ghost abstractions are given in Figure 9.7.

We can now give the model of the atomic points-to assertion as well as its local ownership and observation assertions.

**Definition 9.8 (Model of Atomic Local Ownership and Observations).** We first define what it means to locally observe a history $h$ of $\ell$, and to locally synchronize with that history.

\[
\text{Local}_{\ell,n}(\ell, h) ::= \forall t, v. V^\ell. h(t) = (v, V^\ell) \Rightarrow \exists [\ell \leftarrow \{ \emptyset [w := t] \}]
\]

\[
\text{Local}_{\ell,sy}(\ell, h) ::= \forall t, v. V^\ell. h(t) = (v, V^\ell) \Rightarrow \exists V^\ell \ast \exists [\ell \leftarrow \{ \emptyset [w := t] \}]
\]

That is, the owner of $\text{Local}_{\ell,n}(\ell, h)$ should have observed the timestamps of the writes in $h$, while the owner of $\text{Local}_{\ell,sy}(\ell, h)$ should additionally have observed the message views of those writes. The model of the atomic local ownership and observations is then given within $\nu\text{Prop}$, as follows.

\[
\ell \models h =: \text{Local}_{\ell,n}(\ell, h) \ast \text{atReader}^\gamma(h) \ast \exists V_{na} \ast \text{atLastNA}^\gamma(V_{na}) \ast \exists V_{na}
\]

\[
\ell \equiv_{sa}^h h =: \text{Local}_{\ell,sy}(\ell, h) \ast \text{atReader}^\gamma(h) \ast \exists V_{na} \ast \text{atLastNA}^\gamma(V_{na}) \ast \exists V_{na}
\]

\[
\ell \equiv_{sa}^h h =: \ell \equiv_{sa}^h h \ast \text{atWriter}^\gamma(h) \ast \text{atExclTime}^\gamma(t_x) \ast t_x = \max(\text{dom}(h))
\]

\[
\ell \equiv_{sa}^{atExclTime\gamma(t_x),t_x} h =: \ell \equiv_{sa}^h h \ast \text{atExclTime}^\gamma(t_x) \ast t_x \in \text{dom}(h)
\]

- The seen-history observation $\ell \equiv_{sa} h$ requires that the $h$ is indeed a snapshot history of $\ell$’s atomic points-to identified by $\gamma$ (atReader$^\gamma(h)$), and that the owner has observed the writes in $h$ ($\text{Local}_{\ell,n}(\ell, h)$). The remaining part $\exists V_{na}, \text{atLastNA}^\gamma(V_{na}) \ast \exists V_{na}$ 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 $\gamma$.

- The sync-history observation $\ell \equiv_{sa}^\gamma h$ is similar to the seen-history observation, but additionally requires the synchronization with all the message views in $h$ ($\text{Local}_{\ell,sy}(\ell, h)$).

- The single-writer ownership $\ell \equiv_{sa}^\gamma h$ requires the sync-history observation $\ell \equiv_{sa}^\gamma h$, and holds the writer permission atWriter$^\gamma(h)$ for the current history $h$ as well as the exclusive writer permission
The model of the atomic points-to lift \( V \) 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 \( \ell \geq \gamma_{\text{cas}}(\ell, q) \) \( h \) only requires the seen-history observation, and its fraction \( q \) is the fraction for exclusive single-writer timestamp \( \text{atExclTime}^\gamma(t_x) \), 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 \( t_x \) (\( t_x \in \text{dom}(h) \)).

**Definition 9.9 (Model of the Atomic Points-To).** We now give the model of the atomic points-to assertions, also within \( v\text{Prop} \). It relies on a “lift-view” function \( \text{liftV}(h, V_{na}) \) that lifts all \( h \)’s message views to include the view \( V_{na} \) of the “non-atomic to atomic” switch.

\[
\text{liftV}(h, V_{na}) := \left[ \ell \leftarrow (v, V^\gamma \cup V_{na}) \mid h(t) = (v, V^\gamma) \right]
\]

\[
\ell \xrightarrow{\text{na}} h := \exists h', \alpha_w, \alpha_1, \alpha_2, V_{na}, h = \text{liftV}(h', V_{na}) \ast t_x \in \text{dom}(h)
\]

\[
\begin{align*}
\text{Local}_{\text{sw}}(\ell, h) & \ast \text{Local}_{\text{na}}^\gamma(\ell, \alpha_w) \\
\text{Local}_{\text{na}}^\gamma(\ell, \alpha_1) & \ast \text{Local}_{\text{na}}^\gamma(\ell, \alpha_2, V_{na}) \\
\text{Hist}(\ell, h') & \ast \text{Write}^\gamma(\ell, \alpha_w) \\
\text{Read}^\gamma(\ell, \alpha_1) & \ast \text{Read}^\gamma(\ell, \alpha_2) \\
\text{atAuth}^\gamma(h, t_x, V_{na}) & \\
\begin{cases} 
\theta = \text{sw} & \text{True} \\
\theta = \text{cas} & \text{atWriter}^\gamma(h) \\
\theta = \text{con} & \text{atWriter}^\gamma(h) \ast \text{atExclTime}^\gamma(t_x)
\end{cases}
\end{align*}
\]

The model of the atomic points-to \( \ell \xrightarrow{\text{na}} h \) 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 \( \text{Hist}(\ell, h') \) of the unlifted history \( h' \) \( (h = \text{liftV}(h', V_{na})) \) and of (2) the difference parts of the race detector state (the atomic writes set \( \text{Write}^\gamma(\ell, \alpha_w) \), and the non-atomic and atomic reads sets \( \text{Read}^\gamma(\ell, \alpha_1) \) and \( \text{Read}^\gamma(\ell, \alpha_2) \)).\(^9\) 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., \( \text{Local}_{\text{na}}^\gamma(\ell, \alpha_1, V_{na}) \).

The main difference with the non-atomic points-to is the ghost ownership. The atomic points-to owns the authoritative state \( \text{atAuth}^\gamma(h, t_x, V_{na}) \) to hold the authority over the local ownership and observations (given model in **Definition 9.8**). It further owns the remaining ghost permissions for different atomic modes: (i) for the single-writer mode \( \text{sw} \) it owns nothing more, because all permissions are owned by the single-writer ownership; (ii) for the CAS-only mode \( \text{cas} \) it owns the writer permission \( \text{atWriter}^\gamma(h) \) 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 \( \text{con} \) it owns all the ghost permissions, as the clients of the \( \text{con} \) mode only have the seen-history or sync-history observations.
9.2.1 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 (§7.5). Then we unfold the model of non-atomic points-to (Definition 8.3). 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 (§7.5). Our goal then looks as follows.

\[
\begin{align*}
\text{Context:} & \quad \text{Goal:} \\
\exists V_0 * @V_0 P * @V_0 \forall V & \\
@V_0 (\text{Local}_h(\ell, [t \leftarrow (v, V)]) * \text{Local}_{\gamma}^{rlx}(\ell, \alpha_w)) & \\
@V_0 (\text{Local}_{\gamma}^{rlx}(\ell, \alpha_2) * \text{Local}_a(\ell, \alpha_1, V_{na})) & \\
\text{Hist}(\ell, [t \leftarrow (v, V)]) * \text{Write}^{rlx}(\ell, \alpha_w) & \\
\text{Read}^a(\ell, \alpha_1) * \text{Read}^{rlx}(\ell, \alpha_2) & \\
\exists_{\gamma}, t, V, \exists V * @V (P * \ell \ni \exists_{\gamma} [t \leftarrow (v, V)] * \ell \ni \vee_{\gamma} [t \leftarrow (v, V)]) & \\
\end{align*}
\]

From \( @V_0 \text{Local}_h(\ell, \alpha_1, V_{na}) \) and the vProp definition of \( \text{Local}_h^a \) (Definition 8.1), we have \( V_{na} \subseteq V_0 \).

We then allocate a new ghost location \( \gamma \) for the RA ATOMICR, using GHOST-ALLOC (§5.2), with the initial history \( h = [t \leftarrow (v, V \cup V_0)] \), the exclusive write timestamp \( t \), and the new non-atomic view \( V_0 \). This allocation will give us these following extra ownership atAuth\(^\gamma(h, t, V_0) * \) atLastNA\(^\gamma(V_0) * \) atExclTime\(^\gamma(t) * \) atWriter\(^\gamma(h) \).

We note that from \( @V_0 \ni V \), by VA-VS (§7.5), we have \( V \subseteq V_0 \). Consequently, \( V \cup V_0 = V_0 \). We then instantiate the existential quantifiers respectively with \( \gamma \), \( t \), and \( V_0 \). Note how \( V_0 \), the view of the switch, indeed becomes the new message view for \( t \). We can easily discharge \( \exists V \) and \( @V_0 P \), and then arrive at the following goal.

\[
\begin{align*}
\text{Context:} & \quad \text{Goal:} \\
\exists V_0 * @V_0 \forall V & \\
@V_0 (\text{Local}_h(\ell, [t \leftarrow (v, V)]) * \text{Local}_{\gamma}^{rlx}(\ell, \alpha_w)) & \\
@V_0 (\text{Local}_{\gamma}^{rlx}(\ell, \alpha_2) * \text{Local}_a(\ell, \alpha_1, V_{na})) & \\
\text{Hist}(\ell, [t \leftarrow (v, V)]) * \text{Write}^{rlx}(\ell, \alpha_w) & \\
\text{Read}^a(\ell, \alpha_1) * \text{Read}^{rlx}(\ell, \alpha_2) & \\
\text{atAuth}^\gamma(h, t, V_0) * \text{atLastNA}^\gamma(V_0) & \\
\text{atExclTime}^\gamma(t) * \text{atWriter}^\gamma(h) & @V_0 (\ell \ni \exists_{\gamma} h * \ell \ni \vee_{\gamma} h) \\
\end{align*}
\]

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:

\[
\begin{align*}
\text{Context:} & \quad V \subseteq V_0 \ast V_{na} \subseteq V_0 \ast \@_{V_0} \text{Local}_{R}^{na}(\ell, \alpha_1, V_{na}) \\
\text{Goal:} & \quad \@_{V_0}(\ell \sqsupseteq \gamma_{\text{sy}} \ast \text{Local}_{R}^{na}(\ell, \alpha_1, V_{0}))
\end{align*}
\]

This is easily done because \(h\) is the singleton \([t \leftarrow (v, V_0)]\), and \(\text{Local}_{R}^{na}\) is view monotone, so \(\text{Local}_{R}^{na}(\ell, \alpha_1, V_{na}) \implies \text{Local}_{R}^{na}(\ell, \alpha_1, V_{0})\) knowing that \(V_{na} \subseteq V_{0}\).

**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 6.2) to truncate \(h'\) to the singleton of its latest write. After sorting out the local observations, we are done.

9.2.2 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 \(\text{atWriter}^\gamma(h)\) 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 \(\text{atWriter}^\gamma(h)\) from the CAS-only atomic points-to out to construct the single-writer ownership. Second, we update the full fraction \(\text{atExclTime}^\gamma(t_2)\) 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.

9.2.3 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 7.5 and Definition 7.4). We then rely on the base logic rules for atomic operations (§6.3) to proceed.

**Proof sketch of AT-READ-SN.** The basis of the proof is to apply BL-HOARE-READ-AT (Figure 6.5). In the pre-condition we have \(\ell \sqsubseteq_{an} h_0 \ast \@_{V_0}(\ell \sqsubseteq \gamma_{\text{sy}} \ast h)\), and we need to satisfy the pre-condition of BL-HOARE-READ-AT. From \(\ell \sqsubseteq_{an} h_0\) we get \(\text{Local}_{\ell}(\ell, h, V.\text{cur})\) (after unfolding the models of Hoare triples and WPs, in the base logic). From \(\@_{V_0}(\ell \sqsubseteq \gamma_{\text{sy}} \ast h)\) we get (1) \(\text{Hist}(\ell, h) \ast \text{Read}_{r}^{\ell\alpha_2}(\ell, \alpha_2)\), because they are objective, and (2) \(\text{Local}_{R}^{\ell\alpha_2}(\ell, \alpha_2, V_0)\). 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 \(\text{Local}_{R}^{\ell\alpha_2}(\ell, \alpha_2 \cup \{r\}, V_0 \cup V'.\text{cur})\), which fits perfectly in AT-READ-SN’s requirement of \(\@_{V_0,V'}(\ell \sqsubseteq \gamma_{\text{sy}} \ast h)\), where it is the case that \(V' = V'.\text{cur}\).

We are then left with the observations and facts about the read message. Note that due to \(\ell \sqsubseteq_{an} h_0\), the current view \(V'.\text{cur}\) before the step
has observed all events in \( h_0 \), so by the post-condition \( V \xrightarrow{R,o,f,l,V',x} V' \) from BL-Hoare-read-at, we know that the read timestamp \( t \) is not earlier than the writes in \( h_0 \), i.e., \( t \geq \max(\text{dom}(h_0)) \). Furthermore, inspection of \( V \xrightarrow{R,o,f,l,V',x} V' \) 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 \( \text{Local}_{w}^{rlx}(\ell, \alpha_w, V_b) \) to \( \text{Local}_{w}^{rlx}(\ell, \alpha_w \cup \{t\}, V_b \cup V') \). Naturally, a careful inspection of the condition \( V \xrightarrow{g,n,f,l,\bot,V} V' \) 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 \( \forall v_0, t_0 \geq \max(\text{dom}(h_0)), h(t_0) = (v_0, \_ \_) \Rightarrow \top \)
  \( v_0 = ? v_r \) implies \( \forall v_0 \in \text{Readable}(h, V). \top \)
- show that AT-cas-sn-gen’s \( (P_{\text{cmp}} \Rightarrow \Phi_{\text{cmp}}(\ell, h)) \) implies BL-Hoare-CAS’s \( (P_{\text{cmp}} \Rightarrow \Phi_{\text{cmp}}(\ell, h)) \); 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.
Invariants in Relaxed Memory

In §5.3, 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 enqueuing 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 \(\text{mo}_l\): 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

---

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 interferences 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.

This long discussion is to motivate general invariants for multiple locations in RMC, whose variants will be presented in this chapter. In Part II, 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 7.13, §7.5), 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 §10.1.

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 §10.2.

In §10.3, 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 II).

### 10.1 Objective Invariants

**Definition 10.1 (vProp Objective Invariants).** The vProp objective invariants directly lift Iris iProp invariants as follows.

\[
\llbracket I \rrbracket^V := \lambda V. \llbracket I \rrbracket^V
\]

That is, the invariant content \(I : vProp\) is put inside an Iris invariant at an arbitrary (universally quantified) view \(V\). This resembles the model of the objective modality (Definition 7.10), hence the name.
Objective invariants satisfy the rules in Figure 10.1. That is, the rules are similar to those of Iris iProp invariants (Figure 5.4), 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 (§7.4).

Intuitively, if a client only stores objective resources, which include pure facts (φ) and ghost state (aγ), 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 7.13). In particular, we can apply VA-INTRO (Figure 7.3) to turn an atomic points-to ℓ \rightarrow h can be turn into the form @V (ℓ \rightarrow h) (knowing that \exists V) which is objective (see Figure 7.3) and thus can be put inside objective invariants using OINV-ALLOC. Fortunately, we have established in Sections 9.1.1 to 9.1.3 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_b', respectively.

In other words, the atomic access rules using atomic points-to (§9.1) 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 11 and in Part III.

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 §9.1.2, which uses the rules AT-WRITE-SW-REL and AT-READ-SN-ACQ. Nevertheless, we will also see examples for resource transfer in Chapter 11.
10.2 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 $\nu$Prop, but we need some extra work to make the returned resources usable: 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.

10.2.1 The Interface of $\nu$Prop Cancelable Invariants

We present the interface of $\nu$Prop cancelable invariants in Figure 10.2.\(^4\) Note that we did not present Iris $i$Prop cancelable invariants, but they only differ from the $\nu$Prop ones in the parts concerning views.\(^5\)

The interface of cancelable invariants involves two kinds of assertions:

1. a persistent, objective assertion $\nufont{I}_{\gamma,N}$ that an invariant with content $I$ exists in the namespace $\mathcal{N}$ with an identifier $\gamma$; and
2. a fractional and timeless invariant token assertion $\nufont{♥}_{\nufont{γ}_1}$ (also identified by $\gamma$) that is needed to know that the invariant is not yet canceled.

**Invariant Allocation.** With the invariant content $I$, we can allocate a cancelable invariant using $\text{CINV-ALLOC}$. Afterwards, we know that the invariant exists ($\nufont{I}_{\gamma,N}$), and we own the full fraction $\nufont{♥}_1$ of its invariant token. The invariant token is fractional, as shown in $\text{CINV-TOK-VALID}$ and $\text{CINV-TOK-FRAC}$, so that we can split the full fraction into pieces and give them to multiple threads and they can access the invariant concurrently. $\text{CINV-TOK-OBJ-SPLIT}$ allows us to split a fraction of a token identified by $\gamma$ 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 $\gamma$ as the token).

**Invariant Access.** A fraction of the invariant token is needed to open the invariant with the same identifier $\gamma$, as can be seen in $\text{CINV-ACC}$. 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 §5.7 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 $\text{WP-INV}$ and $\text{HOARE-INV}$ for opening traditional Iris invariants around an atomic step are derived from $\text{WP-ATOMIC}$ and $\text{INV-ACC}$. We apply the same method.

\(^4\)These are called raw cancelable invariants in the RustBelt Relaxed paper [Dan+20].

\(^5\)Please see Figure 12.1 for a few key rules of Iris traditional cancelable invariants. See also [Iri22, §10.1].
I

[72x343]N), the client can update the current view from

I

at an

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

I

that this requirement is only enforced on non-objective resources in

6

Note that if the client returns I at a

smaller than V ⊔ V′, then it can always return I at V ⊔ V′, thanks to view

monotonicity of I.
Fortunately, the atomic access rules using atomic points-to (in §9.1.1 to §9.1.3) are also designed to be compatible with this requirement: if we provides an atomic points-to \( @V_i \ell \downarrow h \) to the pre-condition of one of the rules, we will receive some \( @V_i@V_{i'} \ell \downarrow h' \) in the post-condition, which is indeed \( @V_i@V_{i'} \ell \downarrow h \) because \( V \subseteq V' \). We will see this point more concretely also in Chapter 11.

Note that we can switch between the view-join and view-at modalities easily using \( \text{VJ-VA-ACC} \). In fact, with \( \text{VJ-VA-ACC} \) and the rules in §9.1.1 to §9.1.3, we can derive atomic access rules that take an atomic points-to in the form \( \downarrow V_i \ell \downarrow h \) in the pre-condition, and return an updated atomic points-to in the form \( \downarrow V_i \ell \downarrow h' \) in the post-condition. We will see the derivations of those rules also in §11.2 (see Figure 11.5).

**Invariant Cancellation.** The second requirement by the access rule \( \text{CInv-acc} \) is what guarantees the soundness of the cancelation rule \( \text{CInv-cancel} \). Cancelation needs to maintain the following safety guarantee.

**Property 10.2 (Cancelation Safety).**

An invariant’s cancellation must happen-after all accesses to it.

\[\text{(Cancel-Safe)}\]

Nevertheless, \( \text{CInv-cancel} \) simply says that with we can trade in the full fraction \( \downarrow I \) of the invariant token for the invariant \( I \) 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 \( \text{CInv-acc} \), the core interface of iRC11 cancelable invariants (\( \text{CInv-alloc}, \text{CInv-acc}, \) and \( \text{CInv-cancel} \)) 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 \( \text{Cancel-Safe} \) (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 \( \text{CInv-acc} \), is rather hard to explain intuitively, without looking into the model of invariant tokens. We therefore delay this explanation until §10.2.2.

**Stronger Allocation Rules.** Figure 10.3 provides several stronger rules for cancelable allocation and access. \( \text{CInv-alloc-open} \) strengthens \( \text{CInv-alloc} \) by not requiring the invariant content \( I \) upfront. Instead the client is first given a fresh identifier \( \gamma \) for the invariant token, and so the client can pick the invariant content \( I \) that may depend on \( \gamma \). The client then receives the the invariant assertion \( I \) but the invariant does not hold yet (hence the mask does not include \( N \)). Once the client provides \( \triangleright I \), the invariant is established and the client receives the full invariant token \( \downarrow I \).

\( \text{CInv-alloc-frac} \) strengthens \( \text{CInv-alloc} \) in a slightly different way. The client first receives some fraction \( q \) of the invariant token with a fresh identifier \( \gamma \), the the client can pick and provide the invariant content \( I \) that may contain \( \downarrow I \) itself. After the invariant is established the client receives the remaining fraction \( \downarrow I_{q'} \), i.e., \( q + q' = 1 \). Note that \( q \) and \( q' \) are picked by the client.

\(^7\)see Figure 12.1 and [Iri22, §10.1]
A Stronger Access Rule. \(\text{CInv-acc-gen} \) (Figure 10.3) generalizes the access rule \(\text{CInv-acc} \) in several non-trivial ways. First, one does not need to trade in the invariant token \(\gamma_q \) to access the invariant: one does need to provide it, but then receives it back immediately together with the invariant content \(I_{V_i \triangleright I} \). Second, there are 3 options to close the invariant.

The first option is similar to the closing viewshift of \(\text{CInv-acc} \): to close the invariant, the client gives back the invariant content \(I_{V_i \triangleright I} \). 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 \(\gamma_q \) at some view \(V' \) of its choice, and (2) a wand viewshift \(\gamma_q \), then produces the invariant content \(I_{V_i \triangleright I} \) and some remaining resource \(P \). The invariant then consumes \(I_{V_i \triangleright I} \) to re-establish and close the invariant, and returns the remaining resource \(P \) to the client.

The last closing option embeds the cancelation rule \(\text{CInv-cancel} \): instead of returning the invariant content \(I \), the client can trade the full fraction \(\gamma_q \) of the invariant token for (1) a seen-view observation \(\forall V_i \) 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 \(E_N \triangleright \gamma_q \text{ True} \) without additional requirement to close the invariant. We note that \(\text{CInv-cancel} \) is derivable from \(\text{CInv-acc-gen} \).

Finally, we note that the stronger rules (\(\text{CInv.alloc-open}, \text{CInv.alloc-frac}, \) and \(\text{CInv-acc-gen} \)) without the view-join modality are all sound in traditional Iris SC logics. In those logics, the second closing option of \(\text{CInv-acc-gen} \) also coincides with the rule’s first option.

10.2.2 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
Definition 10.3 (RA for iRC11 Cancelable Invariants). We need the fractional view-lattice RA $\text{FracViewR} = \text{Auth}(\text{Option}(\text{Frac} \times \text{Lat}(\text{View})))$.

We define notations for two kinds of elements of the RA.

\[
\begin{align*}
\text{PartialV}_q(V_p) &::= \circ (\text{Some}(q, V_p)) \\
\text{FullV}(V_f) &::= \bullet (\text{Some}(1, V_f))
\end{align*}
\]

Definition 10.4 (Model of iRC11 Cancelable Invariants). The model of invariant tokens and cancelable invariants is given directly in $\nu\text{Prop}$, using the RA and objective invariants, as follows.

\[
\begin{align*}
\Box_q^\gamma &::= \exists V_{\text{tok}}. \left[ \left( \text{PartialV}_q(V_{\text{tok}}) \right) \cdot \text{FracViewR} \right]^\gamma * \exists V_{\text{tok}} \\
I_{\gamma, N} &::= \exists V_i. \left[ \left( \text{PartialV}_1(V_i) \right) \vee \left( \text{FullV}(V_i) \right) \right]^\gamma I
\end{align*}
\]

INVARIANT TOKENS. First of all, invariant tokens $\Box_q^\gamma$ are view-dependent assertions: even though owning a token $\Box_q^\gamma$ means owning only the ghost element $\text{PartialV}_q(V_{\text{tok}})$, this ghost ownership is tied to the current implicit view $V$ at which the assertion is interpreted through the token view $V_{\text{tok}}$. In particular, the ghost element $\text{PartialV}_q(V_{\text{tok}})$ records both the fraction $q$, which represents how much of the invariant this token owns, and the token view $V_{\text{tok}}$, 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 $\Box_q^\gamma$ has observed: $\exists V_{\text{tok}}$ (the seen-view observation, see Definition 7.12).

INVARIANT ASSERTIONS. The model of invariant assertions $I_{\gamma, N}$ simply encodes the two possible states of the invariant: “active” or “canceled”. Thus it is an objective invariant (§10.1) of a disjunction (see $\text{CInv-Model-Tok}$). 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 $\text{FullV}(V_i)$ that records the view $V_i$ in the ghost location $\gamma$. The left-hand side of the disjunction encodes the canceled state, which asserts ownership of the full fractional element $\text{PartialV}_1(V_i)$. Recall that the invariant assertion $I_{\gamma, N}$ itself is objective. The underlying invariant’s content is also objective: it is wrapped under a view-at modality of the content view $V_i$. The relation between the content view $V_i$ and the token views $V_{\text{tok}}$’s is managed entirely by the ghost elements $\text{FullV}(V_i)$ and $\text{PartialV}_q(V_{\text{tok}})$.

Concept 10.5 (Synchronized Ghost State). In essence, by its model in $\text{CInv-Model-Tok}$, cancelable invariant tokens are just ghost state. However, unlike the vanilla ghost state ownership which is objective, invariant tokens are not objective as they are tied to their owner’s observations. We generally call these “synchronized ghost state”. The RA

---

9 see Figure 10.1

10 see Figure 7.3

11 see Ghost-Obj, §7.4
FRACVIEWR has two interesting kinds of elements that help us implement the idea of "synchronicity": (1) the unique element FullV(V_f) that is used to record the full view V_f, and (2) fractional elements 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 PartialV_q(V_p)'s) is always equal to the full view V_f in FullV(V_f).

(Sync-Ghost)

This property guarantees that the partial view V_p of the full fractional element PartialV_1(V_p) is actually equal to the full view V_f of 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 cancellable 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 FullV(V_i) (see CInv-Model). The token view V_tok in the ghost element PartialV_q(V_tok) of some token ⊘^q tracks the changes to I made by each access that ⊘^q participated in. By Sync-Ghost, the full token view V_tok (1) of the full token ⊘^1 will thus be equal to the content view V_c. Consequently a thread owning ⊘^q must have observed all changes to the invariant content I, i.e., it must have □ V_c. Effectively, Cancel-Safe is maintained and CInv-cancel can safely eliminate the view-at modality protecting I (using VA-elim, §7.5) and return □ I at the canceling thread's current view.

Formal Properties of FRACVIEWR. To maintain Sync-Ghost, the RA FRACVIEWR admits the rules in Figure 10.4. CInv-Model-Sync says that any token view V_tok is included in the content view V_c, and the full token view V_tok (1) of ⊘^1 is exactly V_c. 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_tok by extending it with some V' only if we simultaneously update the content view V_c in the same way. This makes sure that every change in the full view V_c is accounted for by some token view V_tok, and thus Sync-Ghost is maintained.

Formally, CInv-Model-Sync comes from validity of FRACVIEWR. If we own both PartialV_1(V_p) and FullV(V_f), by Ghost-op and Ghost-valid (§5.2), we have valid(FullV(V_f) · PartialV_1(V_p)) = valid(•(Some(1, V_f)) · (Some(1, V_p))). By Auth-both-valid (§5.9), we have that Some(1, V_p) ≤ Some(1, V_f). By the definition of RA inclusion (RA-incl, §5.2), it must be the case that Some(1, V_p) = Some(1, V_f), i.e., V_p = V_f.
PROOF SKETCHES. To understand how the model works, we briefly present the proofs of $\textbf{CINV} - \text{CANCEL}$ and $\textbf{CINV} - \text{ACC}$.

Proof sketch of $\textbf{CINV} - \text{CANCEL}$. We prove the rule in vProp. After unfolding the model (Definition 10.4), we have the following goal.

\[
\text{Context: } \text{Goal:} \\
\begin{array}{c}
\{\text{PartialV}_1(V)\}^{I} * \exists V_{\text{tok}} \\
\exists V_i. \forall V_i, (\{\text{PartialV}_1(V_{\text{tok}})\}^{I} * \text{FullV}(V_i))^{I} \\
\end{array}
\Rightarrow \exists I > I
\]

We then open the underlying objective invariant using $\textbf{OINV} - \text{OBJ}$ 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 $\text{PartialV}_1(\_)$ and $\textbf{CINV} - \text{MODEL} - \text{SYNC}$ 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 $\textbf{CINV} - \text{MODEL} - \text{SYNC}$ we know that $V_{\text{tok}} = V_i$, so by owning $\exists V_{\text{tok}}$, the thread must have observed all changes to the invariant content: $\exists V_i$. With that, we now can take the content $\ominus V_i, V_i$ out of the invariant and eliminate the view-at modality with $\text{VA-ELIM}$, and return $\triangleright I$ for the user. To finish the proof, we put $\{\text{PartialV}_1(V_{\text{tok}})\}^{I}$ in to switch the underlying invariant to the cancelled state and close it.

Proof sketch of $\textbf{CINV} - \text{ACC}$. We also prove the rule in vProp. As in cancellation, we unfold the model, then open the underlying invariant with $\textbf{OINV} - \text{OBJ}$ and deduce that it must be in the active state. Our goal then looks as follows.

\[
\text{Context: } \text{Goal:} \\
\begin{array}{c}
(\triangleright \exists V_i, \ldots) \land \land \land \exists I > I \\
\{\text{PartialV}_1(V_{\text{tok}})\}^{I} * \exists V_{\text{tok}} \\
\text{FullV}(V_i)^{I} * \ominus V_i, \triangleright I \\
\end{array}
\Rightarrow \exists V_i, \exists V_i, \triangleright I + (\exists V_i, \triangleright I)^{I} \land \land \land \exists I > I
\]

We instantiate the existential quantification with $V_i$, and then use $\text{VA-TO-VJ}$ (Figure 7.3) 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:

\[
\begin{align*}
&\Delta \exists V_i. \ldots \equiv \neg \exists E \neg E \text{ True} \\
&\{\text{Partial}V_{\text{tok}}(V_i)\} \ast \exists V_{\text{tok}} \\
&\{\text{Full}V_i(V_i)\} \ast \Delta V_i \triangleright I \\
\text{Goal:} &\{\text{Partial}V_{\text{tok}}(V_i)\} \ast \exists V_{\text{tok}} \\
\end{align*}
\]

We now use \text{VJ-ELIM-VA} (also Figure 7.3) to turn \(\Delta V_i \triangleright I\) and \(\neg \exists V_{\text{tok}}\) into \(\Delta V_i \triangleright I\) and \(\neg \exists V_{\text{tok}}\), respectively. We then use \text{CI-NV-MODEL-UPDATE} to update \(\{\text{Full}V_i(V_i)\} \ast \{\text{Partial}V(V_{\text{tok}})\} \ast \exists V_{\text{tok}}\) to \(\{\text{Full}V_i(V_i) \cup V'\} \ast \{\text{Partial}V(V_{\text{tok}} \cup V')\} \ast \exists V_{\text{tok}}\).

We then use the closing viewshift \((\triangleright \exists V_i. \ldots) \equiv \neg \exists E \neg E \text{ True}\) with the view \(\Delta V_i \cup V'\) and the resources \(\{\text{Full}V_i(V_i) \cup V'\} \ast \{\text{Partial}V(V_{\text{tok}} \cup V')\} \ast \exists V_{\text{tok}}\) to re-establish and close the invariant. We are left with the goal for the invariant token.

Context: Goal:

\[
\{\text{Partial}V_{\text{tok}}(V_{\text{tok}})\} \ast \exists V' \equiv V' \equiv V' \equiv V' \equiv V'
\]

From \(V' \equiv V_{\text{tok}}\), we know that \(V_{\text{tok}} \cup V' = V'\), so this is easily done.

10.3 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.\(^\text{12}\) We will thus also need non-atomic invariants for our RustBelt Relaxed work (Part II).

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. Its model employs an RA similar to that of cancelable invariants, in order to carefully track the view of the invariant content. 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.\(^\text{13}\)

Nevertheless, we present the interface of iRC11 non-atomic invariants below, which is exactly the same as that of Iris.\(^\text{14}\) The interface is given in Figure 10.5.

Non-atomic invariants also have two kinds of assertions: (1) a persistent, objective assertion \(\text{NaInv}^{\delta ; N}(I)\) that a non-atomic invariant with content \(I\) exists in the namespace \(N\) with of invariant pool \(\delta\); and (2) a timeless invariant token assertion \(\text{Na \, : \, p \, \varepsilon}\) that is needed to access the invariants in the set \(\varepsilon\) 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/gpfs1/-/blob/master/theories/logic/na_invariants.v

\(^{14}\) Iris Team (The), The Iris 3.6 Technical Appendix [Iri22], §10.3.
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 \([Na : p.E]\), 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 \(\text{NAInv-new-pool}\), where we obtain the invariant token \([Na : p.\top]\) for \(p\) with the full set of invariant names \((p.\top)\). The token can be split and joined using \(\text{NAInv-tok-split}\), which supports accessing disjoint sets of invariants. With some invariant content \(I\), we can allocate a non-atomic invariant in some namespace \(N\) of the pool \(p\) using \(\text{NAInv-alloc}\).

Finally, the most important rule is the access rule \(\text{NAInv-acc}\): with a token \([Na : p.E']\) for some mask \(p.E'\) that includes \(p.N\), we can open the invariant \(\text{NaInv}^{p,N}(I)\) to access the content \(\triangleright I\). During the access, we lose the ownership of the invariant names in \(p.N\), so we only have the remaining token \([Na : p.E' \setminus N]\) 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 \([Na : p.E']\). We note that the access is non-atomic because we are not forced to have mask-changing fancy updates/viewshifts: through out the access the usual atomic invariants in \(E\) still hold. Recall that the masks in fancy updates ((\(\triangleright E\) ) are used to maintain non-reentrancy for atomic invariants, while the masks in non-atomic invariant tokens ((\([Na : p.E']\)) 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 11, and more of them in the rest of this dissertation. We will see the application of (3) in Part II.
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 §11.1 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 (§8) and atomic points-to (§9), objective invariants (§10.1), view-explicit modalities (§7.5) and fence modalities (§7.3). In §11.2 we verify the message-passing but with resource reclamation (deallocation), demonstrating cancelable invariants (§10.2) and the switching from atomic back to non-atomic points-to (§9.5). In §11.3, 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. Finally, in §11.4, we verify a release-acquire implementation of the linked-list based Treiber stack against a simple “bag” specification, which demonstrates the atomic points-to CAS rule with pointer comparison. We will revisit the Treiber stack with a stronger specification in Compass (Part III).

11.1 Release-Acquire Message-Passing

In Figure 11.1b, we provide two λRust + ORC11 implementations of the message passing example, together with the desired specification. Figure 11.1a presents the implementation using release-acquire accesses, and corresponds to Example 2.1c. 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 ℓ + 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 ℓ + 1. The program mp_acq_fence (Figure 11.1b) 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 ::= 
\[\begin{align*}
\pi_1: & \quad \text{let } \ell := \text{alloc}(2) \text{ in} \\
\pi_2: & \quad \ell := \text{na} 0; (\ell + 1) := \text{na} 0; \\
\pi_3: & \quad \text{fork } \{ \rho_1: (\ell + 1) := \text{na} 42; \rho_2: \ell := \text{rel} 1 \} \\
\pi_4: & \quad \text{repeat } (^\text{acq} \ell ! = 0); \\
\pi_5: & \quad \text{na}(\ell + 1) // 42
\end{align*}\]

mp\_acq\_fence ::= 
\[\begin{align*}
\pi_1: & \quad \text{let } \ell := \text{alloc}(2) \text{ in} \\
\pi_2: & \quad \ell := \text{na} 0; (\ell + 1) := \text{na} 0; \\
\pi_3: & \quad \text{fork } \{ \rho_1: (\ell + 1) := \text{na} 42; \rho_2: \ell := \text{rel} 1 \} \\
\pi_4: & \quad \text{repeat } (^\text{rel} \ell ! = 0); \\
\pi_5: & \quad \text{fence\_acq}; \\
\pi_6: & \quad \text{na}(\ell + 1) // 42
\end{align*}\]

(a) MP with release-acquire accesses.

(b) MP with an acquire fence.

\[\text{repeat } (e) ::= (\text{rec } f([[]]) := \text{let } v := e \text{ in} \]

\[\begin{array}{ll}
\text{if } v == \text{false then } f([[])] \text{else } v([[])]
\end{array}\]

\text{MP\_SPEC } 
\{\text{True} \} \text{ mp in } \pi \{ v, v = 42 \}\)

\text{MP\_SPEC\_STRONG } 
\{\text{True} \} \text{ mp in } \pi \{ v, v = 42 \} \Rightarrow [1, 42] \Rightarrow [2, 42] \}

\text{Figure 11.1: Message-Passing with Loops}

\[\text{of the allocated block is returned, to prevent memory leaks. Note that}
\text{the notation } \ell \mapsto [1, 42] \text{ stands for } \ell \mapsto 1 \ast \ell + 1 \mapsto 42. \text{ We will look at a}
\text{similar proof to that of } \text{MP\_SPEC\_STRONG in } \S 11.2.
\]

\text{A high-level proof sketch of } \text{mp.} \text{ We start in line } \pi_1 \text{ using } \text{NA\_ALLOC (} \S 8.1\text{),}
\text{from which we get the block ownership } \gamma^2 \ell \text{ and two non-atomic points-to assertions for } \ell \text{ and } \ell + 1. \text{ The two points-to are}
\text{sufficient for initialization in line } \pi_2, \text{ using } \text{NA\_WRITE.} \text{ We then use } \text{NA\_AT\_SW (} \S 9.1\text{) to turn the}
\text{non-atomic points-to } \ell \mapsto 0 \text{ of } \ell \text{ to an atomic one with a single-writer permission (} \ell \gamma^{\text{ns}}_{sv} \ast \ell \gamma^{\text{sv}}_{\pi} \text{).} \text{ We then use } \text{OINV\_ALLOC\_OBJ (} \S 10.1\text{) to allocate an}
\text{objective invariant that contains that atomic points-to } \ell \gamma^{\text{ns}}_{sv} \ast \ell \gamma^{\text{sv}}_{\pi}. \text{ In line } \pi_3 \text{ when forking } \rho, \text{ we give the non-atomic points-to}
\text{of } \ell + 1 \mapsto 0 \text{ of } \ell + 1 \text{ and the single-writer ownership } \ell \gamma^{\text{sv}}_{\pi} \text{ of } \ell \text{ to } \rho, \text{ as}
\text{well as the fact that the invariant has been established.}
\]

\text{In thread } \rho: \text{ in line } \rho_1, \text{ with the non-atomic points-to of } \ell + 1, \text{ we can write the message 42 using } \text{NA\_WRITE,}
\text{and then get } \ell + 1 \mapsto 42. \text{ In line } \rho_2, \text{ we can open the invariant with } \text{OINV\_ACC} \text{ to access the atomic}
\text{points-to } \ell \gamma^{\text{ts}}_{sv} \ast \ell \text{ so that we can perform the release write of 1 to it, using}
\text{AT\_WRITE\_SW\_REL (} \S 9.1.2\text{) and the single-writer ownership } \ell \gamma^{\text{sv}}_{\pi} \text{ of } \ell. \text{ When closing the invariant we return to the}
\text{invariant not only } \ell \text{'s atomic points-to but also } \ell + 1 \mapsto 42 \text{ at the view of the release write, so that }
\text{we can regain it.}
\]

\text{Back to thread } \pi: \text{ in line } \pi_4, \text{ in the repeat loop we can open the invariant and access } \ell \text{'s atomic points-to and keep reading } \ell, \text{ using}
\text{AT\_READ\_SN\_ACQ (} \S 9.1.1\text{). One we read that } \ell \text{ is non-zero and the loop ends,}
\text{we know that } \ell + 1 \mapsto 42 \text{ must be inside the invariant, and thread } \pi \text{ has observed the view of } \ell + 1 \mapsto 42. \text{ We need a unique token } \diamond \text{ to say that } \pi
\text{ is the only one who can acquire } \ell + 1 \mapsto 42. \text{ This must be prepared in}
\text{the beginning before allocating the invariant, and the token } \diamond \text{ is given to
To write out the proof more formally, we need to define the exclusive ghost token \( \diamond \) and the objective invariant for \( \text{mp} \).

**Definition 11.1 (Exclusive Tokens).** We use the exclusive RA of unit \( \text{EX}(1) \) to define exclusive tokens.

\[
\diamond^\gamma := \{ \text{EX}(1) \}
\]

Exclusive tokens satisfy the following rules.

<table>
<thead>
<tr>
<th>timeless(( \diamond^\gamma ))</th>
<th>EXCL-TOK-ALLOC</th>
<th>EXCL-TOK-EXCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \vdash \exists \gamma. \diamond^\gamma )</td>
<td>( \diamond^\gamma ) ( \cdot ) ( \diamond^\gamma ) ( \vdash ) False</td>
<td></td>
</tr>
</tbody>
</table>

**Definition 11.2 (Invariant for \( \text{mp} \)).** The invariant of \( \text{mp} \) needs to be objective, and contains the atomic points-to ownership of \( \ell \) in single-writer mode, since only the thread \( \rho \) is writing.

\[
\text{mpl}(\ell, \gamma_\ell, \gamma) : \text{vProp} := \\
\exists h, t_0, V_0, V_1. \text{let } h_0 := [t_0 \leftarrow (0, V_0)] \text{ in} \\
\text{if } b = \text{false} \text{ then } h = h_0 \text{ else} \\
\exists t_1 > t_0. V_1. h = [t_0 \leftarrow (0, V_0)][t_1 \leftarrow (1, V_1)] \cdot (\diamond^\gamma \lor \text{let } \delta := \text{of } \ell + 1 \Rightarrow 42) \\
\]

More concretely, the invariant \( \text{mpl} \) owns \( \text{let } \delta := \text{of } \ell + 1 \Rightarrow 42 \) for some atomic period identifier \( \gamma_\ell \) and some history \( h \), at some view \( V_1 \). The history \( h \) of \( \ell \) can be in two states, dictated by the existential quantified boolean \( b \). If \( b \) is \text{false}, then \( \ell \) 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 \text{true}, \( \ell \) is in its "signaled" state, where its history \( h \) has one extra write message \( (t_1, 1, V_1) \). When \( \ell \) is in the signaled state, \( \text{mpl} \) either owns \( \diamond^\gamma \), or owns \( \ell + 1 \Rightarrow 42 \) at the view \( V_1 \) (of the signaling write). If \( \text{mpl} \) owns \( \ell + 1 \Rightarrow 42 \), it means that \( \rho \) has released the non-atomic of \( \ell + 1 \) but \( \pi \) has not acquired it yet. Once \( \pi \) has acquired \( \ell + 1 \)'s non-atomic points-to, \( \text{mpl} \) will own \( \diamond^\gamma \).

\( \text{mpl} \) 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 \( \text{mp} \).** We present the detailed proof sketch of \( \text{mp} \) using Hoare proof outlines, in Figure 11.2. Note that the post-condition of the proof is almost satisfying the stronger specification \( \text{MP-SPEC-STRONG} \): it is only missing the points-to \( \ell \Rightarrow 1 \).

**Proof sketch of \( \text{mp-acq-fence} \).** The proof of \( \text{mp} \) uses the same invariant \( \text{mpl} \), and follows the proof of \( \text{mp} \) closely. The main difference is that in thread \( \rho \)'s read of \( \ell \), which is a relaxed read instead of an acquire read, we use the rule \( \text{AT-READ-SN} \). Consequently, in the case where \( \rho \) reads 1 from \( \ell \), we will acquire \( V_{\pi} \Rightarrow V' \) \( \cdot \text{let } \ell + 1 \Rightarrow 42 \), where \( V' \subset V_1 \). Then, with the acquire fence, we apply \( \text{HOARE-ACQ-FENCE} \) (§7.3) to get \( \subseteq V' \).
\{True\}

\(\pi_1\): let \(\ell := \text{alloc}(2)\) in \(\{\ell \mapsto \mathbb{N} \ast \ell + 1 \mapsto \mathbb{N} \ast \ell^2\} \) // NA-ALLOC

\(\pi_2\): \(\ell := \text{na}_0; (\ell + 1) := \text{na}_0; \{\ell \mapsto 0 \ast \ell + 1 \mapsto 0 \ast \ell^2\} \) // NA-WRITE

\(\{\ell + 1 \mapsto 0 \ast \ell^2 \ast \mathbb{N} \ast \gamma \ast t_0; V_0, \ast \mathbb{N} \ast V_0 \ast \ell \ast \mathbb{N} \ast \gamma \ast s_0\} + 0 \ast (0, V_0) + \ell \ast \mathbb{N} \ast \gamma \ast s_0\} + 0 \ast (0, V_0)\} \) // NA-ALLOC

Context: \(V_0 \) // EXCL-TOK-ALLOC and NA-AT-SW

\(\{\ell + 1 \mapsto 0 \ast \ell^2 \ast \mathbb{N} \ast \gamma \ast t_0; V_0, \ast \mathbb{N} \ast V_0 \ast \ell \ast \mathbb{N} \ast \gamma \ast s_0\} + 0 \ast (0, V_0) + \ell \ast \mathbb{N} \ast \gamma \ast s_0\} + 0 \ast (0, V_0)\} \) // VA-INTRO

\(\{\ell + 1 \mapsto 0 \ast \ell^2 \ast \mathbb{N} \ast \gamma \ast t_0; V_0, \ast \mathbb{N} \ast V_0 \ast \ell \ast \mathbb{N} \ast \gamma \ast s_0\} + 0 \ast (0, V_0) + \ell \ast \mathbb{N} \ast \gamma \ast s_0\} + 0 \ast (0, V_0)\} \) // VA-INTRO

Context: \(\ell \ast \mathbb{N} \ast \gamma \ast t_0; V_0, \ast \mathbb{N} \ast V_0 \ast \ell \ast \mathbb{N} \ast \gamma \ast s_0\} + 0 \ast (0, V_0) + \ell \ast \mathbb{N} \ast \gamma \ast s_0\} + 0 \ast (0, V_0)\} \) // OINV-ALLOC-OBJ

\(\pi_3\): fork \(\{\ldots\} \) // HOARE-FORK

\(\rho_1\): \((\ell + 1) := \text{na}_0; (\ell + 1) \mapsto 42\ast \ell \mapsto \mathbb{N} \ast \gamma \ast t_0; V_0, \ast \mathbb{N} \ast V_0 \ast \ell \mapsto \mathbb{N} \ast \gamma \ast s_0\} + 0 \ast (0, V_0) + \ell \mapsto \mathbb{N} \ast \gamma \ast s_0\} + 0 \ast (0, V_0)\} \) // OINV-ACC-OBJ

\(\{\ell + 1 \mapsto 42 \ast \ell \mapsto \mathbb{N} \ast \gamma \ast t_0; V_0, \ast \mathbb{N} \ast V_0 \ast \ell \mapsto \mathbb{N} \ast \gamma \ast s_0\} + 0 \ast (0, V_0) + \ell \mapsto \mathbb{N} \ast \gamma \ast s_0\} + 0 \ast (0, V_0)\} \) // AT-SW-AGREE

// By unfolding mpl and using a stronger version of AT-SW-REL

\(\rho_2\): \(\ell := \text{rel}_1; \)

\(\{\exists t_1, V_1, \ast V_0 \ast \ell \mapsto \ell \mapsto + 42\ast \ell \mapsto \mathbb{N} \ast \gamma \ast t_0; V_0, \ast \mathbb{N} \ast \gamma \ast t_1; (1, V_1)\} \) // VA-ELIM

\(\{\ell \mapsto \mathbb{N} \ast \gamma \ast t_0; V_0, \ast \mathbb{N} \ast \gamma \ast t_1; (1, V_1)\} \) // OINV-ACC-OBJ

\(\{\mathbb{N} \ast \gamma \ast \ell \mapsto \mathbb{N} \ast \gamma \ast t_0; V_0, \ast \mathbb{N} \ast \gamma \ast t_1; (1, V_1)\} \) // VA-ELIM

\(\mathbb{N} \ast \gamma \ast \ell \mapsto \mathbb{N} \ast \gamma \ast t_0; V_0, \ast \mathbb{N} \ast \gamma \ast t_1; (1, V_1)\} \) // AT-READ-SN-ACQ

We have \(v \neq 0 \Rightarrow b = \text{true} \lor V_1 = V\). If \(v = 0\), we return the invariant content unchanged.

If \(v \neq 0 \land b = \text{true}\), by EXCL-TOK-EXCL we have \(\ast V_1 \ast (\ell + 1 \mapsto + 42)\). With \(V' \supseteq V \cup V\) and \(\supseteq \), by VS-MONO we have \(\supseteq V_1\). With VA-ELIM, we get \(\ell + 1 \mapsto + 42\). We put back \(\gamma \ast \ell \ast \mathbb{N} \ast \gamma \ast s_0\} + 0 \ast (0, V_0) + \ell \ast \mathbb{N} \ast \gamma \ast s_0\} + 0 \ast (0, V_0)\} \) // “loop invariant”, we use LOf induction before the loop

\(\{\ell + 1 \mapsto \ell^2 \ast \ell + 1 \mapsto + 42\} \) // \(\ell^2 \ell \) was framed

\(\pi_5\): \(\text{na}_0(\ell + 1) \{v, v = 42 \ast \ell \ast \ell + 1 \mapsto + 42\} \) // NA-READ

\(\{v, v = 42\} \)

Figure 11.2: Hoare proof outlines for mp
Then we use VS-MONO to get $\triangledown V_1$, 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.

\[
\begin{align*}
\{ \wedge \ell \ast \Diamond \gamma \} \\
\pi_4: \texttt{repeat}(\ast \text{acq} \ell \neq 0); \\
\text{Loop body} \{ \Diamond \gamma \} \triangledown V_1 \ast \text{rlx} \ell \// \text{OINV-ACC and AT-READ} \\
\{ \wedge \ell \ast V' \ast \text{@} \ell + 1 \mapsto 42 \} \\
\pi_5: \texttt{fence}_\text{acq}; \{ \wedge \ell \ast V' \ast \text{@} \ell + 1 \mapsto 42 \} // \text{HOARE-ACQ-FENCE} \\
\{ \wedge \ell \ast \ell + 1 \mapsto 42 \} // \text{VS-MONO then VA-ELIM} \\
\pi_6: \ast \texttt{na}(\ell + 1) \{ v.v = 42 \ast \wedge \ell \ast \ell + 1 \mapsto 42 \} // \text{NA-READ} \\
\{ v.v = 42 \}
\end{align*}
\]

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 11.2 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 11.2 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 more simple than our proof here, because the invariant mpI is indeed a single-location invariant that governs the atomic accesses of $\ell$.

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 §15 (Part II), 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 III). 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.

### 11.2 Release-Acquire Message-Passing with Reclamation

We now look at the verification of the mp_reclaim, whose implementation and specification are given in Figure 11.3. 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

\(^5\)Kaiser et al., “Strong Logic for Weak Memory: Reasoning About Release-Acquire Consistency in Iris” [Kai+17], §3.2.2, §4.1.
Example Verifications with iRC11

\[
\text{mp\_reclaim ::=}
\]

\[
\pi_1: \text{let } \ell := \text{alloc}(2) \text{ in }
\]

\[
\pi_2: \ell := \text{na} 0; (\ell + 1) := \text{na} 0;
\]

\[
\pi_3: \text{fork } \{ \rho_1: (\ell + 1) := \text{na} 42; \rho_2: \ell := \text{rel} 1; \}
\]

\[
\pi_4: \text{repeat } (\text{acq } \ell != 0);
\]

\[
\pi_5: \text{let } v := \text{na}((\ell + 1)) \text{ in }
\]

\[
\pi_6: \text{free}(\ell, 2);
\]

\[
\pi_7: v \text{ // 42}
\]

\text{Figure 11.3: Message-Passing with Reclamation}

The Hoare proof outlines for \text{mp\_reclaim} are given in Figure 11.4. We note several important points.

- Between line \(\pi_2\) and \(\pi_3\), we allocate a cancelable invariant for mpcl. But instead of using \text{CINV-ALLOC}, we use the stronger rule

\[
\text{mp\_reclaim in } \pi
\]

\[
\{ v, v = 42 \}
\]

\text{MP-CLAIM-SPEC}

\[
\{ \text{True} \}
\]

\text{Call of free in line } \pi_6 \text{ is safe. We note that verifying } \text{mp\_reclaim} \text{ against MP-CLAIM-SPEC is the same as verifying mp against MP-SPEC-STRONG.}

Recall that by the end the proof for mp (Figure 11.2), we are missing only the non-atomic points-to \(\ell \mapsto 1\) of the location \(\ell\), 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 \(\ell\) from the invariant, we need to be able to cancel the invariant once thread \(\pi\) 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 \(\odot\) and the disjunction as in mpI (see Definition 11.2). On the other hand, we now need to deal with the invariant token \(\heartsuit\) of cancelable invariants (§10.2).

\text{Definition 11.3 (Invariant for mp\_reclaim). The invariant of mp\_reclaim does not need to be objective, so it contains the atomic points-to ownership of } \ell \text{ in single-writer mode, without being under a view-at modality.}

\[
\text{mpcl}(\ell, \gamma, \gamma) : \forall \text{Prop ::=}
\]

\[
\exists h, b, t_0, V_0, \ell \overset{sw_2}{\mapsto} h \times \text{let } h_0 := [t_0 \leftarrow (0, V_0)] \text{ in }
\]

\[
\text{if } b = \text{false then } h = h_0 \text{ else }
\]

\[
\exists t_1 > t_0, V_1, h = [t_0 \leftarrow (0, V_0)][t_1 \leftarrow (1, V_1)] \times \heartsuit_{V_1}(\heartsuit_{1/2} \odot \ell + 1 \mapsto 42)
\]

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 \(\odot\) and the non-atomic points-to of \(\ell + 1\), the invariant just holds the points-to together with half of the cancelable invariant token \(\heartsuit_{1/2}\), at the message view \(V_1\).

Intuitively, at the beginning the invariant token \(\heartsuit_{1/2}\) is split into two halves, and each participant (threads \(\pi\) and \(\rho\)) will keep one half to access the invariant. Once thread \(\rho\) is done, it sends back its half \(\heartsuit_{1/2}\) together with \(\ell + 1 \mapsto 42\) (by putting them in mpcl so that thread \(\pi\) can acquire both, reconstruct the full token \(\heartsuit_{1/2}\), and cancel the invariant to get back \(\ell \overset{sw_2}{\mapsto} h\).

\text{Proof sketch for mp\_reclaim. The Hoare proof outlines for mp\_reclaim are given in Figure 11.4. We note several important points.}

- Between line \(\pi_2\) and \(\pi_3\), we allocate a cancelable invariant for mpcl. But instead of using \text{CINV-ALLOC}, we use the stronger rule
\begin{align*}
\{ \text{True} \} \\
\pi_1: & \text{ let } \ell := \text{alloc}(2) \text{ in } \\
\pi_2: & \ell := \text{na} 0; (\ell + 1) := \text{na} 0; \{ \ell \mapsto 0 * \ell + 1 \mapsto 0 * [\ell] \} \quad \text{// NA-ALLOC and NA-WRITE} \\
\{ \ell + 1 \mapsto 0 * \ell + 1 \mapsto [\ell] \} \quad \text{// NA-AT-sw} \\
\text{Context: } & \exists V_0 \neq \ell \not\in_{\text{sv}} [t_0 \leftarrow (0, V_0)] \quad \text{// AT-sw-syn and AT-syn-sn} \\
\{ \ell + 1 \mapsto 0 * \ell \not\in_{\text{sv}} [t_0 \leftarrow (0, V_0)] \} \quad \text{// CINV-ALLOC-OPEN} \\
\text{Context: } & \text{mpcl}(\ell, \gamma, \gamma) \not\in_{\text{sv}} \{ \text{True} \} \\
\pi_3: & \text{ fork } \{ \ldots \} \quad \text{// HOARE-FORK} \\
\rho_1: & \{ \ell + 1 := \text{na} 42; \{ \ell + 1 \mapsto 42 * \ell \not\in_{\text{sv}} [t_0 \leftarrow (0, V_0)] \} \not\in_{\text{sv}} \gamma / 1 \} \quad \text{// CINV-ACC-GEN} \\
\rho_2: & \ell := \text{rel} 1; \{ \exists V_1, V_2, \exists V_2 \not\in_{\text{sv}} [t_0 \leftarrow (0, V_0)] \} \quad \text{// By applying AT-write-syn-rel-vj with } \exists V_0 \\
& \text{We pick the “signaled” state for } \gamma \text{ and return the invariant content unchanged.} \\
\pi_4: & \text{ repeat } \{ \text{acq } \ell != 0; \} \\
\text{Loop } \rho & \{ \gamma / 1 \} \quad \text{// CINV-ACC-GEN} \\
& \{ \gamma / 1 \neq \exists V_1, \exists V_1 \not\in_{\text{sv}} \text{mpcl}(\ell, \alpha, \gamma) \ldots \text{(* closing viewshifts *)} \} \quad \text{// CINV-ACC-GEN} \\
& \{ \gamma / 1 \neq \exists V_0 \not\in_{\text{sv}} [t_0 \leftarrow (0, V_0)] \} \quad \text{// AT-read-syn-acc-vj} \\
& \text{With } \gamma / 1 \neq 0 \Rightarrow b = \text{true } \land V_1 = V \). If \( v = 0 \), we return the invariant content unchanged. \\
\text{If } b = \text{true}, \text{ we have } \exists V_1 (\gamma / 1 \neq \ell + 1 \mapsto 42). \text{ With VS-MONO and VA-ELIM, we get } \\
\gamma / 1 \neq \ell + 1 \mapsto 42. \text{ Combining with } \pi \text{’s own } \gamma / 1 \neq \ell + 1 \mapsto 42. \text{ By cancelation (CINV-ACC-GEN), we get } \exists V_1, \text{ which can be used with } VJ \text{-ELIM to get } \ell \mapsto 42. \text{ Then use } AT-\text{NA to get } \ell \mapsto 1. \\
& \{ v = 0 \mapsto \text{rel} \neq_{\text{sv}} [t_0 \leftarrow (0, V_0)] \} \quad \text{// CINV-ACC-GEN} \\
& \{ v = 0 \mapsto \text{rel} \neq_{\text{sv}} [t_0 \leftarrow (0, V_0)] \} \quad \text{// AT-read-syn-acc-vj} \\
& \{ v = 0 \mapsto \text{rel} \neq_{\text{sv}} [t_0 \leftarrow (0, V_0)] \} \quad \text{// NA-read} \\
\pi_5: & \text{ let } v := \text{rel} (\ell + 1) \text{ in } \{ v = 42 * \ell \mapsto [1, 42] \} \quad \text{// NA-READ} \\
\pi_6: & \text{ free}(\ell, 2); \{ v = 42 \} \quad \text{// NA-UNSYNC and NA-DEALLOC} \\
\pi_7: & v \{ v = 42 \} \\
\end{align*}
CINV-alloc-open (Figure 10.3), 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 mpcl(ℓ, γℓ, γ).

- After the allocation of mpcl(ℓ, γℓ, γ)γ, we also get the full invariant token ∃γ1, which we split into two halves of ∃γ1/2, 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 ρ2, 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 10.3). The second option allows us to use the half token ∃γ1/2

after the write, we instantiate the second closing option with

That is, after the write, we have ρ1 (∃γ1/2 * ℓ + 1 → 42) where V1 is the message view of the write.

The second closing option looks as follows.

∀V′, P. (@V′ ∃γ1/2 * (@V′ ∃γ1/2 ≡ ∃E ∀N (∪V1 ℓ mpcl(ℓ, γℓ, γ)) * P))

E ∀N ≡ E

With @V1 (∃γ1/2 * ℓ + 1 → 42) and the atomic points-to

∪V1 (ℓ ≥ 2γsv t0 ← (0, V0)) t1 ← (1, V1))

after the write, we instantiate the second closing option with V′ := V1 and P := True. We give up @V1 ∃γ1/2 for the left-hand side of the separating conjunction before the wand viewshift (∃E ∀N ≡ E). For the right-hand side, it is easy to show that

@V1 (∃γ1/2 * ℓ + 1 → 42) * ∪V1 (ℓ ≥ 2γsv t0 ← (0, V0)) t1 ← (1, V1))

γ1/2NV γ V1

γ1/2γNV γ V1

γ1/2NV γ V1

γ1/2NV γ V1

γ1/2NV γ V1

by picking the signaled state (b = true) for mpcl(ℓ, γℓ, γ). We note that @V1 (∃γ1/2 * ℓ + 1 → 42) ⊢ ∪V1 @V1 (∃γ1/2 * ℓ + 1 → 42), due to VJ-VA (Figure 7.3).

After the instantiation, we get γ1/2NV γ V1 ≡ 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 @V1 ℓ ≡ γ. We do not need this feature here, but it can be useful elsewhere (see §15).

- The same situation applies for thread π's atomic access in line π4: we need to use CINV-acc-gen to open the invariant, and in case the
invariant is in the signaled state (\(b = \text{true}\)), we need to use \(\text{VA-VA}\) to get \(\forall V_i \models \ell \vdash_{\text{sv}} h \land \forall V_i (\ell \vdash_{\text{sv}} h) \bullet P \models \ell := \text{rel} v \in \pi \) for some \(v \in \pi\).

\[
\{ \forall V_0 \models \ell \vdash_{\text{sv}} h \land \forall V_i (\ell \vdash_{\text{sv}} h) \bullet P \models \ell := \text{rel} v \in \pi \}
\]

\(\text{AT-write-sw-rel-vj}\)

\[
\{ \forall V_0 \models \ell \vdash_{\text{sv}} h \land \forall V_i (\ell \vdash_{\text{sv}} h) \bullet P \models \ell := \text{rel} v \in \pi \}
\]

\(\text{AT-read-sn-acq-vj}\)

\[
\{ \forall V_0 \models \ell \vdash_{\text{sn}} h_0 \land \forall V_i (\ell \vdash_{\text{sn}} h_0) \bullet \text{acq} \ell \in \pi \}
\]

\(\text{Figure 11.5: Derived iRC11 atomic access rules with the view-join modality}\)

• Last but not least, we note that the atomic access rules \(\text{AT-write-sw-rel}\) and \(\text{AT-read-sn-acq}\) are not directly applicable to this proof, because the rules require an atomic points-to under a view-at modality, i.e., \(\forall V_i \ell \mapsto_{\text{g}} h\), while we get a points-to under a view-join modality from the non-cancellable invariant access rule, i.e., \(\forall V_i \ell \mapsto_{\text{g}} h\). Fortunately, in general, rules with the view-join modality can be derived from those with the view-at modality. We show the derived versions \(\text{AT-write-sw-rel-vj}\) and \(\text{AT-read-sn-acq-vj}\) in Figure 11.5. We demonstrate the derivation of \(\text{AT-write-sw-rel-vj}\) below.

\[\text{Proof sketch of } \text{AT-write-sw-rel-vj}.\] With \(\forall V_0 \land \forall V_i (\ell \vdash_{\text{sv}} h)\), we use \(\text{VJ-elim-VA}\) (Figure 7.3) to get

\[\forall V' \models \forall V' \land \forall V_i (\ell \vdash_{\text{sv}} h)\]

for some \(V' \models V_0\). We apply \(\text{AT-write-sw-rel}\) with \(\forall V' \land \forall V_i (\ell \vdash_{\text{sv}} h)\). In the post-condition we get back the atomic points-to in the form

\[\forall V' \land \forall V_i (\ell \vdash_{\text{sv}} h[t \leftarrow (v, V)])\]

for some \(V \models V_0\) and \(\forall V\). We can rewrite it using \(\text{VA-VJ}\) to the form

\[\forall V' \land \forall V_i (\ell \vdash_{\text{sv}} h[t \leftarrow (v, V)])\]

We now use \(\text{VA-elim}\) to get back \(\forall V_i (\ell \vdash_{\text{sv}} h[t \leftarrow (v, V)])\). Note that we have \(\forall (V' \cup V)\) from \(\forall V'\) and \(\forall V\) using \(\text{VS-join}\). □
11.3 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 11.6.

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(ℓ, v), which writes the computation result v non-atomically to ℓ + 1, and signals the completion with a release write of 1 to ℓ. A waiting thread calls join([ℓ]), which waits with a repeat loop reading ℓ, and once the loop finishes it reads the result from ℓ + 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 ℓ + 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([ℓ, 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([ℓ, f()]) }.

The specifications of the library functions assume a predicate Φ : Val → vProp, where the computation is expected to produce not just some value v, but also the resource Φ(v). The specifications assume a namespace N that will be used to allocate the library invariant. The specifications then involve two assertions: the finish handle FinishHandleN(ℓ, Φ) and the join handle JoinHandleN(ℓ, Φ). 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

\[
\text{spawn ::= }\lambda [f]. \\
\text{finish ::= }\lambda [ℓ, v]. \\
\text{join ::= }\lambda [ℓ]. \\
\begin{align*}
1: & \text{let } ℓ := \text{alloc}(2) \text{ in} \\
2: & ℓ := \text{na } 0; \text{ // init to } 0 \\
3: & \text{fork } \{ f([ℓ]) \}; \text{ // spawn } f \\
4: & ℓ \\
\end{align*}
\]

\begin{align*}
\text{Spawn-spec} & \{ (∀ℓ, Φ. \text{FinishHandle}N(ℓ, Φ) \Rightarrow \text{wp } f([ℓ]) \text{ in } Φ \{ \text{True} \} ) \} \text{spawn([f]) in } π \{ ℓ \vdash Φ, \text{JoinHandle}N(ℓ, Φ) \} \uparrow
\text{Finish-spec} & \{ \text{FinishHandle}N(ℓ, Φ) \ast Φ(v) \} \text{finish([ℓ, v]) in } π \{ \text{True} \} \uparrow
\text{Join-spec} & \{ \text{JoinHandle}N(ℓ, Φ) \} \text{join([ℓ]) in } π \{ Φ(v) \} \uparrow
\end{align*}

Figure 11.6: A Spawn-and-Join library
consume the finish handle that \texttt{spawn} generates, and (2) returns the join handle in the post-condition. \texttt{FINISH-spec} says that once \texttt{f} produces the result \( v \) and finally calls \texttt{finish}, it consumes the finish handle and release the resource \( \Phi(v) \). \texttt{JOIN-spec} says that \texttt{join} consumes the join handle generated by \texttt{spawn} and once it is done the resulting resource \( \Phi(v) \) is returned.

Note that we could have stated the pre-condition of \texttt{SPAWN-spec} with a Hoare triple for \( f \), i.e.,

\[
\forall \ell, \rho. \{ \text{FinishHandle}^N(\ell, \Phi) \} f(\ell) \in \rho \{ \text{True} \} \]

However, stating as it is in \texttt{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 (\( □ \)), so if we use a Hoare triple the client can only use \texttt{FinishHandle}^N(\ell, \Phi) for the proof of \( f \).

**Definition 11.4** (Invariant for Spawn-and-Join). The invariant of Spawn-and-Join is almost the same as that for \texttt{mp_reclaim} (Definition 11.3), except that the message to be sent is now some value \( v \) together with the resource \( \Phi(v) \).

\[
\text{spaw Join}(\ell, γ_l, γ) : v \text{Prop} ::= \exists h, b, t_0, V_0, \ell. \wedge_{sv} h * \textbf{let} h_0 := [t_0 ← (0, V_0)] \textbf{in}
\]

\[\text{if } b = \text{false then } h = h_0 \text{ else } \exists t_1 > t_0, V_1. h = [t_0 ← (0, V_0)][t_1 ← (1, V_1)] * \Diamond_{V_1} ()^{\gamma_1/2} \ast ∃ v. \ell + 1 \mapsto v \ast \Phi(v)\]

**Definition 11.5** (Model of Handles).

\[
\text{FinishHandle}^N(\ell, \Phi) ::= \exists γ_ℓ, γ, t, V, \ell + 1 \mapsto \bigvee_{sv} [t ← (0, V)]
\]

\[\ast \Diamond_{\ell} \ast \text{spaw Join}(\ell, γ_l, γ)\]^{\gamma}\text{N}

\[
\text{JoinHandle}^N(\ell, \Phi) ::= \exists γ_ℓ, γ, t, V, \ell \equiv_{sn} [t ← (0, V)] \ast \bigvee_2 \ell
\]

\[\ast \Diamond_{\ell} \ast \text{spaw Join}(\ell, γ_l, γ)\]^{\gamma}\text{N}

The model for the finish and join handles mirror exactly what we have seen in the proof of \texttt{mp_reclaim} (Figure 11.4). In particular, the finish handle \texttt{FinishHandle}^N(\ell, \Phi) carries the resources needed to send the message (but without \( \Phi(v) \)), mirroring the resources owned by the thread \( \rho \) of \texttt{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 \( \ell \), and (3) the half token \( \Diamond_{\ell}^{\gamma_1/2} \) and the knowledge of the cancelable invariant \([\text{spaw Join}(\ell, γ_l, γ)]\)\text{N}. The join handle \texttt{JoinHandle}^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 \( \pi \) of \texttt{mp_reclaim} from line \( \pi 4 \): (1) the seen-history observation of the flag \( \ell \), (2) the block ownership of \( \ell \), and (3) the other half token \( \Diamond_1^{\gamma_1/2} \) and the knowledge of the same cancelable invariant.

The proofs of \texttt{FINISH-spec} and \texttt{JOIN-spec} then follow almost the same proof of \texttt{mp_reclaim} in Figure 11.4. The main differences is the extra
Context: let $P := \{ \forall \ell, \rho, \text{FinishHandle}^N(\ell, \Phi) \rightarrow w p f([\ell]) \text{ in } \rho \{ \_ . \_ . \_ \} [T] \}$

1: let $\ell := \text{alloc}(2)$ in

2: $\ell := \text{na } 0 \{ \ell \rightarrow 0 * \ell + 1 \rightarrow \_ \_ + \ell \} \text{ // NA-ALLOC and NA-WRITE}$

3: $\textsf{fork} \{ . . . \} \text{ // HOARE-FORK}$

4: $\ell \{ \ell \_ . \_ . \_ \}$

Figure 11.7: Hoare proof outlines for \texttt{SPAWN-SPEC}

releasing and acquiring of the resulting resource $\Phi(v)$. In the proof of \texttt{FINISH-SPEC}, corresponding to line $\rho_2$ of \texttt{mp_reclaim}, when invoking \texttt{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 \texttt{JOIN-SPEC}, at the end of the loop in line $\pi_4$ of \texttt{mp_reclaim}, we acquire also $@_{V_1} \Phi(v)$, which again can be used with \texttt{VA-ELIM} to get $\Phi(v)$.

The proof of \texttt{SPAWN-SPEC} also follows the proof of \texttt{mp_reclaim}. We give its Hoare proof outlines in Figure 11.7.

11.4 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 11.8. The interface includes 5 functions:

1. \texttt{new_stack}([[]]) allocates and returns a new, empty stack handle $s$;
2. \texttt{push}([s, v]) pushes a new element $v$ into the stack $s$;
3. \texttt{pop}([s]) pops and returns an element in $s$, unless $s$ is empty then it returns 0;
4. \texttt{try_push}([s, v]) is a try version of \texttt{push}, which returns true if it has successfully pushed to the stack $s$, or returns false if it fails to do so. \texttt{try_push} can fail due to contention by concurrent pushes/pops. Unlike \texttt{try_push}, \texttt{push} would try again until it succeeds.
5. \texttt{try_pop}([s, v]) is a try version of \texttt{pop}. It returns an element from the stack $s$, or returns empty (0), or returns −1 in case it fails due to
11.4.1 A Release-Aquire 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 sh.

If the CAS fails, it returns false and nothing changes but the node
n’s next field. The CAS may fail if sh 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 sh. If
sh is null (0), it returns empty (0) immediately, in line 3. If sh is
not null, then in line 5 the function reads the pointer to sh’s next
node, stored in the location sh, into n. The function then uses an
acquire CAS to swap n for sh 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 sh + 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.
11.4.2 Bag or Per-element Specifications for Stacks

We will verify the implementation in Figure 11.8 against the so-called bag or per-element specifications, given in Figure 11.9. 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 III. 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 isStack\(N\)(\(s, \Phi\)) which says that the location \(s\) is a stack tied to a predicate \(\Phi : \text{Val} \rightarrow \text{vProp}\). \text{STACK-BAG-NEW} says that new_stack([]) returns a new stack \(s\) that satisfies isStack\(N\)(\(s, \Phi\)) for the user-chosen \(\Phi\) and \(N\). The namespace \(N\) 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 \text{STACK-BAG-TRY-PUSH} and \text{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, \text{STACK-BAG-TRY-POP} and \text{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 \text{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].
11.4.3 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 11.6 (Invariant for Per-element Treiber Stack).

\[
\begin{align*}
\text{null0}(v) &::= (v = \text{Some}(\ell)) \oplus \ell : 0 \\
\text{AllNodes}(vs) &::= \bigstar_{(v, n) \in vs} (v, n \not\in null0(v)) \\
\text{InNodes}(S) &::= \\
& \bigstar_{v, n+1 \rightarrow v \ast \Phi(v) \ast i^2 n \ast \exists q. n \not\in null0(S(i+1))} \\
\text{TreiberBI}(s, \gamma) &::= \\
& \exists V_s, t_0, V_h, vs, S. \\
& \text{let } vs : \text{Loc'} := \text{hd}(S); vs' : (\text{Loc}', \text{View}) := vs \ast ([V_h, V_s]) \text{ in} \\
& \text{let } h := [t_0 + i \leftarrow \text{null0}(v), (v, V)] \text{ in} \\
& \exists_{V_s} (s \not\in \text{can} h) \ast \forall_{V_h} \text{InNodes}(S) \ast \text{AllNodes}(vs') \ast i^1 s \\
\text{isStack}(s) &::= \exists \gamma, h. s \exists_{\text{can}} h \ast \text{TreiberBI}(s, \gamma)^{\gamma} \\
\end{align*}
\]

The invariant content TreiberBI(s, γ) for some location s at an atomic period identifier γ (used by its atomic points-to) is objective, as we will put it in an objective invariant (§10.1) because we do not care to reclaim the stack. Nevertheless, we leave the block ownership (free(1, s) for the stack pointer and i^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. isStack(s, Φ) simply asserts the existence of the objective invariant for TreiberBI(s, γ) in 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 \(v \in \text{Loc'}\) into a nullable location value, where null is the value 0. The invariant content TreiberBI(s, γ) 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 \(s \not\in \text{can} h\) 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

7Note that we do not use the CAS-only mode (cas) atomic points-to yet—that mode was developed mainly for GPS protocols in Part II. 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.
$vs' \in (\text{Loc}^\ast, \text{View})$ 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 $(t_0 + |vs|, \text{null0}(v_h), V_h)$, where $vs$ is the list of non-current writes to $s$. $v_h$ is the head (hd) of the stack’s abstract state $S \in \text{Loc}$, 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 $\text{hd}(S) = \text{None}$, and the latest write to $s$ has the value $\text{null0(}\text{None}) = 0$, i.e., the null value.

- $\text{InNodes}(S)$ 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 $\bigstar$.

  For each node $S(i) = n \in \text{Loc}$ 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 $n \mapsto \text{null0}(S(i + 1))$ 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 $\text{None}$ and the next field of the bottom node will store the null value (0).

- $\text{AllNodes}(vs')$ 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 $n \mapsto \text{null0}(v)$ 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 $s$. $\text{AllNodes}(vs')$ 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 11.7 (On Reclaiming Stack Resources). Looking at the definition of TreiberBl, 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 $\text{InNodes}(S)$ and $\text{AllNodes}(vs')$. $\text{AllNodes}(vs')$ 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 $\text{InNodes}(S)$ and $\text{AllNodes}(vs')$, 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 (§10.2) 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 (§8.1) respectively for allocation and initialization of \( s \). Then we use NA-AT-SW (Figure 9.2) to turn the non-atomic points-to to an atomic points-to in single-writer mode (sw) with some atomic period identifier \( \gamma \), 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 TreiberBl(\( s, \gamma \)), because the history of \( s \) is a singleton with a null location value. We use OINV-ALLOC-OBJ (§10.1) to allocate the invariant, and then we are done.

\[
\begin{align*}
\{ \text{True} \} \\
1: & \text{ let } s := \text{alloc}(1) \text{ in} \\
& \{ s, \begin{array}{l} \exists^1 s \ast s \mapsto \ast \end{array} \} \quad \text{NA-ALLOC} \\
2: & s := \text{na} \; 0; \begin{array}{l} \exists^1 s \ast s \mapsto 0 \end{array} \quad \text{NA-WRITE} \\
& \begin{array}{l} \exists^1 s \ast \exists^2 \gamma, t_0, V_h \mapsto s \ast \exists^2 \gamma, t_0 \mapsto \neg(0, V_h) \ast \exists^2 \gamma, t_0 \mapsto \neg(0, V_h) \end{array} \quad \text{NA-AT-SW and AT-SW-CON and AT-SW-SY and AT-SY-SN and VA-INTRO} \\
& \begin{array}{l} \exists^2 \gamma, t_0 \mapsto \neg(0, V_h) \ast \text{TreiberBl}(s, \gamma) \end{array} \\
3: & s \begin{array}{l} \exists^1 s \ast \text{isStack}^N(s, \emptyset) \end{array} \quad \text{OINV-ALLOC-OBJ}
\end{align*}
\]

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 \( [n] \} \ast S \) and put all of the resources in the pre-condition to that node in the invariant. The proof outlines are given in Figure 11.10.

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 TreiberBl 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., \( s_h \ast \exists^q \text{null0}(n) \) for some \( n \). This fraction will be used to read \( s_h \)’s next field in line 4.

In line 6, if the CAS succeeds, we know that \( s_h \) is indeed the top of the current abstraction \( S \), i.e., \( S = [s_h] \} \ast S' \) for some \( S' \). We then

\*And it is also the source of leaking ownership of nodes in this proof.
\{isStack^V(s, \Phi) \ast \Phi(v) \ast n \rightarrow _* (n + 1) \rightarrow v \ast \dagger^2 n\}

\textbf{Context:}  
\{s \sqsupseteq n \ast \Phi(v) \ast n \rightarrow _* (n + 1) \rightarrow v \ast \dagger^2 n\}_T

\begin{align}
1: \text{let } s_h & := \text{select } s \text{ in} \\
& \{s_h, \exists h_0, t_h, h_0'(t_h) = (s_h, _*) \ast s \sqsupseteq n \ast \text{TreiberBl}(s, \Phi)\}_T \text{ } / \text{ VS-BOT} \\
& \{s \sqsupseteq n \ast \Phi(v) \ast n \rightarrow _* (n + 1) \rightarrow v \ast \dagger^2 n\}_T \\
2: \text{let } n & := \text{na } s_h; \{\Phi(v) \ast n \rightarrow s_h \ast (n + 1) \rightarrow v \ast \dagger^2 n\}_T \text{ } / \text{ NA-WRITE} \\
& \{s \sqsupseteq n \ast \exists V_0, \exists V_0' \ast \exists V_0 \ast \tilde{\Phi}(v) \ast n \rightarrow s_h \ast (n + 1) \rightarrow v \ast \dagger^2 n \ast \text{TreiberBl}(s, \gamma)\}_T \text{ } / \text{ VA-INTRO} \\
& \{s \sqsupseteq n \ast \exists V_0, \exists V_0' \ast \exists V_0 \ast \tilde{\Phi}(v) \ast n \rightarrow s_h \ast (n + 1) \rightarrow v \ast \dagger^2 n \ast \text{TreiberBl}(s, \gamma)\}_T \text{ } / \text{ Unfolding TreiberBl} \\
\end{align}

We know \(s_h\) is a value written to \(h\) at \(t_h\), because \(h_0' \subseteq h\).

\begin{align}
3: \text{CAS}^{rel}(s, s_h, n) \\
& \{b, \exists h', V' \sqsupseteq V_0 \ast V_0', \exists V_0 \ast \tilde{\Phi}(v) \ast n \rightarrow s_h \ast (n + 1) \rightarrow v \ast \dagger^2 n \ast \text{TreiberBl}(s, \gamma)\}_T \text{ } / \text{ AT-CAS-SN-GEN}
\end{align}

Note that any location values readable from \(s\) are alive because we store their fractional non-atomic points-to-ownership in \(\text{AllNodes}(v_s')\).

\begin{align}
& \{b = \text{false} \ast s_h \neq v' \ast h' = h \ast (b = \text{true} \ast s_h = v' \ast \exists V_w \sqsupseteq V' \sqcap V_h \ast h' = h(t_h + 1 \leftarrow (n, V_w)))\}_T \text{ } / \text{ AT-CAS-SN-GEN} \\
& \{b = \text{false} \ast \tilde{\Phi}(v) \ast n \rightarrow _* (n + 1) \rightarrow v \ast \dagger^2 n \ast \text{TreiberBl}(s, \gamma)\}_T \text{ } / \text{ try_push_swap} \\
& \{b = \text{false} \ast \tilde{\Phi}(v) \ast n \rightarrow _* (n + 1) \rightarrow v \ast \dagger^2 n \ast \text{TreiberBl}(s, \gamma)\}_T \\
& \{b = \text{true} \ast \text{TreiberBl}(s, \gamma)\}_T
\end{align}

In the failure case \((b = \text{false})\), we use \text{VA-ELIM} to get back the original resources without the view-at modality.

In the successful case \((b = \text{true})\), the resources are put inside \(\text{InNodes}\) and \(\text{AllNodes}\) of the invariant.

\begin{align}
& \{b = \text{false} \ast \tilde{\Phi}(v) \ast n \rightarrow _* (n + 1) \rightarrow v \ast \dagger^2 n \ast \text{TreiberBl}(s, \gamma)\}_T \text{ } / \text{ try_push_swap} \\
& \{b = \text{true} \ast \text{TreiberBl}(s, \gamma)\}_T
\end{align}

can acquire the resources for \(s_h\) in \(\text{InNodes}(S)\), and leave the rest as \(\text{InNodes}(S')\) for the new abstract state \(S'\).

We give the Hoare proof outlines in Figure 11.11. We note again that we always have deterministic pointer comparison with \(\text{CAS}\)'s, because ownership of all reachable locations in the stack are stored in the invariant \(\text{TreiberBl}\), so we know they are all alive.

\[\square\]

\textbf{Remark 11.8 (On the Use of Acquire CAS in try_pop).} We note that we do not really need the CAS of \textit{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 \textit{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 \(\exists V_n\). The view \(V_n\) should be bigger.
\[\{isStack^V(s, \Phi)\}_T\]

Context: \(\text{TreibenBl}(s, \gamma)\)

\[\{s \triangleright\triangleright \_h_0\}_T\]

\[\{s \triangleright\triangleright \_h_0 \* \triangleright\triangleright \_ \text{TreiberBl}(s, \Phi)\}_T^W \] // VS-BOT

1: let \(s_h := \text{case}\) in

\[\{s_h. \exists_h(t_h, V_0, h'_0(t_h) = (s_h, V_0) \* \triangleright\triangleright \_h_0' \* \text{TreiberBl}(s, \Phi) \* (s_h = 0 \lor s_h \neq 0 \* \exists_h s_h \triangleright\triangleright \_ n)\}_T^W\]

// AT-READ-SN-ACQ

Context: \(h'_0(t_h) = (s_h, V_0) \* \triangleright\triangleright \_h_0'\)

\[\{\text{TreiberBl}(s, \Phi) \* (s_h = 0 \lor s_h \neq 0 \* \exists_h s_h \triangleright\triangleright \_ n)\}_T^W\] // VA-ELIM

2: if \(s_h = 0\) // null

Empty

3: then \(0 \{v, v = 0\}_T\)

4: else

\[\{s_h \triangleright\triangleright \_ n\}\]

5: let \(n := \text{case}_s h \{s_h \triangleright\triangleright \_ n\}_T\] // NA-read

\[\{s_h \triangleright\triangleright \_ n \* \text{TreiberBl}(s, \gamma)\}_T^W\]

\[\{s_h \triangleright\triangleright \_ n \* \exists V_0 \* s \triangleright\triangleright \_ h_0' \* s \triangleright\triangleright \_ s_{\exists_{\text{con}}} h \* \@_{V_0} \text{InNodes}(S) * \text{AllNodes}(v's') \* \dagger_1 s \* \}

\(b \Rightarrow (\text{false} \* s_h \neq v' \* h') = h)\]

\[\{b \Rightarrow (\text{true} \* s_h = v' \* V_0 = V_h \subseteq V' \* t_h = \max(\text{dom}(h)) \* \exists V_0 \* \triangleright\triangleright \_ V_h \* h' = h(t_h + 1 \leftarrow (n, V_w))\}_T^W\]

// AT-CAS-SN-GEN Note that any location values readable from \(s\) are alive because we store their fractional non-atomics points-to ownership in AllNodes(v's').

\[\{\text{case}_s h \{s_h \triangleright\triangleright \_ n \* \text{TreiberBl}(s, \gamma) \* (b = \text{false} \* v \* h') = h\}_T^W\]

\[\{b \Rightarrow (\text{true} \* S = \langle (s_h, V_h) \rangle \* (v' \* \@_{V'} \text{InNodes}(S') * \text{AllNodes}(v's') \* \dagger_1 s * \}

\[\@_{V_0} \text{InNodes}(S') * \text{AllNodes}(v's' \* \dagger_1 s_{\exists_{\text{con}}} h) \* \}

\[\}^W\]

// VA-ELIM since we have \(\exists V'\) and \(V_h \subseteq V'\)

7: then \(\text{case}_s h + 1 \{v, v = -1 \* s_h \triangleright\triangleright \_ n\}_T\]

\[\{v, v = -1 \lor \Phi(v)\}_T\] // Dropping points-to and block ownership of \(s_h\)

\[\}^W\]

8: else \(-1 \{v, v = -1 \* s_h \triangleright\triangleright \_ n\}_T\]

\[\{v, v = -1 \lor \Phi(v)\}_T\]

\[\}^W\]

Figure 11.11: Hoare proof outlines for try_pop
than the view at which the node \( n \) was pushed to the stack, so the seen-view observation \( \exists V_n \) should allow us to access the resources released together with the push of \( n \), i.e., \( (\exists v. n + 1 \mapsto v \ast \Phi(v) \ast \ldots) \) in 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 \text{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., \( \exists V_n^i \). Meanwhile, the resources \( (\exists v. n + 1 \mapsto v \ast \Phi(v) \ast \ldots) \) we want to acquire (when the \text{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 \( \exists V_n^i (\exists v. n + 1 \mapsto v \ast \Phi(v) \ast \ldots) \) in the invariant. So if we want to acquire those resources with a fully relaxed \text{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 \( \exists V_n^i \) we acquired in line 1 to eliminate the view-at modality and acquire \( (\exists v. n + 1 \mapsto v \ast \Phi(v) \ast \ldots) \). In short, to support the fully relaxed \text{CAS} in line 6 of \texttt{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 \texttt{TreiberBI}(\( s, \gamma \)) (Definition 11.6) 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 \text{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 \texttt{try_pop} version with the acquire \text{CAS} (Figure 11.8).

**Chapter Summary.** In this chapter we have demonstrated multiple features of \texttt{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 II 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 \texttt{RBrlx}. Nevertheless, the GPS abstraction in Part II 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 \texttt{try_pop} version that uses a fully relaxed \text{CAS} (see Remark 11.8). In Part III 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.
Part II

RUSTBELT MEETS RELAXED MEMORY
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 typecheck 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.

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 \(Φ(e)\), 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:

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 \(\text{Arc}, \text{Rc}, \text{Cell}, \text{RefCell}, \text{Mutex},\) and \(\text{RwLock}\).

2. *Safety of the \(\lambda_{\text{Rust}}\) type system*: The syntactic typing rules of \(\lambda_{\text{Rust}}\)

---

\(^{1}\)Klabnik and Nichols, *The Rust Programming Language* [KN18].

\(^{2}\)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].
Challenging RustBelt and Relax Memory

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 $P$ is well-typed, and its only uses of unsafe features appear within the libraries that have been verified safe (in part 1), then $P$ 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.

RUSTBelt for Relax Memory. In the original RustBelt work, Iris was instantiated with a sequentially consistent (SC) semantics for $\lambda_{Rust}$. 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 $\lambda_{Rust} + ORC11$ 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 $\lambda_{Rust}$ 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.

### 12.1 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
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 cancellable invariants (Figure 12.1), a generalization of Gotsman et al.’s “storable locks”. A cancellable invariant \( I \in \mathcal{E} \) 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 \( \triangledown \gamma_q \), where \( q \) is a fraction in \((0,1]\). If a thread \( \pi \) can assert ownership of \( \triangledown \gamma_1 \) (i.e., the “full” \( \gamma \) token), it knows that no other thread can assert that the invariant is active; thus it is safe for \( \pi \) 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, cancellable invariants played a crucial role in verifying the safety of destructors such as Arc’s.

However, adapting cancellable invariants to the relaxed-memory setting turns out to be quite tricky—tricky enough that no existing relaxed-memory separation logic supports them. The main problem arises in how to model the cancellable 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 §17), 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 cancellable invariants becomes unsound! In particular, the access rule SC-CINV-ACC is not sound in relaxed memory.

Our solution is to instead model invariant tokens using a novel notion

---

7Gotsman et al., “Local Reasoning for Storable Locks and Threads” [Got+07].
8Hobor et al., “Oracle Semantics for Concurrent Separation Logic” [HAN08].
9iGPS supports a related notion of “fractional protocol”, but it is not nearly as powerful as cancellable invariants and is thus not general enough to account for resource reclamation in Rust.
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 §10.2. In the subsequent chapters in this part, we will demonstrate its effectiveness on a number of real Rust libraries.

### 12.2 Task 2: Re-prove the Safety of the \( \lambda_{Rust} \) Type System under RMC

In contrast to RustBelt’s proofs of safety for libraries, its proof of safety for the \( \lambda_{Rust} \) 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 RB\(_{rlx}\) can inherit RustBelt’s safety proof for the \( \lambda_{Rust} \) 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.

### 12.3 Contributions of RustBelt Relaxed

RustBelt Relaxed, or RB\(_{rlx}\) 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 \( \lambda_{Rust} \)’s type system, by virtue of being built atop the lifetime logic, did not need to be changed at all.

RB\(_{rlx}\) has ported all verifications done in RustBelt, including the following concurrency libraries: `thread::spawn`, `rayon::join`, `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 since been merged into the Rust codebase.\textsuperscript{10} Meanwhile, the verifications of sequential libraries—\texttt{Rc}, \texttt{Cell}, \texttt{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 13 briefly reviews the \textit{lifetime logic}, the core abstraction needed for the original RustBelt’s soundness proof the Rust’s type system, and Chapter 14 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 15 discusses how to construct \textit{GPS single-location protocols}\textsuperscript{11} using atomic points-to ($\S$9), and how to combine them with cancellable invariants ($\S$10.2) to build \textit{cancellable} GPS protocols. Chapter 16 presents how to combine cancellable GPS protocols with the lifetime logic to verify safety of Rust’s \texttt{RwLock} library. Chapter 17 presents how to combine iRC11 cancellable invariants with raw GPS protocols (and the lifetime logic) to verify safety of Rust’s \texttt{Arc} library. It also discusses a bit of history of how the bug in \texttt{Arc} was found.

\textsuperscript{10}Jourdan, \textit{Insufficient synchronization in \texttt{Arc:get_mut}} [Jou18].

\textsuperscript{11}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].
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 11, 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 (§13.1). We will then review in §13.2 how RustBelt's original lifetime logic (in SC) comes to the rescue. In Chapter 14, we will discuss how to port the lifetime logic to relaxed memory.

13.1 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
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 13.1 (Indirect Reclamation of Resources). Consider the following example taken from the RustBelt paper. The example makes use of the function `index_mut` from the `Vec` library.

```rust
let mut v = vec![21, 57];
{
    let mut head = v.index_mut(0);
    // head : &'a mut i32
    // Cannot access v: v.push(42) rejected
    *head = 23;
}

v.push(42);
```

In this example, we start with a vector `v` 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 `v` 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 `v` from its parent block. Borrowing the whole vector is due to `index_mut`, whose type is

```rust
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 borrow 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.

### 13.2 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 13.1. Let us begin with an overview of the new predicates:

- The full borrow `&|^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 `[κ]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 `Kill(κ)` is a unique permission needed to kill the lifetime `κ`.

- The timeless and persistent dead token `[†κ]` is used to witness the knowledge that lifetime `κ` is dead.

- The inheritance `Inh(κ, P)` asserts the right to reclaim the ownership of borrowed resource P once `κ` is dead.
The lifetime logic of SC RustBelt

### Selected Rules of SC RustBelt’s Lifetime Logic

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LFTL-Begin</strong></td>
<td>$\text{LFTL-Begin} (\text{BEGIN})$</td>
</tr>
<tr>
<td><strong>LFTL-Kill</strong></td>
<td>$\text{LFTL-Kill} (\text{Kill}(\kappa))$</td>
</tr>
<tr>
<td><strong>LFTL-Tok-Frac</strong></td>
<td>$\text{LFTL-Tok-Frac} (\text{Ret}(\kappa))$</td>
</tr>
<tr>
<td><strong>LFTL-Tok-Dead</strong></td>
<td>$\text{LFTL-Tok-Dead} (\text{Dead}(\kappa))$</td>
</tr>
<tr>
<td><strong>LFTL-Full-Bor</strong></td>
<td>$\text{LFTL-Full-Bor} (\text{Bor}(\kappa))$</td>
</tr>
<tr>
<td><strong>LFTL-Full-Inh</strong></td>
<td>$\text{LFTL-Full-Inh} (\text{Inh}(\kappa))$</td>
</tr>
<tr>
<td><strong>LFTL-Full-Acc</strong></td>
<td>$\text{LFTL-Full-Acc} (\text{Acc}(\kappa))$</td>
</tr>
<tr>
<td><strong>LFTL-Full-Ret</strong></td>
<td>$\text{LFTL-Full-Ret} (\text{Ret}(\kappa))$</td>
</tr>
<tr>
<td><strong>LFTL-Full-Sep</strong></td>
<td>$\text{LFTL-Full-Sep} (\text{Sep}(\kappa))$</td>
</tr>
</tbody>
</table>

### The Lifecycle of Borrow and Lifetimes

- The **return policy** $\text{Ret}(\kappa, P, q)$ is used as part of the protocol for accessing the contents of a full borrow.

We briefly explain the rules in Figure 13.1 with the help of Figure 13.2, which depicts the life cycle of a lifetime and a full borrow. We start from the right of Figure 13.2, where we create a new lifetime using $\text{LFTL-Begin}$ (BEGIN in Figure 13.2). This yields the full token $[\kappa]_1$ for a new lifetime $\kappa$, as well as the corresponding $\text{Kill}(\kappa)$ permission. Lifetime tokens are **fractional** ($\text{LFTL-Tok-Frac}$, $\text{FRACT}$), so that they can be split into (and joined back from) smaller pieces which enable multiple threads to simultaneously witness that $\kappa$ is still alive.

Next, on the left of Figure 13.2, we see the “flagship” rule of the lifetime logic: given ownership of any assertion $P$, and any lifetime $\kappa$, we can use the borrowing rule $\text{LFTL-Full-Bor}$ (BOR in Figure 13.2) to create a borrow of $P$ for $\kappa$. The rule splits ownership of $P$ in time between two separately ownable assertions: (1) a full borrow $\text{&}_{\text{full}} P$ that represents ownership of $P$ while $\kappa$ is alive; and (2) an inheritance $\text{Inh}(\kappa, P)$ that can be used to reclaim $P$ after $\kappa$ 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 $\text{&}_{\text{full}} P$ and a token $[\kappa]_q$ (proving $\kappa$ is alive) can trade them to obtain $P$ using the accessing rule $\text{LFTL-Full-Acc}$ (ACC in Figure 13.2). As part of the trade, the thread is also given the return policy $\text{Ret}(\kappa, P, q)$. Once the thread is done using $P$, it trades
$P$ and $\text{Ret}(\kappa, P, q)$ to get back \&$\kappa_{\text{full}}$ $P$ and $[\kappa]_q$ \(\text{LFT-FULL-RET, RET in Figure 13.2}\).

Once all accesses to borrows at lifetime $\kappa$ are done, we can recollect the full token $[\kappa]_1$ and use the killer permission $\text{Kill}(\kappa)$ with \text{LFT-KILL} (KILL) to end the lifetime. This yields the dead token $[\kappa]$. Since $\kappa$ is now dead, the content $P$ in \&$\kappa_{\text{full}}$ $P$ cannot be accessed any more and can thus be reclaimed. Anyone owning $[\kappa]$ and the inheritance $\text{Inh}(\kappa, P)$ can use \text{LFT-FULL-INH} (INH in Figure 13.2) to reclaim $P$. \text{LFT-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 $[\kappa]$.

Note that \text{LFT-KILL} uses a wand step viewshift (Notation 6.8), i.e., its user needs an actually step to discharge the later in the viewshift. As such, we use \text{LFT-KILL} around a so-called ghost instruction \text{endlft} (Notation 4.3, Figure 4.2). In fact, $\lambda_{\text{Rust}}$ code will be populated with ghost instructions \text{newlft} and \text{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 13.2, another crucial rule of the lifetime logic is \text{LFT-FULL-SEP}, which lets one go back and forth between a borrow of $P \ast Q$ and separate borrows of $P$ and $Q$. This rule is essential in verifying the soundness of Rust functions like \text{index_mut} (§13.1) that split a reference to an object into references to its subcomponents.

**Example 13.2** (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 13.3. Here, instead of transferring the location $\ell_x$ from thread 1 to thread 2 directly, we transfer a lifetime token, which thread 2 then uses to reclaim ownership of $\ell_x$. The use of the lifetime logic here is clearly overkill since direct ownership transfer of $\ell_x$ already suffices, but it will nonetheless give the reader a concrete feel for the lifetime logic in action.

In Figure 13.3a, we start by creating a lifetime $\kappa$ (\text{LFT-BEGIN}). Then, with the ownership of $\ell_x \mapsto 0$, we create a borrow \&$\kappa_{\text{full}}(\ell_x \mapsto \_)$ using \text{LFT-FULL-BOR}. We assume a send-receive protocol \text{SendRecv} for $\ell_y$ that satisfies the rules \text{SENDRECV-CREATE}, \text{SC-SEND}, and \text{SC-RECV} (in the top of Figure 13.3). 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 \&$\kappa_{\text{full}}(\ell_x \mapsto \_)$, and Send to the thread $\pi$, and give the other half of the $\kappa$ token, the killer, the inheritance, and Recv to thread $\rho$. 
\begin{figure}[h]
\begin{align*}
\text{SEND-SC-SEND} & : \quad \ell_y \mapsto 0 \implies \exists \kappa \in \mathcal{E} \text{ Send}_{\ell_y}(P) \ast \text{Recv}_{\ell_y}(P) \quad \{\text{Send}_{\ell_y}(P) \ast P\} \ell_y :=_{\text{sc}} 1 \text{ \{True}\} \\
\text{SEND-SC-RECV} & : \quad \{\text{Recv}_{\ell_y}(P)\} ^{^{\text{sc}} \ell_y} \{v, v = 0 \lor P\}
\end{align*}
\end{figure}

\begin{figure}[h]
\begin{align*}
\{[\kappa]_{1/2} \ast \text{Kill}(\kappa) \ast \ell_x \mapsto _{-} \ast \text{Send}_{\ell_y}([\kappa]_{1/2})\} & \quad \{[\kappa]_{1/2} \ast \text{Kill}(\kappa) \ast \ell_x \mapsto _{-} \ast \text{Send}_{\ell_y}([\kappa]_{1/2})\} \\
\{[\kappa]_{1/2} \ast \text{Kill}(\kappa) \ast \ell_x \mapsto _{-} \ast \text{Send}_{\ell_y}([\kappa]_{1/2})\} & \quad \ell_x := 42; \quad \ell_x \mapsto _{-} \ast \text{Ret}(\kappa, \ell_x \mapsto _{-}, -1/2) \ast \text{Send}_{\ell_y}([\kappa]_{1/2})
\end{align*}
\end{figure}

\begin{figure}[h]
\begin{align*}
\{[\kappa]_{1/2} \ast \text{Kill}(\kappa) \ast \ell_x \mapsto _{-} \ast \text{Send}_{\ell_y}([\kappa]_{1/2})\} & \quad \text{if } (^{^{\text{sc}} \ell_y} ! = 0) \quad \{[\kappa]_{1/2} \ast \text{Kill}(\kappa) \ast \ell_x \mapsto _{-} \ast \text{Send}_{\ell_y}([\kappa]_{1/2})\} \\
\{[\kappa]_{1/2} \ast \text{Kill}(\kappa) \ast \ell_x \mapsto _{-} \ast \text{Send}_{\ell_y}([\kappa]_{1/2})\} & \quad \ell_x :=_{\text{sc}} 1; \quad \{\text{True}\} \quad \ell_y \mapsto _{-} _{\text{na-write}} \quad \ell_x := 57; \quad \ell_x \mapsto 57 \quad \text{NA-WRITE}
\end{align*}
\end{figure}

In Figure 13.3b, thread \( \pi \) trades the token and the borrow to access \( \ell_x \mapsto _{-} \) with LFTL-FULL-ACC and writes to \( \ell_x \). After that, with LFTL-FULL-RET, thread \( \pi \) trades the return policy and \( \ell_x \mapsto _{-} \) to get back the token and the borrow. Finally, thread \( \pi \) writes to \( \ell_y \) and sends the token \( [\kappa]_{1/2} \) to thread \( \rho \).

In Figure 13.3c, thread \( \rho \) uses Recv to get back the full token. Owning \( \text{Kill}(\kappa) \), it ends the lifetime and earns the dead token \( [\kappa] \) (LFTL-KILL). Combining that with the inheritance, thread \( \rho \) reclaims the ownership of \( \ell_x \mapsto _{-} \) (LFTL-FULL-INH) and can safely write (non-atomically) to \( \ell_x \).

**Remark 13.3 (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 \( \&_{\text{full}} P \) via the accessing rule LFTL-FULL-ACC, the lifetime token \( [\kappa]_{\ell_y} \) and the borrow assertion are “kept” by the return policy and are only returned in exchange for the borrowed content \( P \). By withholding \( [\kappa]_{\ell_y} \) and only returning it after the access finishes, the rule ensures that no party can have the full token \( [\kappa] \) needed to kill \( \kappa \) while others are still accessing borrowed associated with \( \kappa \). Consequently, the inheritance can only be used after all accesses have finished.

This safety property is no different from the safety property CANCEL-SAFE (Property 10.2) of cancelable invariants, both in Iris-SC (Figure 12.1) and in iRCC11 (§10.2). 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 \( \&_{\text{full}} P \) 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 13.2, only the thread \( \pi \) accesses the full borrow, while the thread \( \rho \) 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 §14, we will see that porting atomic borrows to RMC faces the same challenge as porting cancelable invariants to RMC—we need to maintain \textsc{CancelSafe} 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 \( \lambda_{\text{Rust}} + \text{ORC11} \).

Note 13.4 (Rules with Viewshifts). Finally, let us note that the rules given in Figure 13.1 have been streamlined for better representation, with the additional concepts of the killer permission \( \text{Kill}(\kappa) \), the inheritance \( \text{Inh}(\kappa, P) \), and the return policy \( \text{Ret}(\kappa, P, q) \). 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 14.1 (§14.1).

Furthermore, note that the namespace \( N_{\text{lt}} \) is a global, public namespace that is needed to establish the invariants for the internal model of the lifetime logic.

\textbf{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.
14

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 \texttt{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 cancellable invariants (\texttt{CINV-ACC}, §10.2.1) 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 \textit{synchronized ghost state} (Concept 10.5) 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, \textit{the SC rules in Figure 13.1 for lifetimes and full borrows still hold in iRC11.}

We discuss more rules for lifetimes and full borrows in §14.1, and the interface of other borrow forms in §14.2. We simultaneously review these constructs (which are already developed in the original lifetime logic)\textsuperscript{1} and describe the changes due to adaptation as they appear. Readers interested in more details of the original lifetime logic are recommended to refer to the original paper and its technical appendix. In §14.3, we will discuss how to adapt the lifetime logic's model on top of iRC11.

14.1 More Rules for the Lifetime Logic

In addition to the rules in \textit{Figure 13.1}, the relaxed lifetime logic built atop iRC11 also admits stronger rules, some of which are given in \textit{Figure 14.1}. The rule \texttt{LFTL-BEGIN-BD} bundles \texttt{LFTL-BEGIN} and \texttt{LFTL-KILL} together, and similarly \texttt{LFTL-FULL-BOR-BD} bundles \texttt{LFTL-FULL-BOR} and \texttt{LFTL-FULL-INH}, and \texttt{LFTL-FULL-ACC-BD} bundles \texttt{LFTL-FULL-ACC} and \texttt{LFTL-FULL-RET}. The rest of \textit{Figure 14.1} presents stronger rules that are originally from SC lifetime logic, but now adapted to relaxed memory.

14.1.1 Lifetime Tokens Track Observations

\texttt{LFTL-TOK-OBJ-SPLIT} strengthens \texttt{LFTL-TOK-FRAC} by allowing splitting into two bits where one bit can be objective and thus can be put in invariant. This mirrors the same rule \texttt{CINV-TOK-OBJ-SPLIT} (§10.2.1) for iRC11 cancelable invariant tokens $\diamondsuit_q$. Intuitively, both types of tokens are now

\footnotesize
\textsuperscript{1}Jung et al., “RustBelt: Securing the Foundations of the Rust Programming Language” [Jun+18a].
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. \texttt{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 7.10), so that it can be sent to other threads without extra synchronization. The same situation applies for \texttt{CINV-TOK-OBJ-SPLIT} and cancelable invariant tokens.

The rule \texttt{LFTL-TOK-NOT-DEAD-SUBJ} strengthens \texttt{LFTL-TOK-NOT-DEAD}: the fact \([\nabla \kappa]\) that the lifetime is dead is useful globally without the need of synchronization. That is, it is still sufficient if we have \([\nabla \kappa]\) under a subjective modality (§7.6). In fact, it is often the case that we only need \(\langle \text{subj} \rangle [\nabla \kappa]\) (see also faking, below). We would need \([\nabla \kappa]\) locally in inheritance to ensure the safety of inheritance (see \texttt{LFTL-FULL-BOR-BD} or \texttt{LFTL-FULL-INH}, and Remark 13.3).

14.1.2 Faking

\texttt{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 &\texttt{full} \(P\). In other words, the borrow assertions &\texttt{full} \(P\) become meaningless then. \texttt{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 \(\langle \text{subj} \rangle [\nabla \kappa]\)).

14.1.3 Lifetime Inclusion

The Rust’s type system involves subtyping rules that rely on an inclusion between lifetime. Intuitively, a lifetime \(\kappa\) is included in a lifetime \(\kappa’\) if \(\kappa\) is shorter than \(\kappa’\)—in other words, \(\kappa’\) outlives \(\kappa\). The subtyping rule with respect to lifetime (\texttt{T-BOR-LFT} in [Jun+18a, §3.3]) then allows a (type-level) borrow associated with \(\kappa’\) to be a subtype of the borrow associated with the longer \(\kappa’\).

In RustBelt, the (reflexive, transitive) lifetime inclusion relation \(\subseteq\) gives rise to a meet semi-lattice for lifetimes, where the meet composition (\(\sqcap\)) is commutative and associative. In other words, lifetimes \(\kappa\) 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 \texttt{LFTL-INCL-INTER} and \texttt{LFTL-INCL-GLB}. \texttt{LFTL-FULL-SHORTEN} says that a borrow with a lifetime \(\kappa’\) can be turned into a borrow with a shorter lifetime \(\kappa’\). This lifetime-logic-level rule is the model for the type-level subtyping rule \texttt{T-BOR-LFT}.

\texttt{LFTL-TOK-INTER} and \texttt{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 \(\kappa\) or \(\kappa’\) is dead. This
Bundled Rules.

**LFTL-BEGIN-BD**

\[
\text{True} \models_e \exists k. [k]_q * \square([k]_1 \#^{\mathcal{N}_M} [k])
\]

**LFTL-FULL-BOR-BD**

\[
\&_{\text{full}} P * [k]_q \models_{\mathcal{N}_M} \triangleright P \models_{\mathcal{N}_M} \&_{\text{full}} P * ([k] \#^{\mathcal{N}_M} \triangleright P)
\]

**LFTL-FULL-ACC-BD**

\[
\&_{\text{full}} P * [k]_q \models_{\mathcal{N}_M} \triangleright P \models_{\mathcal{N}_M} \&_{\text{full}} P * [k]_q
\]

Lifetime Liveness in Relaxed Memory.

**LFTL-TOK-OBJ-SPLIT**

\[
[k]_q + [k]_q \vdash [k]_q * \langle \text{obj} \rangle [k]_{q'}
\]

**LFTL-TOK-NOT-DEAD-SUBJ**

\[
[k]_q * \langle \text{subj} \rangle [k]_q \vdash \text{False}
\]

**LFTL-BOR-FAKE**

\[
\langle \text{subj} \rangle [k] \models_{\mathcal{N}_M} \&_{\text{full}} P
\]

Lifetime Inclusion.

**LFTL-INCL-INTER**

\[
[k] \subset k' \models k \sqcap k'' \subset k
\]

**LFTL-INCL-GLB**

\[
\begin{align*}
\text{LFTL-INCL-GLB} & : k \sqcap k'' \subset k' \\
\text{LFTL-FULL-SHORTEN} & : \models k' \sqsubseteq k & \models \&_{\text{full}} P \Rightarrow \&_{\text{full}} P
\end{align*}
\]

**LFTL-TOK-INTER**

\[
[k]_q * [k']_{q'} \vdash \exists q''. k \sqcap k'' \Rightarrow [k \sqcap k']_{q''} \rightarrow [k]_q * [k']_{q''}
\]

**LFTL-UNIT-STATIC**

\[
\kappa \sqcap \varepsilon = \kappa
\]

**LFTL-INCL-STATIC**

\[
\kappa \sqsubseteq \varepsilon
\]

**LFTL-STATIC-NOT-DEAD**

\[
\models [\varepsilon] \models \text{False}
\]

**LFTL-TOK-STATIC**

\[
\vdash [\varepsilon]_q
\]

Reborrowing.

**LFTL-REBORROW**

\[
\kappa' \sqsubseteq k \models \&_{\text{full}} P \models_{\mathcal{N}_M} \&_{\text{full}} P * ([k] \#^{\mathcal{N}_M} \&_{\text{full}} P)
\]

**LFTL-BOR-UNNEST**

\[
\&_{\text{full}} P \models_{\mathcal{N}_M} \&_{\text{full}} P \#^{\mathcal{N}_M} \&_{\text{full}} P
\]

Stronger Access Rules.

**LFTL-FULL-ACC-STRONG**

\[
\&_{\text{full}} P * [k]_q \models_{\mathcal{N}_M} \exists k', \kappa \sqsubseteq k' \triangleright P * \left( \forall Q. \triangleright (\triangleright Q * \langle \text{subj} \rangle [k'] \#^{\mathcal{N}_M} \triangleright P) * \triangleright Q \models_{\mathcal{N}_M} \&_{\text{full}} Q * [k]_q \right)
\]

**LFTL-FULL-ACC-ATOMIC-STRONG**

\[
\&_{\text{full}} P \models_{\mathcal{N}_M} \exists k', \kappa \sqsubseteq k' \triangleright (\triangleright P') \triangleright P * \left( \forall Q. \triangleright (\triangleright Q * \langle \text{subj} \rangle [k'] \#^{\mathcal{N}_M} \triangleright P) * \triangleright ([P'] \sqcap Q) \right)
\]

Figure 14.1: More selected rules for lifetimes and full borrows, ported to \(\lambda_{\text{full}} + \text{ORC11}\)
implies that if the shorter lifetime $\kappa$ ($\kappa \subseteq \kappa'$) is alive, the $\kappa'$ is also alive, and if $\kappa'$ is dead, then $\kappa$ must also be dead.

The semi-lattice has a unit $\varepsilon$ (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). Consequently, we can always acquire a fraction of the static lifetime token (LFTL-TOK-STATIC).

**Definition 14.1** (Dynamic Lifetime Inclusion). Lifetime inclusion in RustBelt is in fact defined semantically or dynamically, using a token trading scheme. A lifetime $\kappa$ is included in $\kappa'$ if, given a fraction of the token for $\kappa$, we can produce some fraction of the token for $\kappa'$.

\[ \kappa \subseteq \kappa' \equiv \Box \forall q. [\kappa]_q \Rightarrow \bigstar_{N_{\kappa}} \left( \exists q'. [\kappa']_{q'} \ast \left( [\kappa']_{q'} \Rightarrow \bigstar_{N_{\kappa}} [\kappa]_{q'} \right) \right) \]

14.1.4 Reborrowing

The rule LFTL-REBORROW strengthens LFTL-FULL-SHORTEN: it lets us re-borrow a $\&_{\kappa \subseteq \kappa'} P$ into a borrow $\&_{\kappa} P$ where $\kappa' \subseteq \kappa$. And when the shorter lifetime $\kappa'$ ends, we get our original full borrow back. As such, while LFTL-FULL-SHORTEN can also be seen as a re-borrow, it simply forgets the difference between $\kappa'$ and $\kappa$. On the other hand, LFTL-REBORROW gives us an inheritance, effectively allowing us to regain and use the original borrow when $\kappa'$ is already dead but $\kappa$ is still alive. Note that the inheritance requires the observation of $\kappa'$'s end locally, i.e., not under a subjective (⟨subj⟩) 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 ($\&_{\kappa \subseteq \kappa'} P \&_{\kappa} P$) into a full borrow of the intersected lifetime $\&_{\kappa'} P$. The catch is that we need a wand step viewshift (Notation 6.8) to strip off an extra later modality that appears between the two borrows.

14.1.5 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 ⟨subj⟩ $[\triangleright \kappa]$ when proving that $Q$ entails $P$. In exchange, we get back a full borrow $\&_{\kappa \subseteq \kappa'} Q$ of $Q$. Intuitively, the rule allows us to turn a full borrow $\&_{\kappa} P$ into $\&_{\kappa'} Q$ 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 $\kappa'$ is dead ($[\triangleright \kappa']$). 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 $\kappa \subseteq \kappa'$, which comes from a part of RustBelt’s model for the lifetime logic’s borrows. That is, the model of a full borrow $\&_{\kappa} P$ actually ties the resource $P$ to a bigger lifetime $\kappa'$, and hence the borrow assertion is downward-closed.
with respect to lifetime inclusion, and renders the proofs of rules like \texttt{LFTL-FULL-SHORTEN} easy. This technique is employed quite frequently in the lifetime logic’s model.

Finally, the rule \texttt{LFTL-FULL-ACC-ATOMIC-STRONG} provides a way to access a full borrow \textit{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 \( \kappa \) is still alive or is already dead. If it is already dead, we get that fact subjectively, \( i.e.\), \( (subj) \lbrack \kappa ' \rbrack \). 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 \texttt{LFTL-FULL-ACC-STRONG}. Since we do not need to provide a lifetime token \( \lbrack \kappa \rbrack_q \), the access cannot be non-atomic, because the lifetime \( \kappa \) may be killed during such a non-atomic access. The atomicity of the access is enforced by the mask-changing viewshifts.

Note that 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 \( P' : \text{iProp} \) holds simultaneously \( i.e.\), with a classical conjunction) with \( P \) or \( Q \). The “embed” modality \( \lbrack \cdot \rbrack \) embeds \text{iProp} propositions into \text{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 \text{iProp}) needs to be maintained at the same view.

\texttt{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 \textit{fractured} borrow—which we will see next.

### 14.2 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, \( i.e.\), shared references. This motivates two alternatives of full borrows: \textit{fractured} borrows and \textit{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 (§10.3). This gives rise to \textit{non-atomic} persistent borrows.

All of these different borrows are originally developed in the SC RustBelt work.\footnote{Jung et al., “RustBelt: Securing the Foundations of the Rust Programming Language” [Jun+18a].} Table 14.1 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
Table 14.1: Comparison of borrow types

<table>
<thead>
<tr>
<th>Borrow type</th>
<th>Access type</th>
<th>Access amount</th>
<th>Persistent</th>
<th>Communication among accesses</th>
<th>Access at local view</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full borrows $&amp;_{\text{full}}^\kappa P$</td>
<td>sequential, non-atomic</td>
<td>full</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Non-atomic borrows $&amp;_{\text{na}}^{\kappa/p.N} P$</td>
<td>sequential, non-atomic</td>
<td>full</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Fractured borrows $&amp;_{\text{frac}}^\kappa \Phi$</td>
<td>concurrent, non-atomic</td>
<td>fractions</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Atomic borrows $&amp;_{\text{at}}^{\kappa/p.N} P$</td>
<td>concurrent, atomic</td>
<td>full</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

in §14.3.) All three forms of borrows—$\&_{\text{frac}}^\kappa \Phi$, $\&_{\text{at}}^{\kappa/p.N} P$, and $\&_{\text{na}}^{\kappa/p.N} P$ for fractured, atomic and non-atomic borrows, respectively—are created from a full borrow $\&_{\text{full}}^\kappa P$. 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 $[Na : p.N]$ that ties the borrow to the invariant pool $p.N$.

We now look more closely at each form of borrows, with their respective rules given in Figure 14.2.

### 14.2.1 Fractured Borrows

To meaningfully talk about fractions of resources, fractured borrows assume a predicate $\Phi$ over fractions that is compatible with fraction addition: $\Phi(q_1 + q_2) \Leftrightarrow \Phi(q_1) * \Phi(q_2)$. With that, \texttt{LFTL-FULL-FRACTURE} (Figure 14.2) allows converting a full borrow $\&_{\text{full}}^\kappa \Phi(1)$ of the full resource $\Phi(1)$ into a fracture borrow $\&_{\text{frac}}^\kappa \Phi$. Note that $\Phi$'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 $\kappa$ 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 \texttt{LFTL-FRACT-ACC} gives access to $\Phi(q')$ for some fraction $q'$. Once that exact $q'$ of $\Phi$ is returned, we regain the lifetime token that we started the access with.

\texttt{LFTL-FRACT-SHORTEN} allows shortening the lifetime of a fractured bor-
Fractured borrows.

\[
\text{LTL-FULL-FRACTURE} \quad \vdash \Box (\forall q_1, q_2. \Phi(q_1 + q_2) \iff \Phi(q_1) \ast \Phi(q_2)) \\
\&_{\text{full}}^N P \Rightarrow N_{\text{at}}^N P)
\]

\[
\text{LTL-FRACT-ACC} \quad \&_{\text{frac}}^N \Phi \vdash [\kappa]_q \Rightarrow \exists q'. \vdash \Phi(q') \ast \Box \Phi(q') \Rightarrow N_{\text{at}}^N [\kappa]_q
\]

\[
\text{LTL-FRACT-SHORTEN} \quad \kappa' \sqsubseteq \kappa \vdash \&_{\text{frac}}^N \Phi \Rightarrow \&_{\text{frac}}^{\kappa'} \Phi
\]

\[
\text{LTL-FRACT-IFF} \quad \vdash \Box (\forall q. \Phi(q) \iff \Psi(q)) \vdash \&_{\text{frac}}^\Phi \Phi \Rightarrow \&_{\text{frac}}^\Psi \Psi
\]

Atomic persistent borrows.

\[
\text{LTL-FULL-AT} \quad N \not\equiv N_{\text{at}}^N P \\
\&_{\text{full}}^N P \Rightarrow N_{\text{at}}^N P
\]

\[
\text{LTL-ACC} \quad \&_{\text{at}}^N P \vdash [\kappa]_q \text{ if } N_{\text{in}}^N \Rightarrow N_{\text{at}}^N \exists \forall q. \vdash P \ast \Box P \Rightarrow N_{\text{at}}^N N_{\text{in}}^N [\kappa]_q
\]

\[
\text{LTL-SHORTEN} \quad \kappa' \sqsubseteq \kappa \vdash \&_{\text{at}}^N P \Rightarrow \&_{\text{at}}^{\kappa'} P
\]

\[
\text{LTL-IFF} \quad \vdash \Box (P \iff Q) \vdash \&_{\text{at}}^{\kappa'/N} P \Rightarrow \&_{\text{at}}^{\kappa'/N} Q
\]

Non-atomic persistent borrows.

\[
\text{LTL-FULL-NA} \quad \&_{\text{full}}^N P \Rightarrow N_{\text{in}}^N \&_{\text{na}}^{\kappa/p,N} P
\]

\[
\text{LTL-ACC} \quad \&_{\text{na}}^{\kappa/p,N} P \vdash [\kappa]_q \ast \text{ Na : p} \cdot N \Rightarrow N_{\text{in}}^N \ast \forall q. \vdash P \ast \Box N_{\text{in}}^N [\kappa]_q \ast [\text{Na : p} \cdot N]
\]

\[
\text{LTL-SHORTEN} \quad \kappa' \sqsubseteq \kappa \vdash \&_{\text{na}}^{p,N} P \Rightarrow \&_{\text{na}}^{\kappa'/p,N} P
\]

\[
\text{LTL-IFF} \quad \vdash \Box (P \iff Q) \vdash \&_{\text{na}}^{\kappa/p,N} P \Rightarrow \&_{\text{na}}^{\kappa/p,N} Q
\]

4: Other Forms of Borrows 181

row, like \text{LTL-FULL-SHORTEN} for full borrows. \text{LTL-FRACT-IFF} says that fractured borrows are closed under the equivalence of the fractional predicate.

14.2.2 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 \text{LTL-AT-ACC} using mask-changing viewshifts, like \text{INV-ACC} or \text{CINV-ACC}. Recall that the namespace \( N_{\text{at}}^N \) is used for the internal invariant of the lifetime logic’s model. The client of atomic borrows picks a namespace \( N \) disjoint from \( N_{\text{at}}^N \), at the creation of a atomic borrow from a full borrow using \text{LTL-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 \textit{immutable} shared resources, atomic borrows are designed to model \textit{mutable} shared resources in concurrent libraries. Naturally, these libraries use atomic memory accesses whose rules \(^5\) are compatible with \text{LTL-AT-ACC}.

\(^5\) e.g., those in §9

Most importantly, \text{LTL-AT-ACC} only allows accesses to the borrowed resource \( P \) under a view-join modality. That is, the access gives us

\[ q. \Phi(q) \iff \Psi(q) \]\n
\( \Downarrow_{V_b} \triangleright P \) and requires us to return the same \( \Downarrow_{V_b} \triangleright P \) at the end of the access. This is exactly the same requirement as that of the access rule \textsc{CInv-acc} for iRC11 cancelable invariants (§10.2.1), 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 14.1. 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 \textsc{LftL-at-acc} without the view-join modality were sound, a thread \( \pi \) would access and modify \( P \) (i.e., after the access \( P \) holds at \( \pi \)'s local view), and immediately in the next step a different thread \( \rho \) would get synchronized access to \( P \) at its own local view, including the modifications made by \( \pi \). In that case, the logic would allow synchronization from \( \pi \) to \( \rho \) 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., \( \Downarrow_{V_b} \triangleright P \), and this would make atomic borrows roughly related to objective invariants (§10.1). 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 13.3), in the same exact way as how iRC11 cancelable invariants maintains \textsc{Cancel-safe} (§10.2).

Finally, \textsc{LftL-at-shorten} and \textsc{LftL-at-iff} say that atomic borrows are closed with respect to lifetime inclusion and predicate equivalence.

### 14.2.3 Non-Atomic Persistent Borrows

A non-atomic borrow \&\textsc{na}\( \kappa/p, N \) \( P \) can be created from a full borrow \&\textsc{full}\( \kappa \) \( P \) using \textsc{LftL-full-na} (Figure 14.2). Non-atomic borrows offer the same sequential, non-atomic access style as that of full borrows, but manages the unique access restriction through the invariant token \([Na : p.N] \) (for the namespace \( N \) under the pool \( P \)), instead of the borrow assertion \&\textsc{full}\( \kappa \) \( P \) itself like in the case of full borrows. As such, non-atomic borrow assertions \&\textsc{na}\( \kappa/p, N \) \( P \) 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 RefCe11.

The access rule \textsc{LftL-na-acc} thus requires the invariant token \([Na : p.N] \) in addition to the lifetime token \([\kappa]_q \). Under the hood, the rule is supported by the non-atomic invariant access rule \textsc{Nainv-acc} and the access rules for borrows.

\textsc{LftL-na-shorten} and \textsc{LftL-na-iff} say that non-atomic borrows are also closed with respect to lifetime inclusion and predicate equivalence.
14.3 Adaption of the Lifetime Logic’s Model in iRC11

In this section, we briefly discuss the adaption 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 14.2 (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 \( \text{LftInv}(\kappa) \) for each lifetime \( \kappa \) is roughly as follows.

- \( \kappa \) is either alive or dead: \( \text{LftInv}(\kappa) = \text{LftAlive}(\kappa) \lor \text{LftDead}(\kappa) \).

- For each borrow \( \&^\kappa P \) associated with \( \kappa \), when \( \kappa \) is already dead, \( \text{LftDead}(\kappa) \) tracks if the inheritance \( \text{Inh}(\kappa, P) \) has been used. For borrows whose inheritance have not been used, \( \text{LftDead}(\kappa) \) carries the resources of those borrows. This part of the protocol is needed for \text{LFTL-FULL-INH}.

- When \( \kappa \) is still alive, \( \text{LftAlive}(\kappa) \) tracks the states of each borrow \( \&^\kappa P \) associated with \( \kappa \). The borrow can be in one of three states:
  - The borrow is not being accessed, and the resource \( P \) is still being hold by \( \text{LftAlive}(\kappa) \).
  - The borrow is being opened by someone, so the resource \( P \) has been taken out of \( \text{LftAlive}(\kappa) \). On the other hand, as part of the borrowing process, some fraction \( q \) of the lifetime token \( [\kappa]_q \) must have been put in \( \text{LftAlive}(\kappa) \) as a deposit. This corresponds to the rules \text{LFTL-FULL-ACC} and \text{LFTL-FULL-RET}.
  - The borrow has been reborrowed at a strictly shorter lifetime \( \kappa' \). \( \text{LftAlive}(\kappa) \) then needs to track how to reclaim resources for \( \&^\kappa P \) from the reborrowers when \( \kappa \) is killed. This part of the protocol models the reborrowing rule \text{LFTL-REBorrow}.

As typical for logics built in Iris, the lifetime invariant \( \text{LftInv}(\kappa) \) together with the lifetime and borrow assertions (e.g., the lifetime token \( [\kappa]_q \) or the borrow assertion \( \&^\kappa \text{full} P \), as listed in §13.2) 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 \text{LFTL-TOK-NOT-DEAD}), and \( \&^\kappa \text{full} P \) is defined with an exclusive ghost element, together with some invariant that ties \( P \) to the global invariant \( \text{LftInv}(\kappa) \).

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 \( \kappa \):

\[
[\kappa]_q := \dagger^q [\kappa] \quad \text{(LFTL-TOK-SC-MODEL)}
\]
This model is insufficient for the RMC lifetime logic, because it does not guarantee the safety of inheritance (Remark 13.3) in the presence of concurrent borrow accesses that do not establish synchronization. In RBrlx, we instead need to enrich this model so that it depends on the view at which $[\kappa]_q$ is asserted:

$$[[\kappa]_q] := \exists V_{tok} \cdot [\kappa]_q V_{tok} \, V_{tok} \ast V_{tok} \tag{LFT-TOK-RLX-MODEL}$$

In this model, the ghost element is no longer just a fraction $q$, but a pair of the fraction and the token view $V_{tok}$. The token view $V_{tok}$ 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: $\sqsubseteq V_{tok}$.

Note that this model of lifetime token is exactly the same as the model of cancelable invariant token $\bigotimes_q$, given by CI-NV-MODEL-TOK,\(^{11}\) 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.

14.3.1 Full Borrows

**Proving Inheritance.** To prove LFT-L-FULL-INH (Figure 13.1) sound for RBrlx, we get to assume $[[\kappa]]$ and $\text{Inh}(\kappa, P)$ 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 $[[\kappa]]$, we know that the global lifetime invariant LftInv($\kappa$) is in the LftDead($\kappa$) disjunct, and by owning $\text{Inh}(\kappa, P)$, we know that the inheritance has not been applied yet (we are the one to apply it), so LftDead($\kappa$) is still holding on to $P$ at the view $V_b$. In other words, LftDead($\kappa$) is holding $\forall V_b \triangleright P$.

When we apply the inheritance, we will therefore be trading $\text{Inh}(\kappa, P)$ for $\forall V_b \triangleright P$ from LftDead($\kappa$). If we can show that $\exists V_b$, i.e., the current thread has locally observed $V_b$, we can use VA-ELIM (Figure 7.3) to obtain $\triangleright P$ and finish the proof.

The key to proving the goal $\exists V_b$ 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 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 $\&_{\text{full}} P$:

- the content view $V_b$ at which the lifetime invariant keeps $P$;
- the token views $V_{\text{tok}}$, one for every $[\kappa]_q$;
- the full token view $V_\kappa$ of the full token $[\kappa]_1$, defined as the join of all token views $V_{\text{tok}}$;
- the dead token view $V_\dagger \subseteq V_\kappa$; and
- the borrow view $V_B$ of the borrow assertion $\&_{\text{full}} P$.

In order to prove $\subseteq V_b$ for LFT-\textsc{full-inh}, we enforce the following property in $\text{LftInv}(\kappa)$ for the borrow in question:

$$V_b \subseteq V_\kappa \quad \text{(LFT-\textsc{full-bor-inv-1})}$$

By owning $[\dagger \kappa]$, we have for the current thread $\sqsubseteq V_1$. By view-monotonicity (i.e., VS-MONO, Figure 7.3), the current thread must have observed $\sqsubseteq V_\kappa$ and $\sqsubseteq V_b$. □

This shows that LFT-\textsc{full-bor-inv-1} allows us to prove LFT-\textsc{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 ($V_b \subseteq V_\kappa$). 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 $P$ ($V_b \subseteq V_\kappa \subseteq V_1$), so it can use the inheritance to reclaim $P$ at its local view.

How do we maintain LFT-\textsc{full-bor-inv-1}? $V_b \subseteq V_\kappa$ is maintained by the rule LFT-\textsc{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 $V_{\text{tok}}$ to $V_{\text{tok}} \sqcup V_c'$ before returning it to the user. Since $V_\kappa$ is the join of all lifetime tokens, this effectively updates $V_\kappa$ to $V_{\kappa} \sqcup V_c' \sqsubseteq V_c'$. 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-\textsc{acc} in §10.2.2.

Proving accesses. To prove the rule for accessing full borrows, LFT-\textsc{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 $\text{Ret}(\kappa, P, q)$, we still need to ensure that $P$ holds locally, i.e., we have $\sqsubseteq V_b$. For this, we rely on the following invariant:

$$V_b \subseteq V_B \quad \text{(LFT-\textsc{full-bor-inv-2})}$$

That is, the content view $V_b$ of $P$ is always included in the borrow view $V_B$ of $\&_{\text{full}} P$. In other words, owning $\&_{\text{full}} P$ should imply $\sqsubseteq V_B$, which
in turn implies $\exists V_b$. Like $\text{LFTL-FULL-BOR-INV-1}$, $\text{LFTL-FULL-BOR-INV-2}$ is straightforward to maintain: the borrow assertion $\&_{\text{full}} P$ is withheld during the access; and when an access is returned, we update the borrow assertion’s $V_B$ with the new content view $V_C$.

### 14.3.2 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 $\Phi$ 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 $\Phi$ is no longer a single view, but consists of two views:

1. the view $V_{\text{YTBA}}$ of the yet-to-be-accessed portion of $\Phi$,
2. the view $V_{\text{AA}}$ of the already-accessed portion of $\Phi$.

$V_{\text{YTBA}}$ is the view of the chunk of $\Phi$ that has not been given out to any access, as well as the view at which the fractured borrow was created. Meanwhile, $V_{\text{AA}}$ 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 $\&_{\text{frac}} \Phi$:

$$V_{\text{YTBA}} \cup V_{\text{AA}} \subseteq V_{\kappa} \quad (\text{LFTL-FRACT-BOR-INV-1})$$

That is, instead of using a single content view $V_b$ like in $\text{LFTL-FULL-BOR-INV-1}$, we use the view $V_{\text{YTBA}} \cup V_{\text{AA}}$ which is the view of the full resource $\Phi(1)$, and require that it is always included in the full token view $\kappa$. Thus, by similar reasoning to $\text{LFTL-FULL-BOR-INV-1}$, 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:

$$V_{\text{YTBA}} \subseteq V_B \quad (\text{LFTL-FRACT-BOR-INV-2})$$

This invariant enables the synchronized access to a fraction of $\Phi$ in the accessing rule $\text{LFTL-FRACT-ACC}$: when the rule is applied with a thread owning a borrow assertion $\&_{\text{frac}} \Phi$ with some borrow view $V_B$, we have that the thread has observed $V_B$, i.e., $\exists V_B$. Consequently, the thread has $\exists V_{\text{YTBA}}$, and it can then obtain some portion $\Phi(q')$ from the yet-to-be-accessed chunk at its local view.

### 14.3.3 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 $\&_{\text{at}}^N P$, 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 $\text{CI}_{\text{NV-ACC}}$, the atomic borrow access rule $\text{LFTL-AT-ACC}$ 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 $\text{VJ-Elim}$ (Figure 7.3) 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 Section 10.2.2).

**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 $\lambda_{\text{Rust}}$ 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.
15

*GPS Single-Location Protocols*
16

Verification of RwLock
Verification of Arc

The verification of the Arc library is by far the most challenging library verification in RB\textsubscript{rlx}. To make the verification go through, we needed to strengthen two atomic reads from rl\textsubscript{lx} 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 §17.4.

Arc\textsubscript{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\textsubscript{T}).

Example 17.1 (A Client of Arc). The following Rust example instantiates Arc with an atomic integer Atomic\textsubscript{Usize} and demonstrates how Arc is typically used:

```rust
let arc1 = Arc::new(Atomic\textsubscript{Usize}::new(5)); // create the first Arc
let arc2 = Arc::clone(&arc1); // clone the second Arc
thread::spawn(move || { // give arc2 to child thread
    println!("child: {:?}", arc2.fetch_add(1, Ordering::Relaxed));
    // drop(arc2);
});
println!("main: {:?}", arc1.fetch_add(2, Ordering::Relaxed));
// 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 decre-
new(v) ::= let a := alloc(2) in
    a.counter := na 1;
    a.data := na v;
    a
deref(a) ::= *na.a.data

drop(a) ::= if FAA^{rel}(a.counter, -1) == 1
    fence_{acq};
    free(a, 2)

clone(a) ::= FAA_{rlx}(a.counter, 1);
    a

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.

17.2 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. The rule NA-dealloc (Figure 8.1, §8.1) states the requirement for deallocating a single location ℓ: we need to have the full ownership of ℓ, represented by its
non-atomic points-to assertion \( \ell \mapsto \_ \). To deallocate a block \( a \) of two locations using \texttt{free}(\( a, 2 \)), the general deallocation rule requires us to have the full ownership of the whole block i.e., both \( a\.data \mapsto \_ \) and \( a\.counter \mapsto \_ \).

In short, we start out with the full ownership \( a\.data \mapsto v \) and \( a\.counter \mapsto 1 \) in the \texttt{new} function, then we share both \( a\.data \) and \( a\.counter \) for concurrent accesses, and at the end reclaim both \( a\.data \mapsto \_ \) and \( a\.counter \mapsto \_ \) in the last \texttt{drop} for deallocation. Our job is to set up the sharing to satisfy this scheme. Because the \texttt{data} field only needs concurrent \texttt{read} accesses, we employ \textit{fractional ownership}\(^1\) on the points-to assertion \( a\.data \mapsto v \). That is, we start out with the full fraction \( a\.data \mapsto v = a\.data \mapsto v \) and for every newly-created pointer we give it a small fraction \( a\.data \rightarrow_{v} v \), where \( q \in (0, 1) \). Each fractional points-to assertion \( a\.data \rightarrow_{v} v \) is sufficient to perform concurrent reads. When a pointer goes out of scope, its small fraction \( a\.data \rightarrow_{v} v \) is recollected. Before the very end, we recollect all the small fractions into the full fraction \( a\.data \mapsto v = a\.data \mapsto v \). Then we are ready for deallocating \( a\.data \).

The \texttt{counter} field, on the other hand, needs concurrent \texttt{FAA} accesses, so we will use \texttt{iRC11} cancellable single-location invariants to share it. The cancellable invariant is also used for recollecting the small fractions of the \texttt{data} field. And now we need to understand what a \texttt{iRC11} cancellable single-location invariant is.

\begin{align*}
\text{GPS-CINV-NEW} & & q + q' = 1 \quad \forall \tau, \tau_q \star P \Rightarrow \mathcal{I}(v) \star Q \\
& & \ell \mapsto v \star P \Rightarrow \exists \tau, \tau_q \star \tau \star P \Rightarrow \mathcal{I}(v) \star Q \\
\text{GPS-CINV-FAQ-RLX} & & \forall v, P \ast \mathcal{I}(v) \Rightarrow \mathcal{I}(v + n) \star Q(v) \\
& & \forall \tau, \tau_q \star \mathcal{I}(v) \Rightarrow \exists \tau, \tau_q \star \tau \star \mathcal{I}(v + n) \star Q(v) \\
\text{GPS-CINV-FAQ-SREL} & & \forall v, \tau \ast \mathcal{I}(v) \Rightarrow \exists \tau \ast \mathcal{I}(v + n) \star Q(v) \\
\text{GPS-CINV-CANCEL} & & \ell \mapsto [\tau_q \ast \mathcal{I}(v) \ast Q(v)] \\
\end{align*}

\(^1\)Boyland, “Checking interference with fractional permissions” [Boy03].
17.2.1 The Specifications for Core Arc

Using fractional ownership for the data field and cancellable invariant for the counter field of Core Arc, we want to prove the following simple specification:

\[
\{ \text{True} \} \text{new}(v) \{ a. \exists \tau, \gamma. \text{ARC}^\gamma(a, v, \tau, I) \} \quad (\text{ARC-NEW-SPEC})
\]
\[
\{ \text{ARC}^\gamma(a, v, \tau, I) \} \text{clone}(a) \{ \text{ARC}^\gamma(a, v, \tau, I) \land \text{ARC}^\gamma(a, v, \tau, I) \} \quad (\text{ARC-CLONE-SPEC})
\]
\[
\{ \text{ARC}^\gamma(a, v, \tau, I) \} \text{drop}(a) \{ \text{True} \} \quad (\text{ARC-DROP-SPEC})
\]

17.2.2 The GPS Protocol for Core Arc

**Definition 17.2** (Model of the ARC Assertion). We define an abstract predicate \( \text{ARC}^\gamma(a, v, \tau, I) \) to represent the logical ownership of an Arc pointer:

\[
\text{ARC}^\gamma(a, v, \tau, I) := \exists q. a.\text{data} \xrightarrow{\gamma} v * [\tau]_q \Rightarrow \begin{cases} a.\text{counter} \xrightarrow{\gamma} \text{Count}(q) \\ (\text{ARC-MODEL}) \end{cases}
\]

Owning an Arc pointer \( \text{ARC}^\gamma(a, v, \tau, I) \) means that we own: (1) some small fraction \( q \) of the data field \( a.\text{data} \xrightarrow{\gamma} v \) at the value \( v \), which allows us to safely read \( a.\text{data} \) for the value \( v \); (2) the fact that the counter field is governed by an invariant \( I \) protected by \( \tau \), as well as the access token \( [\tau]_q \)—with the same fraction \( q \)—to access the invariant, which allows us to concurrently access \( a.\text{counter} \); and finally (3) an unsynchronized ghost element \( \text{Count}(q) \) that represent the single count of this pointer in the total count (see below).

**Definition 17.3** (GPS Protocol for a.counter). The protocol for the location \( a.\text{counter} \) is defined as follows:

\[
I^{\gamma.\text{data}}(n) := \begin{cases} \text{False} & n < 0 \\ \{ \text{TotalCount}(0, 0) \} & n = 0 \\ \exists q_{in}, q_{out} \in (0, 1). a.\text{data} \xrightarrow{\gamma} v_{data} * [\tau]_{q_{in}} * q_{in} + q_{out} = 1 * \text{TotalCount}(n, q_{out}) & n > 0 \\ (\text{ARC-INV}) \end{cases}
\]

First, \( I \) 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 \( v \) and the invariant owns the unsynchronized ghost element \( \text{TotalCount}(v, q_{out}) \). The element \( \text{TotalCount}(v, q_{out}) \) tracks the globally-consistent knowledge that there are currently \( v \) pointers and the sum of all fractional permissions owned by those pointers is \( q_{out} \). The invariant further requires that the remaining fractions \( q_{in} = 1 - q_{out} \) must be owned by the invariant. This includes the fractional ownership of \( a.\text{data} \) and the access token \( [\tau]_{q_{in}} \) of a.counter. The fraction \( q_{in} \) is in fact the used fraction that has been recollected by \( I \) from the pointers that have been dropped. Thus the invariant makes sure that any fractions of the \( a.\text{data} \) and \( \tau \) are all accounted for. Finally, when the counter reaches 0, the invariant is simply trivial.

---

[2] Here, out means ownership of the fractions outside of the invariant.
COUNTING-START
\( q \in (0,1] \vdash \exists \gamma. \{ \text{TotalCount}(n,q) \}^\gamma \ast \{ \text{Count}(q) \}^\gamma \)

COUNTING-NEW
\( q + q' \leq 1 \vdash \{ \text{TotalCount}(n,q) \}^\gamma \Rightarrow \{ \text{TotalCount}(n + 1,q + q') \}^\gamma \ast \{ \text{Count}(q) \}^\gamma \)

COUNTING-AGREE
\( \{ \text{TotalCount}(n,q) \}^\gamma \ast \{ \text{Count}(q) \}^\gamma \vdash n \geq 1 \wedge q' \leq q \leq 1 \)

COUNTING-DROP
\( \{ \text{TotalCount}(n + 1,q + q') \}^\gamma \ast \{ \text{Count}(q) \}^\gamma \Rightarrow \{ \text{TotalCount}(n,q) \}^\gamma \wedge (n = 0 \Rightarrow q = 0) \)

The ghost elements \( \{ \text{TotalCount}(n,q) \}^\gamma \) and \( \{ \text{Count}(q) \}^\gamma \) is an instance of counting permissions, used here to track the outside fractions associated with each single count. They satisfy the axioms in Figure 17.3.

17.2.3 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 \( a.\text{data} \rightarrow v \) and \( a.\text{counter} \rightarrow 1 \) to the abstract permission \( \text{ARC}^\gamma (a,v,\tau,I) \) for some \( \tau \) and \( \gamma \). That is, turning our unique ownership into sharing mode.

To do so, we have planned to initialize iRC11 cancellable invariant for a.counter. The rule GPS-CINV-NEW (Figure 17.2) creates for the location \( \ell \) a new cancellable invariant protected by some token \( \tau \). As a result, we get the full token \( [\tau]_I \) which can be split using GPS-CINV-TOK so that the pieces can be given to multiple threads for sharing. What we need to provide are the points-to \( \ell \rightarrow v \) and the interpretation \( I(v) \).

For a.counter, we do have its points-to assertion as \( a.\text{counter} \rightarrow 1 \), so we only need to provide \( I^{\gamma,a}(1) \) for some \( \gamma \). First, for \( \gamma \), we use COUNTING-START to create the total count and the first single count with \( q := 1/2 \). That is, we get \( \{ \text{TotalCount}(1,1/2) \}^\gamma \) and \( \{ \text{Count}(1/2) \}^\gamma \). We use \( \{ \text{TotalCount}(1,1/2) \}^\gamma \) for \( I \) and \( \{ \text{Count}(1/2) \}^\gamma \) for ARC. Similarly, we split \( a.\text{data} \rightarrow v \) into two halves \( a.\text{data} \rightarrow^{1/2} v \)'s and use each for \( I \) and ARC.

With \( a.\text{data} \rightarrow^{1/2} v \) and \( \{ \text{TotalCount}(1,1/2) \}^\gamma \), we only need \( [\tau]_{1/2} \) for \( I^{\gamma,a}(1) \). Fortunately, GPS-CINV-NEW allows us to use some of the token to establish \( I \). In our case, we need \( [\tau]_{1/2} \) to complete \( I^{\gamma,a}(1) \).
We pick \( q = q' := 1/2 \) and \( P := a.data \cdot \tau \cdot v + \{\text{TotalCount}[1/2]^{\gamma} \} \), and thus establish the cancellable invariant for \( a \cdot \text{counter} \). We get as a result \( [\tau]_{1/2} \cdot \gamma \cdot \text{counter} \). We combine this with the remaining \( a.data \cdot \tau \cdot v + \{\text{Count}[1/2]^{\gamma} \} \) to complete the first \( \text{ARC}^*(a, v, \tau, I) \) permission.

17.2.4 Verifying clone

In proving \( \text{ARC-CLONE-SPEC} \), we need to split one \( \text{ARC}^*(a, v, \tau, I) \) into two. Unfolding the definition of \( \text{ARC}^*(a, v, \tau, I) \) (see \( \text{ARC-MODEL} \)), we see that the fractions \( a.data \cdot \tau \cdot v + [\tau]_{q} \) (for some \( q \)) can be split into halves i.e., \( a.data \cdot \tau \cdot v + [\tau]_{q/2} \), each for one new \( \text{ARC} \). The invariant assertion \( [\text{a.counter} \cdot \tau]^{\gamma} \) is freely duplicable. So we only need to transform one single count \( \{\text{Count}[\gamma]^{\gamma} \} \) into two. To match the fraction \( q/2 \), we actually need two single counts of the form \( \{\text{Count}[\gamma/2]^{\gamma} \} \). Unfortunately, \( \{\text{Count}[\gamma/2]^{\gamma} \} \) is not splittable into two \( \{\text{Count}[\gamma/2]^{\gamma} \} \)’s. So we can only get two \( \{\text{Count}[\gamma]^{\gamma} \} \)’s with the help of the total count \( \{\text{TotalCount}[\gamma]^{\gamma} \} \), which is inside the invariant. To do so, at the relaxed \( \text{FAA} \) made by \( \text{clone} \) (Figure 17.1), we invoke the rule \( \text{GPS-CINV-FAA-RLX} \).

First, the key novelty of our logic compared to previous logics is the ability to cancel a single-location invariant. Here, the \( \text{GPS-CINV-FAA-RLX} \) (Figure 17.2) is an access rule to a cancellable invariant on a location \( \ell \). In order to safeguard the access, the rule must know that the invariant has not been canceled. Thus it requires such a proof from us (the invoker of the rule) in the form of the access token \( [\tau]_{q} \) (see the precondition of the Hoare triple in \( \text{GPS-CINV-FAA-RLX} \)). The token \( [\tau]_{q} \) proves that no one has used the full token \( [\tau]_{1} \) to cancel the invariant. The rule additionally withholds the token during the access and only returns it afterwards (see the postcondition of the Hoare triple).

Second, a \( \text{FAA} \) is a read-modify-write (RMW) operation that has the effect of both a read and a write, and thus can make use of the interpretation \( I \) of the value it read for the interpretation of the value it is going to write. This is demonstrated in the premise of \( \text{GPS-CINV-FAA-RLX} \). Here, \( v \) is the value read and \( v + n \) is the value to be written. The rule allows us to use some of our local resource \( P \) and the interpretation of the read \( I(v) \) to establish the interpretation of the write \( I(v + n) \), and we can additionally take out any remaining resource \( Q \). This is the standard way in RMM logics for RMW operations to communicate with other reads or RMWs. In our case, it is the way for \text{clone’s} to communicate about the total count (see below).

Third, in \text{clone}, we use a relaxed \( \text{FAA} \) which is a relaxed read and a relaxed write. Therefore in order to use our local resource \( P \) for the interpretation, \( P \) needs to be protected by a release modality: \( \Delta P \) (see the precondition of the Hoare triple in \( \text{GPS-CINV-FAA-RLX} \)). A resource can be put under a release modality if that resource is available at the last release fence, as required by the rule \text{HORACE-REL-FENCE}. On the other hand, a relaxed read gives us a resource \( Q \) under the acquire modality: \( \nabla Q \) (see the postcondition in \( \text{GPS-CINV-FAA-RLX} \)). The acquire modality
can be removed by an acquire fence, as shown in the rule HOARE-ACQ-FENCE. Together the two fence rules establish the synchronization pattern of the chain “release fence → relaxed write → relaxed read → acquire fence”.

Finally, 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 GPS-CINV-FAA-RLX for clone.

In particular, as we have \( \{\text{Count}(q)\} \), we use GHOST-RELMOD to get \( \Delta_{\text{count}(q)} \). Then, using our token \([\tau]_q\), we invoke GPS-CINV-FAA-RLX with

\[
P := \{\text{Count}(q)\} \quad \text{and} \quad Q := \{\text{Count}(q/2)\} * \{\text{Count}(q/2)\}.
\]

We now have to show that

\[
\mathcal{I}^{\tau,v}(v') * \{\text{Count}(q)\} \Rightarrow \mathcal{I}^{\tau,v}(v' + 1) * \{\text{Count}(q/2)\} * \{\text{Count}(q/2)\},
\]

where \( v' \) is the value the FAA reads from a counter.

First, by the definition of \( \mathcal{I}^{\tau,v}(v') \) (see ARC-INV), we know that \( v' \geq 0 \). By owning \( \{\text{Count}(q)\} \), we also know that \( v' \) cannot be 0, because if \( v' = 0 \), we can combine \( \{\text{TotalCount}(0,0)\} \) with \( \{\text{Count}(q)\} \) and use the rule COUNTING-AGREE to derive the contradiction that \( 0 \geq 1 \). Thus \( v' > 0 \).

Now, we are not going to change the fractions \( (q_{in/out}) \) and the fractional ownerships: we will keep them the same (i.e., framing) for \( \mathcal{I}^{\tau,v}(v' + 1) \). Therefore our job is simply transform \( \{\text{TotalCount}(v',q_{out})\} * \{\text{Count}(q)\} \) to \( \{\text{TotalCount}(v' + 1,q_{out})\} * \{\text{Count}(q/2)\} * \{\text{Count}(q/2)\} \).

This is simple: We first use COUNTING-DROP to drop the single count \( \{\text{Count}(q)\} \) associated with \( q \) and get \( \{\text{TotalCount}(v' - 1,q_{out} - q)\} \). We then call COUNTING-NEW twice on \( \{\text{TotalCount}(v' - 1,q_{out} - q)\} \), each time creating a new single count \( \{\text{Count}(q/2)\} \), and in the end we get back \( \{\text{TotalCount}(v' + 1,q_{out})\} \). Note that we always satisfy the side condition of COUNTING-NEW because \( q_{out} \leq 1 \).

Finally, after the access, we get back the access token \([\tau]_q\) and two single counts:

\[
\nabla Q = \nabla \left( \{\text{Count}(q/2)\} * \{\text{Count}(q/2)\} \right)
\]

Since the single counts are unsynchronized ghost state, we use ACQMOD-GHOST to get \( \{\text{Count}(q/2)\} * \{\text{Count}(q/2)\} \). Now we can split the token \([\tau]_q\) and the fractional ownership \( a.data\_\Delta v \) into two halves and gain two ARC\(^\gamma\)(a, \( v, \tau, I \))’s.

\( \square \)

17.2.5 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 ARC\(^\gamma\)(a, \( v, \tau, I \)) to the invariant. This includes the fractional ownership \( a.data\_\Delta v \), the access token \([\tau]_q\), and the single count element \( \{\text{Count}(q)\} \). The former two will be stored in the invariant 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 invariant that the local fraction and

\( \square \)
the fractions stored in the invariant 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 \([\tau]_q\) to access the invariant and simultaneously use the token to establish the interpretation of the invariant. This is supported in the rule GPS-CINV-FAA-SREL. The difference with GPS-CINV-FAA-RLX is that in the premise we can additionally use \([\tau]_q\) to reestablish \(I(v + n)\). Consequently, we would not regain \([\tau]_q\) in the postcondition of the rule. Note that this rule is only sound for a release FAA, and thus we can use our local resource \(P\) without using a release fence.

Now, at the release FAA of drop, using \([\tau]_q\), we invoke GPS-CINV-FAA-SREL with the following \(P\) and \(Q\).

\[
P := a.\text{data} \downarrow_{\mathsf{v}} v \star \left[\begin{array}{c}
\text{TotalCount}\\+
\end{array}\right]_q
\]

\[
Q(v') := \begin{cases}
\text{True} & v' \neq 1 \\
a.\text{data} \mapsto v \star [\tau]_q & v' = 1
\end{cases}
\]

We then have to prove that \([\tau]_q \star P \star I^{\tau,v}(v') \Rightarrow I^{\tau,v}(1) \star Q(v')\) where \(v'\) is the old value of a.counter. Similarly to the reasoning in clone, with \([\text{Count}(q)]_v\) from \(P\), we know that \(v' > 0\) and the invariant has some fractional permissions \([\tau]_{q_{\text{in}}}\) and \(a.\text{data} \uparrow_{\mathsf{v}} v\) for some \(q_{\text{in}}\) (see ARC-INV).

Now, if this is not the last drop i.e., \(v' = 1 > 0\), we need to re-establish \(I^{\tau,v}(v' - 1)\) with some new fractions \(q'_{\text{in/out}}\). We pick them as follows:

\[
\begin{align*}
q'_{\text{in}} & := q_{\text{in}} + q \\
q'_{\text{out}} & := q_{\text{out}} - q
\end{align*}
\]

From \([\tau]_q \star P \star I^{\tau,v}(v')\), we can easily get \([\tau]_{q'_{\text{in}}} = [\tau]_{q_{\text{in}}} \star [\tau]_q\) and \(a.\text{data} \uparrow_{\mathsf{v}} v = a.\text{data} \uparrow_{\mathsf{v}} v \star a.\text{data} \downarrow_{\mathsf{v}} v\), which are needed for \(I^{\tau,v}(v' - 1)\). Our remaining work is to transform \([\text{TotalCount}(v',q_{\text{out}})]_v \star [\text{Count}(q)]_v\) to \([\text{TotalCount}(v' - 1,q_{\text{out}})]_v\). Fortunately, this is but a simple application of COUNTING-DROP. Then we are done because \(v' \neq 1\).

In the case where this is the last drop’s FAA, we have \(v' = 1\) and we must prove \([\tau]_1 \star P \star I^{\tau,v}(1) \Rightarrow I^{\tau,v}(0) \star Q(1)\). From \(I^{\tau,v}(1)\) we have \([\text{TotalCount}(1,q_{\text{out}})]_v\) and from \(P\) we have \([\text{Count}(q)]_v\). By an application of COUNTING-DROP, we have \([\text{TotalCount}(0,0)]_v\), which is exactly \(I^{\tau,v}(0)\), and additionally the fact that \(q_{\text{out}} = q\). From \(I^{\tau,v}(1)\) we also know that \(q_{\text{in}} + q_{\text{out}} = q_{\text{in}} + q = 1\). Thus combining what we have left from our assumption \([\tau]_q \star P \star I^{\tau,v}(1)\), we have \(Q(1) = a.\text{data} \mapsto v \star [\tau]_1\). So we finish the last drop’s FAA and gain \(\nabla Q(1)\).

As the return value is \(v' = 1\), we perform an acquire fence (see the code of drop in Figure 17.1). Thanks to the acquire fence rule HOARE-ACQ-FENCE, we remove the modality and regain \(Q(1) = a.\text{data} \mapsto v \star [\tau]_1\). We are almost done: We only need to get back the points-to ownership of a.counter. For this we cancel the invariant for a.counter using the cancellation rule GPS-CINV-CANCEL. The rule requires the full token \([\tau]_1\), which we do have, to ensure that the cancellation happens after all accesses to the invariant. At long last, after the cancellation we now have the full ownership of both fields and can safely use NA-DEALLOC to free them.
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 RB₁₄. Its most interesting APIs are given in Figure 17.4. 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 (Figure 17.4). 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.

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 invariants are in general unsound for RMM, we need to use separate GPS protocols for each counter and at the same time maintain their relation. This is a known challenge, as has been observed by GPS. 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.

**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 pointer can be used again.

---

**Figure 17.4**: An excerpt of Rust’s Arc<T> and Weak<T> APIs
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;
    }
}

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 (Figure 17.5). 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 type &mut T. 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.

5The 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.
release the lock on the \texttt{Weak} counter with a release write of value 1.

\textbf{Correctness} The two checks by \texttt{is\_unique} ensure the synchronization guarantee for temporary reclamation. The second check ensures that the thread is synchronized with all other \texttt{Arc::drop} calls. This means that it is synchronized with all accesses to the content made by all other \texttt{Arc} pointers. The thread, of course, must have synchronized with all accesses made by the current \texttt{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 \texttt{read}, instead of a \texttt{CAS}. If it were a \texttt{CAS}, then we are guaranteed to read the latest value of the \texttt{Arc} counter, and thus synchronizing with all other \texttt{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 \texttt{Arc} counter in Figure 17.6, where our call to \texttt{get\_mut} was initiated somewhere before the latest write 1(c) to the counter. Since we do not know exactly when \texttt{get\_mut} was initiated, the second check by \texttt{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 \texttt{Arc::drop}'s or \texttt{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 \texttt{Arc} counter from a stale 1 to 2, denoted as \texttt{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 \texttt{Arc::clone} or from a \texttt{Weak::upgrade}. If the update is from an \texttt{Arc::clone}, like in 1(a), the thread must have observed it because that update must have been performed by some \texttt{Arc} pointer—unique at that time—of which the current \texttt{Arc} pointer (which this thread owns) is a \textit{descendant}.

The remaining case is when the update is from a \texttt{Weak::upgrade}, like in 1(b). By the first check the thread is synchronized with all \texttt{Weak::drop}'s by all \texttt{Weak} pointers. Note that \texttt{Weak::drop}, similar to Core \texttt{Arc::drop} (Figure 17.1), does a release \texttt{FAA} to decrement the \texttt{Weak} counter. However, unlike in Core \texttt{Arc}, the last \texttt{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.

If an update stale(1 ⇝ 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 ⇝ 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 ⇝ 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 raw cancellable invariants, we use the ghost state of form $\circ q (\{q, O\})^\gamma$ 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 ⇝ 2) update event on the Arc counter. The iRC11 logic therefore must additionally provide unique identifiers for update events. In the implementation of the logic we simply expose the timestamp of an update/write event as its identifier in the protocol assertion. Using the timestamps as identifiers, we thus tie the logical ghost state with the write events through the protocol assertion, making the observations actually physical and therefore can only be transferred with physical synchronization.

In the verification of Arc, we use two different constructions: one, $\circ q (\{q, O_u\})^\mu$, to track the observations coming from Weak::upgrade, and another, $\circ q (\{q, O_c\})^\gamma$, to track those coming from Arc::clone. The former construction $\circ q (\{q, O_u\})^\mu$ enjoys similar properties to that of raw cancellable invariants. That is, the observations can be joined (using set union), and if we own the full fraction $\circ q (\{q, O_u\})^\mu$, then we are guaranteed that $O_u$ contains all possible Weak::upgrade’s stale(1 ⇝ 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 ⇝ 2) event.

The latter construction $\circ q (\{q, O_c\})^\gamma$ is a bit different. Even if we only own a fraction $\circ q (\{q, O_c\})^\gamma$, we need to know that $O_c$ contains all possible Arc::clone’s stale(1 ⇝ 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 $\circ q (\{q, O_c\})^\gamma$. This reflects the fact that any Arc pointer must have seen all Arc::clone’s 1 ⇝ 2 updates, and that any Arc::clone’s 1 ⇝ 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 $\circ q (\{q, O_c\})^\gamma$ for some $q$ (the same $q$ in $\tau$ and a.data $\delta b$, see ARC-MODEL) 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 \( \frac{\sigma q, O_u}{\mu} \) for some \( q \) and \( O_u \) (because only `Weak` pointers can do `Weak::upgrade`). We further require that `Arc::drop` also releases the fraction \( \frac{\sigma q, O_u}{\mu} \) like releasing the other fractions, and similarly that `Weak::drop` releases \( \frac{\sigma q, O_u}{\mu} \).

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 \( \Rightarrow \) 2) updates. First, when acquiring the “lock” on the `Weak` counter, the thread also acquire the full fraction \( \frac{\sigma 1, O_1}{\mu} \) from the `Weak` counter protocol. The full fraction is available in the protocol because all `Weak` pointers have been dropped. With \( \frac{\sigma 1, O_1}{\mu} \), the thread is guaranteed to have seen all `Weak::upgrade`’s stale(1 \( \Rightarrow \) 2) updates. Second, since the thread owns an `Arc` pointer, it owns a fraction \( \frac{\sigma q, O_2}{\gamma} \), which guarantees that the thread has seen all `Arc::clone`’s stale(1 \( \Rightarrow \) 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 \( T \).

\[ \square \]

17.4 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, Figure 17.5), 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 a `∇` modality). 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`:

```rust
let mut arc1 = Arc::new(0);
let arc2 = Arc::clone(&arc1);
thread::spawn(move || {
    let _ : u32 = *arc2; /* drop(arc2); */
});
loop {
    match Arc::get_mut(&mut arc1) {
        None => {}
        Some(m) => { *m = 1; return; }
    }
}
```

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 happen after the child thread finishes and \texttt{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 \texttt{arc2} goes out of scope, it will be destructed by \texttt{Arc::drop}, which uses a release (\texttt{rel}) RMW (see the code at line 16, Figure 17.5). This release RMW will be read by \texttt{get_mut} (line 4, Figure 17.5) in the parent thread (line 7). If this read had been \texttt{acq}, then there would have been a release-acquire synchronization between the release RMW of \texttt{drop} and the acquire read of \texttt{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 \texttt{rlx}, thus no happen-before relationship can be established between the two non-atomic operations.
Part III

COMPASS
Part III presents the Compass specification framework. Chapter 18 starts by reviewing specifications with logical atomicity in both SC and RMC settings. Chapter 19 discusses how to encode Yacovet specs in iRC11 with logical atomicity. Chapter 20 present the proofs of RMC stacks and queues against the Compass specs, relying on general multi-location invariants (Chapter 10) and atomic points-to (Chapter 9). Chapter 21 discusses helping with logical atomicity, and its role in the specs of exchangers. Chapter 22 discusses the composition of the stack spec and the exchanger spec to verify the elimination stack. Chapter 23 discusses the flexibility of the specs, and the relations among them. The top of Figure 1.1 presents the dependency between these chapters and with previous chapters.
18

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 18.1), and in Chapter 19 we will present several of our logically atomic specs in the RC11 model. We review, in §18.1, the traditional Hoare-triple-based specs for sequential queues; in §18.2, logical atomicity\(^1\) and its uses to give strong specs for concurrent SC queues; and in §18.3, how Cosmo\(^2\) extends those specs for RMC with thread views.

18.1 Sequential Specifications for Queues

The separation logic sequential specs for queues are given as \texttt{SEQ-ENQ} and \texttt{SEQ-DEQ} (Figure 18.1). \texttt{SEQ-ENQ} specifies that an enqueue function call \texttt{enq([q, v])} can run safely as long as it has \texttt{Queue(q, vs)}, an abstract separation logic assertion that represents full ownership of the queue object \textit{q} (an instance of the data structure). An implementation can define \texttt{Queue(q, vs)} as arbitrary resources that it specifically needs. But from the perspective of clients, \texttt{Queue(q, vs)} is abstract because it asserts that \textit{q}’s current state can be seen abstractly as a list of values \textit{vs}—that is, the queue’s elements are currently \textit{vs}, ordered by the list order.

\texttt{SEQ-ENQ} then says that \texttt{enq([q, v])} requires and consumes \textit{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 \texttt{vs ++ [v]}, reflecting the operation’s effects: \textit{v} has been enqueued to the end of \textit{q}. Conversely, by \texttt{SEQ-DEQ}, a dequeue \texttt{deq([q])} also consumes \textit{q}’s ownership and, if the queue is not empty, returns the head value \textit{v} of \textit{vs} and gives back the ownership with only its tail \textit{vs’}. (The notation \{ \textit{v}. \texttt{Q} \} denotes the postcondition with a returned value \textit{v}.) Otherwise, if \textit{q} is empty, \texttt{deq([q])} returns empty (\texttt{[]} \texttt{)} and the fact that the abstract state is also empty (\texttt{[]} \texttt{).}

That an operation is allowed to consume the queue ownership for

\(^1\)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].
the whole duration of its execution is what makes the specs sequential: a group of threads cannot access the ownership \(\text{Queue}(q, vs)\) concurrently in order to perform concurrent enqueues and/or dequeues. To have strong specs for such fine-grained concurrency, we need logical atomicity.

18.2 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 \(Q\).

Using LATs, we can give strong specs like SC-ENQ and SC-DEQ (Figure 18.1) 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 \(\langle vs. \text{Queue}(q, vs) \rangle\): 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 postcondition. For example, SC-ENQ says that \(\text{enq}([q, v])\) can withstand arbitrary concurrent updates to the state \(vs\) of \(q\), up until the commit point when it atomically transforms \(\text{Queue}(q, vs)\) (where \(vs\) is the state at that instant) to the new state \(\text{Queue}(q, vs + + [v])\). 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 \(\text{Queue}(q, vs)\) for an arbitrary but unchanging \(vs\), thereby prohibiting concurrent interference.

Last but not least, we add a local precondition \(\text{isQueue}(q)\), 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 \(\text{Queue}(q, vs)\) which is

\(^3\)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].
SEQ-ENQ
\{\text{Queue}(q, vs)\} \text{enq}(\langle q, v \rangle) \{() . \text{Queue}(q, vs \leftrightarrow [v])\}

SEQ-DEQ
\{\text{Queue}(q, vs)\} \text{deq}(\langle q \rangle) \{v . (vs = \langle v = \epsilon \ast \text{Queue}(q, []) \rangle) \lor (\exists vs'. vs = v \leftrightarrow vs' \ast \text{Queue}(q, vs'))\}

SC-ENQ
\text{isQueue}(q) \vdash (\langle vs . \text{Queue}(q, vs)\rangle \text{enq}(\langle q, v \rangle) \{() . \text{Queue}(q, vs \leftrightarrow [v])\})

SC-DEQ
\text{isQueue}(q) \vdash (\langle vs . \text{Queue}(q, vs)\rangle \text{deq}(\langle q \rangle) \{v . (vs = \langle v = \epsilon \ast \text{Queue}(q, []) \rangle) \lor (\exists vs'. vs = v \leftrightarrow vs' \ast \text{Queue}(q, vs'))\})

ABS-SC-ENQ
\text{isQueue}(q) \ast \mathcal{V} \vdash (\langle vs . \text{Queue}(q, vs)\rangle \text{enq}(\langle q, v \rangle) \{() . \text{Queue}(q, vs \leftrightarrow [(v, V)])\})

ABS-SC-DEQ
\text{isQueue}(q) \ast \mathcal{V} \vdash (\langle vs . \text{Queue}(q, vs)\rangle \text{deq}(\langle q \rangle) \{v . (v = \epsilon \ast \text{Queue}(q, vs)) \lor (\exists vs', V' . vs = (v, V') \leftrightarrow vs' \ast \text{Queue}(q, vs') \ast \mathcal{V}'\})

neither duplicable nor local.

Intuitively, it should be clear that \(\langle P \rangle e \langle Q \rangle^E\) is a stronger spec than \(\{P\} e \{Q\}_E\), 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 \(\text{Queue}(q, vs)\)? 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 \text{Hoare-Inv} given in Figure 5.6: 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 \text{LAT-Inv strengthens Hoare-Inv}, as it relaxes the restriction of “accessing around atomic instructions” (\(\text{atomic}(e)\)) to “accessing around logically atomic expressions”.

\[
\text{LAT-INV} \quad \frac{\langle I \ast P \rangle e \langle I \ast Q \rangle^{E \land N}}{N \subseteq E} \quad \text{with this rule, clients can build protocols to use and combine libraries with LAT specs. For example, we can allocate an invariant}
\]

\[
\exists vs_1, vs_2. \text{Queue}(q_1, vs_1) \ast \text{Queue}(q_2, vs_2) \ast R(vs_1, vs_2)^N
\]

that ties together two queues by a relation \(R\), and then use \text{LAT-Inv} with \text{SC-ENQ} and \text{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.

Figure 18.1: Specifications of Queue operations, from sequential, to SC concurrency and strong RMC.
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. 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.

18.3 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.

18.3.1 Cosmo Specs for Queues

\texttt{ABS-So-ENQ} and \texttt{ABS-So-DEQ} (in Figure 18.1) 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 18.1): (1) the specs take the “seen view” assertion \( \exists V \) 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 \texttt{ABS-So-ENQ}, an enqueue releases its local view \( V \) at its commit point, and by \texttt{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 LAT\textsubscript{abs} style. (The complete Cosmo specs also track so among enqueues and among dequeues.)

18.3.2 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 \texttt{ABS-So-DEQ}, a failing empty dequeue is a read-only operation, and the LAT\textsubscript{abs} specs do not give us any new facts about vs. This is
weaker than in the SC model, where SC-DEQ says that dequeues fail with \( \epsilon \) 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 \( \pi \) sees the queue as empty and returns \( \epsilon \), but the queue is in fact not empty, because a fresh enqueue by another thread \( \rho \) has not become visible to \( \pi \) 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 18.2. 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 (§19), 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 18.2: 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.

```plaintext
enq([q, 41]); repeat (\text{acq} flag \neq 0); enq([q, 42]); deq([q]); deq([q]); flag := rel 1 // return 41 or 42, not empty
```
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 §19.1 we present the \texttt{LAT$_{\text{abs}}$} style which generalizes the \texttt{LAT$_{\text{so}}$} style, and its instance for queues, which suffices to verify the MP client in Figure 18.2. In §19.2, we present the \texttt{LAT$_{\text{hb}}$} spec style, a weakening of \texttt{LAT$_{\text{abs}}$}.

19.1 Graph-Based Specs to Encode Partial Orders

The \texttt{LAT$_{\text{hb}}$} style extends the \texttt{LAT$_{\text{so}}$} style by exposing a greater part of \texttt{hb}. An instance for queues is given in \texttt{ABS-HB-ENQ}, \texttt{ABS-HB-DEQ}, and \texttt{ABS-HB-QUEUE-CONSISTENCY} (Figure 19.1). 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 \texttt{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 \texttt{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 $G \in \text{Graph}$ is a general construction inspired by the declarative specs of Yacovet. 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 Figure 19.1. A graph $G$ is a pair of (1) a function that maps each event id...
\[ V \in \text{View} ::= \text{Loc} \rightarrow \text{Time} \]
\[ e \in \text{EventId} ::= \mathbb{N} \]
\[ \text{QueueEvent} ::= \text{Enq}(v) \mid \text{Deq}(v) \mid \text{Deq}(e) \]
\[ M \in \text{LogView} ::= \varphi(\text{EventId}) \]
\[ \text{Event} ::= \text{QueueEvent} \times \text{View} \times \text{LogView} \]
\[ G \in \text{Graph} ::= (\text{EventId} \rightarrow \text{Event}, \varphi(\text{EventId} \times \text{EventId})) \]

QueueConsistent\((vs, G)\) :=
\[
\begin{align*}
\forall (e, d) \in G. so. \exists v. G(e). type &= \text{Enq}(v) \land G(d). type = \text{Deq}(v) \land \ldots \\
\text{Queue-Matches} \\
\forall (e, d) \in G. so. e'. G(e'). type &= \text{Enq}(\_ ) \rightarrow (e', e) \in G. lhb \rightarrow \\
\exists d'. (e', d') \in G. so \land (d, d') \notin G. lhb \\
\text{Queue-FIFO} \\
\forall d, e. G(d). type &= \text{Deq}(e) \rightarrow G(e). type = \text{Enq}(\_ ) \rightarrow \\
(e, \_ ) \notin G. so \rightarrow (e, d) \notin G. lhb \\
\text{Queue-EMPDeq} \\
\ldots
\end{align*}
\]

\text{ABS-Hb-Queue-Consistency}

\[ \text{Queue}(q, vs, G) \vdash \text{QueueConsistent}(vs, G) \]

\textbf{ABS-Hb-Enq}

\[ \text{SeenQueue}(q, G_0, M_0) * \exists V \vdash \]
\[ (G, vs. \text{Queue}(q, vs, G)) \]
\[ \text{enq}([q, v]) \]
\[
\begin{align*}
(\_ ) \exists G' \supseteq G, M' \supseteq M_0, V' \supseteq V. \text{Queue}(q, vs \leftrightarrow [(v, V')], G') \\
* \text{SeenQueue}(q, G', M') * \exists V' * \exists e \notin G. e \in M' \land G' = G[e \mapsto (\text{Enq}(v), V', M')] \\
\end{align*}
\]

\textbf{ABS-Hb-Deq}

\[ \text{SeenQueue}(q, G_0, M_0) * \exists V \vdash \]
\[ (G, vs. \text{Queue}(q, vs, G)) \]
\[ \text{deq}([q]) \]
\[
\begin{align*}
v. \exists vs', G' \supseteq G, M' \supseteq M_0, V' \supseteq V. \text{Queue}(q, vs', G') * \text{SeenQueue}(q, G', M') * \exists V' \\
* \forall \]
\[
\begin{align*}
(v = e \land vs' = vs \land \exists d \notin G. d \in M' \land G' = G[d \mapsto (\text{Deq}(e), V', M')]) \\
* \forall \exists v, vs = (v, V_e) : vs' \land \exists e, M_e, d \notin G. G(e) = (\text{Enq}(v), V_e, M_e) \land (e, \_ ) \notin G. so \land V_e \subseteq V' \\
\land M_e \cup \{e, d\} \subseteq M' \land G' = G[d \mapsto (\text{Deq}(v), V', M')] \land G'. so = \{(e, d) \cup G. so\}
\end{align*}
\]

\text{Figure 19.1: Compass Specs for Queues}
\( e \in \text{EventId} \) to event data of type \( \text{Event} \), and (2) a set of event \( \text{id} \) pairs that encodes the \( \text{so} \) relation. We use \( G(e) \) to denote the event data for \( e \) in \( G \), and \( G.\text{so} \) to denote the \( \text{so} \) relation of \( G \).

The type \( \text{Event} \) is a tuple of (1) an event type (\( \text{type} \)), (2) a physical view (\( \text{view} \)), and (3) a logical view (\( \logview \)). In Figure 19.1 we give an instance of the event type for queues: the events can be an \( \text{enqueue} \) event of \( v \) (\( \text{Enq}(v) \)), a successful \( \text{dequeue} \) event of \( v \) (\( \text{Deq}(v) \)), or a failing (empty) \( \text{dequeue} \) event (\( \text{Deq}(e) \)). 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., \( e \in G(d).\logview \), 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 \( \text{hb} \) relation between library operations, just as the physical view construction is an approximation of \( \text{hb} \) between memory instructions. The difference is that while physical views approximate \( \text{hb} \) globally between memory instructions, logical views only approximate \( \text{hb} \) locally for the library in question. As such, our logical views correspond to the local happens-before \( \text{lh} \) relation of a library object introduced by Yacovet. Henceforth we use \( e \in G(d).\logview \) and \((e, d) \in G.\text{lh} \) interchangeably.

The \( \text{LAT}_{\text{rb}}^{\text{abs}} \) style extends \( \text{LAT}_{\text{rb}}^{\text{abs}} \) 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 \text{ABS-HB-ENQ}, when an \( \text{enqueue} \) of \( v \) commits, the current graph \( G \) of \( q \) is extended atomically with a fresh event \( e \) whose type is \( \text{Enq}(v) \), into \( G' \): \( G \subseteq G' \).

**Local assertions for logical views.** The partial orders are also extended at \( e \)’s commit point to relate it to other operations. In \text{ABS-HB-ENQ}, \( G'.\text{lh} \) extends \( G.\text{lh} \) by setting \( G'(e).\logview = M' \), 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 \( \text{enq} \) call. This tracking of thread-local logical views is done by a new persistent assertion \( \text{SeenQueue}(q, G_0, M_0) \), where \( G_0 \) is a snapshot of the current \( G \) (\( G_0 \subseteq G \)), 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 \( \text{SeenQueue}(q, G', M') \) with the latest snapshot \( G' \) and a new logical view \( M' \), reflecting that the thread has synchronized with more operations (\( M_0 \subseteq M' \)), including the operation \( e \) that it has just executed (\( e \in M' \)). By taking \( \text{SeenQueue} \) as a local precondition, 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 \text{ABS-HB-DEQ} to verify the MP client in Figure 18.2.
Compared to the $\text{LAT}^\text{abs}_{\text{hb}}$ style, in $\text{LAT}^\text{abs}_{\text{hb}}$ each library type has a local logical view assertion like $\text{SeenQueue}$ 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 $\text{isQueue}(q)$ assertion in $\text{ABS-SO-ENQ}$. The logical view assertion plays the same role for logical views as the “seen view” assertion $\exists V$ 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 $\text{LAT}^\text{abs}_{\text{hb}}$ 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 $\text{QueueConsistent}$, the consistency conditions for the queue library type, is given at the bottom right of Figure 19.1. It requires, among other things, that enqueues and dequeues must follow the first-in-first-out principle (FIFO, $\text{QUEUE-FIFO}$), stated in a fashion that is not too strong for RMC (more about that below). The fact that $\text{QueueConsistent}$ is maintained by all operations is encoded in $\text{ABS-HB-QUEUE-CONSISTENCY}$: the queue ownership assertion $\text{Queue}(q, \text{vs}, G)$, which is consumed and reproduced around the commit point, always implies consistency. So when $\text{ABS-HB-ENQ}$ and $\text{ABS-HB-DEQ}$ extend $(\text{vs}, G)$ to new state $(\text{vs}', G')$, the operations can assume $\text{QueueConsistent}(\text{vs}', G')$ and must then re-establish $\text{QueueConsistent}(\text{vs}', G')$.

More specifically, if $\text{deq}$ succeeds with a value $v$, $\text{ABS-HB-DEQ}$ tells the client that $G'.\text{so}$ extends $G.\text{so}$ 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 $\text{ABS-HB-QUEUE-CONSISTENCY}$, the spec additionally says that $(e, d)$ satisfies, among other things, $2$ (1) $\text{QUEUE-MATCHES}$: the return value $v$ of the dequeue $d$ must match the value enqueued by $e$; and (2) $\text{QUEUE-FIFO}$: if there is another enqueue event $e'$ that happens before $e$, then $e'$ must already have been dequeued by some $d' ((e', d') \in G.\text{so})$, and our $d$ cannot happen before $d' ((d, d') \notin G.\text{hb})$. (The consistency conditions on enqueue events are elided, so we will not discuss them.)

$^2$For example, an element can only be dequeued once.

$^3$Herlihy and Wing, “Linearizability: A Correctness Condition for Concurrent Objects” [HW90].

**WEAKER BUT FLEXIBLE.** The $\text{QUEUE-FIFO}$ condition appears weaker than what one might expect, i.e., $(d', d) \in G.\text{hb}$, but such a condition only works for strongly synchronized (e.g., SC) implementations. As stated, $\text{QUEUE-FIFO}$ is also satisfiable by implementations that have little synchronization between dequeues. In fact, we have verified that $\text{QUEUE-FIFO}$ 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 $\text{hb}$ 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.)

Nonetheless, $\text{QUEUE-FIFO}$ is still flexible enough that, for example, if a client decides to use the queue in an SC fashion by adding suffi-
Invariant: \( \exists v, G. \text{Queue}(q, \emptyset, G) * \text{deqPerm}(\text{size}(G.so)) * \text{size}(G.so) \leq 2 * \ldots \)

\[
\begin{align*}
\{\text{SeenQueue}(q, \emptyset, 0) * \emptyset V_1\} & \quad \{\text{SeenQueue}(q, \emptyset, 0) * \text{deqPerm}(1) * \emptyset V_3\} \\
\langle \text{Queue}(q,_) * \ldots \rangle & \quad \langle \text{Queue}(q,_) * \ldots \rangle \\
\langle \text{enq}([q, 41]) \rangle & \quad \langle \text{deq}([q]) \rangle \\
\langle \text{Queue}(q,_,0) * \ldots \rangle & \quad \langle \text{deq}([q]) \rangle \\
\langle \text{Queue}(q,_,0) * \ldots \rangle & \quad \langle \text{Queue}(q,_,0) * \ldots \rangle \\
\langle \text{enq}([q, 42]) \rangle & \quad \{\text{Seequeue}(q, G_1, \{e_1\}) * \ldots \} \\
\{\text{Queue}(q,_,0) * \ldots \} & \quad \{v, \text{Queue}(q, G_3, \{e_1, e_2, e_2\}) * \} \\
\{v, \text{Queue}(q, G_3, \{e_1, e_2, d_2\}) * \} & \quad \{\text{Queue}(q, G_1, \{e_1\}) * \ldots \} \\
\text{flag} := \text{rel} 1 & \quad \{\text{Queue}(q, G_2, \{e_1, e_2\}) * \ldots \}
\end{align*}
\]

Figure 19.2: A proof sketch of Message Passing with queues

By efficient external synchronization, the client can know that \( \text{lhb} \) is total, i.e., \((d', d) \in G.\text{lhb} \lor (d, d') \in G.\text{lhb} \), and can thus exclude the right-hand side of the disjunction and regain the stronger FIFO condition with \((d', d) \in G.\text{lhb} \). 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 \( \text{deq} \) returns empty \((e) \), consistency demands that the added empty dequeue event \( d \) satisfies \( \text{Queue-EmpDeq} \), which is sufficient to verify the MP client (Figure 18.2). Intuitively, \( \text{Queue-EmpDeq} \) 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 \( \text{Queue-EmpDeq} \) 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 19.2. Following the pattern mentioned at the end of §18.2, we put the ownership \( \text{Queue}(q,_,) \) in an invariant to enforce a concurrent protocol on the queue, using a *dequeue permission* called \( \text{deqPerm} \) that can be defined with Iris ghost state. One dequeue permission \( \text{deqPerm}(1) \) is needed to perform one successful dequeue. This requirement can be seen in the invariant: \( \text{deqPerm}(\text{size}(G.so)) \) counts the number of successful dequeues, and a successful dequeue will extend \( G.so \) by 1, so anyone who successfully dequeues needs to put in a \( \text{deqPerm}(1) \) to re-establish the invariant. For our particular example, we also implement \( \text{deqPerm} \) such
that there are only two \( \text{deqPerm}(1) \)'s (i.e., \( \text{deqPerm}(2) \)) 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 \( \text{SeenQueue}(q, \emptyset, \emptyset) \) of the initial empty state.

The verification of the left-most thread is straightforward: for each enqueue, we use \( \text{LAT-INV} \) to open the invariant and then use \( \text{ABS-Hb-ENQ} \). Afterwards the thread has two enqueue events \( \{e_1, e_2\} \) in its logical view, and the write to flag releases \( \text{SeenQueue}(q, G_1, \{e_1, e_2\}) \) to the right-most thread. The verification of the middle thread uses \( \text{LAT-INV} \) and \( \text{ABS-Hb-DEQ} \), and if the dequeue succeeds, \( \text{deqPerm}(1) \) 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 \( \text{SeenQueue}(q, G_1, \{e_1, e_2\}) \) from the left-most thread. We then use \( \text{LAT-INV} \) and \( \text{ABS-Hb-DEQ} \) 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 \( \text{deqPerm}(1) \) in the invariant and finish. If \( d_3 \) is an empty dequeue, we derive a contradiction. As there are only two \( \text{deqPerm}(1) \) 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: \( \text{size}(G, \text{so}) \leq 1 \). 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 \( \text{SeenQueue}(q, G_1, \{e_1, e_2\}) \), both \( e_1 \) and \( e_2 \) happen before \( d_3 \). By \( \text{QUEUE-EMPDEQ} \), we have our contradiction.

\[ \square \]

### 19.2 Weaker Specs by Abandoning Abstract States

The \( \text{LAT}_{\text{abs}} \) 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 \( \text{LAT}_{\text{abs}} \) specs for queues (and therefore transitively the \( \text{LAT}_{\text{so}} \) 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 \( v_s \) in the queue.

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 \( \text{LAT} \) 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 \( \text{LAT}_{\text{hb}} \) specs, a weakening of the \( \text{LAT}_{\text{abs}} \) specs where the abstract state is abandoned. In particular, our instance of the \( \text{LAT}_{\text{hb}} \) specs for queues is exactly the specs \( \text{ABS-Hb-ENQ} \) and \( \text{ABS-Hb-DEQ} \) (Figure 19.1) without \( v_s \).

\( \text{LAT}_{\text{hb}} \) specs may appear weak, but they can still take advantage of

---

\(^4\)Michael and Scott, “Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms” [MS96].

\(^5\)Jung et al., “The future is ours: prophecy variables in separation logic” [Jun+20].
external synchronization information, \textit{i.e.}, the argument in §19.1 about flexibility of the partial orders still applies. Practically, they are sufficient to verify the MP client in Figure 18.2.
Verifications of Stacks and Queues
21

Helping in Exchangers Specifications
22

Verifications of Exchangers and the Elimination Stack
Discussions on Specifications
Conclusion

And so it has ended.
Bibliography


[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, 125).


240 Bibliography


Versicherung an Eides Statt

Ich versichere an Eides statt, dass ich die eingereichte Dissertation selbstständig und ohne unzulässige fremde Hilfe verfasst, andere als die in ihr angegebene Literatur nicht benutzt und dass ich alle ganz oder annähernd übernommenen Textstellen sowie verwendete Grafiken, Tabellen und Auswertungsprogramme kenntlich gemacht habe.

Außerdem versichere ich, dass die vorgelegte elektronische mit der schriftlichen Version der Dissertation übereinstimmt und die Abhandlung in dieser oder ähnlicher Form noch nicht anderweitig als Promotionsleistung vorgelegt und bewertet wurde.

Datum Unterschrift