Shaking Up the Foundations of Modern Separation Logic

Dissertation zur Erlangung des Grades des
Doktors der Ingenieurwissenschaften
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

vorgelegt von
Simon Spies

Saarbrücken, 2025

Tag des Kolloquiums

16. Mai 2025

Abstract

The problem of how to scalably verify large, stateful programs is one of the oldest—and still unsolved—challenges of computer science. Over the last two decades, there has been considerable progress toward this goal with the advent of *separation logic*, a verification technique for modularly reasoning about stateful programs. While originally only developed for imperative, pointer-manipulating programs, separation logic has in its modern form become an essential tool in the toolbox of the working semanticist for modeling programming languages and verifying programs.

This dissertation presents a line of work that revisits the foundations of modern separation logic in the context of the separation logic framework *Iris*. It targets two broader areas: *step-indexing* and *automation*. Step-indexing is a powerful technique for modeling many of the advanced, cyclic features of modern languages. Here, **Transfinite Iris** shows how to generalize step-indexing from proving safety properties to proving liveness properties, and **Later Credits** enable more flexible proof patterns for step-indexing based on separation logic resources. Automation is important for reducing the overhead of verification to scale to larger code bases. Here, **Quiver** introduces a new form of guided specification inference to reduce the specification overhead of separation logic verification, and **Daenerys** develops new resources in Iris that lay the groundwork for automating parts of Iris proofs using SMT solvers.

Zusammenfassung

Wie man skalierbar große Programme verifiziert ist eine der ältesten, ungeklärten Fragen der Informatik. In den letzten zwei Jahrzehnten wurde hier deutlicher Fortschritt erzielt mit der Einführung von Separationslogik, einer Technik für modulare Programmverifikation. Separationslogik wurde ursprünglich nur für imperative Programme mit Zeigern entwickelt, ist aber in ihrer modernen Form essenziell geworden, um Programmiersprachen zu modellieren und Programme zu verifizieren.

Diese Dissertation präsentiert eine Reihe von Arbeiten, die sich im Kontext der Separationslogik *Iris* mit den Grundlagen moderner Separationslogik beschäftigen. Sie fokussiert sich auf zwei Bereiche: *Step-Indexing* und *Automatisierung*. Step-Indexing ist eine wichtige Modellierungstechnik für viele der fortgeschrittenen, zyklischen Funktionen moderner Programmiersprachen. Hier zeigt **Transfinite Iris**, wie man Step-Indexing nicht nur für Sicherheitseigenschaften verwenden kann, sondern auch um Lebendigkeitseigenschaften zu beweisen, und **Later Credits** ermöglichen flexiblere Beweisstrukturen durch die Verwendungen von Separationslogikressourcen. Automatisierung ist wichtig, um den Aufwand von Verifikation zu senken, um größere Programme zu verifizieren. Hier entwickelt **Quiver** eine neue Form der Spezifikationsinferenz, um den Aufwand von Separationslogikspezifikationen zu reduzieren, und **Daenerys** entwickelt neue Iris-Ressourcen, die den Weg bereiten, um Teile von Iris-Beweisen mit SMT-Solvern zu automatisieren.

Acknowledgements

The last few years that I worked toward this dissertation have been some of the most rewarding years of my life. When I started, I did not yet realize that my new *job* would involve traveling the world and making new friends—all while working on interesting research problems to my heart's content. For this experience, I will be eternally grateful, and it would not have been possible without all the wonderful people in my life.

First and foremost, I want to thank my advisor Derek. His Semantics course is what excited me to pursue a PhD in the area of programming languages and verification in the first place. I could not have asked for a better advisor. Derek has many remarkable qualities—except for punctuality perhaps. The one that always impressed me the most is his unparalleled ability to extract high-level ideas from the convoluted, technical explanations of his students. Besides, who else can teach you how to define intricate logical relations and how to communicate research ideas clearly, but can also teach you what "Highland fridge smell" tastes like? I thank you from the bottom of my heart, Derek.

Next, I want to thank Deepak Garg and François Pottier for reviewing this dissertation and Martina Maggio and Aïna Linn Georges for serving on my thesis committee. I also want to thank Gert Smolka for mentoring me throughout my undergraduate studies and for introducing me to mechanized theorem proving with Rocq, and Yannick Forster for giving me valuable advice and guidance over all these years. Moreover, I want to thank the office and IT staff of MPI-SWS for a seamless experience throughout my PhD. I also want to thank Lennard Gäher, Janine Lohse, Niklas Mück, Benjamin Peters, Rose Hoberman, Yannick Forster, and Michael Sammler for proofreading this dissertation and giving me valuable feedback.

In addition, I want to thank all the fantastic people that I have had the honor to collaborate with over the last few years. In particular, I want to thank Lennard Gäher, Michael Sammler, Niklas Mück, Haoyi Zeng, Daniel Gratzer, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Andrea Lattuada, Peter Müller, Lars Birkedal, and Derek Dreyer for collaborating with me on the work presented in this dissertation. I especially want to thank Ralf and Robbert for all the knowledge about Iris that they have shared with me over the years. Without their continued irritation over my desire to change fundamental aspects of Iris, it would have been much less fun to fork Iris for a *third* time. I also want to thank Joseph for suggesting the idea of using transfinite step-indices to remodel Iris, which eventually turned into Transfinite Iris, and for helping me navigate the depths of Iris when I forked it for the *first* time. In addition, I want to thank Google for supporting me with a Google PhD Fellowship and, in particular, Wontae Choi for all of his helpful advice over the years. I also want to thank Jane Street for giving me the opportunity to visit them for an internship.

Moreover, I want to thank the students at MPI and at Saarland University, especially all the students and post-docs who have been part of the group over the years, including Michael Sammler, Lennard Gäher, Kimaya Bedarkar, Laila Elbeheiry, Niklas Mück, Benjamin Peters, Janine Lohse, Jan Menz, Ralf Jung, Hoang-Hai Dang, Jan-Oliver Kaiser, Rodolphe Lepigre, Emanuele D'Osualdo, Youngju Song, Aïna Linn Georges, Vincent Lafeychine, Neven Villani, Irene Yoon, Milijana Surbatovich, Johannes Hostert, and Haoyi Zeng. Thank you for all the inspiring conversations over the years and for sharing your research ideas at "tea times". In particular, I want to thank Michael for teaching me valuable project management skills, for showing me what proper Swiss Fondue and a "Weißwuascht Frühstück" taste like, and for having common sense whenever we crossed the US border; Lennard for exploring the depths of the Slovenian mountains with me and for keeping his faith in the Deutsche Bahn when I had long lost it; Aïna, Andrea, and Benjamin for more coffee trips to the new cafe on campus than I can count; and Niklas for his willingness to fight the SMT solver Z3 for a couple of weeks. I also want to thank Johannes and Haoyi. It was a pleasure to advise you both.

Last but not least, I want to thank my friends and family. It is no secret that every PhD has its ups and downs. You have stood by my side through the highs and the lows, and for that, I cannot thank you enough. In particular, I want to thank Julian and Frederik for countless coffee dates all over Saarbrücken—both during the pandemic and after—and my oldest friend Max for always being there when it mattered. I also want to thank my parents Agnes and Martin and my sister Nora, not only for supporting me all these years, but also for encouraging me to pursue a Bachelor's degree in computer science when I was not yet sure what I wanted to do with my life. Without your encouragement, this dissertation would probably not exist. My deepest gratitude, however, goes to my partner Merve. I do not even know where to start, so I will keep it short: You are the reason that our apartment felt like "home", even when we both knew it would only be temporary. Thank you for being by my side these last few years. I look forward to all the time that is yet to come, be it London or any other place in the world.

Simon Spies
Saarbrücken, January 2025

Contents

Chapter 1

Introduction

Over the last two decades, there has been considerable progress on the verification of large, stateful programs—ranging from low-level systems code in languages like C,[1] Rust,[2] and assembly[3] to code in high-level languages like Java,[4] Python,[5] Scala,[6] Go,[7] and OCaml.[8] One of the central catalysts at the heart of this movement is the development of *separation logic*,[9] a powerful foundation for modular reasoning about programs with shared state.

The key innovation of separation logic over traditional Hoare logic[10] is the idea of *ownership reasoning*: assertions not only state facts about the current program state but also carry permissions to access or modify the state. For example, the hallmark "points-to assertion" $\ell \mapsto v$ of separation logic describes the state of the memory (*i.e.,* location ℓ in the memory currently stores the value v) and, additionally, conveys ownership of this particular piece of memory (*i.e.,* it allows reading from and writing to ℓ). In particular, if a part of a program *owns* a location ℓ, then it can be sure that no other parts of the program interfere with ℓ unless it explicitly gives them permission to do so.

Ownership reasoning makes the verification of large, stateful programs modular: it allows one to decompose the program into smaller parts and then verify each part locally only with respect to the resources that it affects. This is concisely illustrated by the characteristic *frame rule* of separation logic:

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

It means that when we have proven a separation logic triple $\{P\}\ e\ \{Q\}$ with precondition P and postcondition Q, then we can "frame on" any additional resources R, because they are guaranteed to be unchanged by the expression e. More specifically, if a resource R holds *separately* from the precondition P, written $P * R$, then it is not affected by the execution of e and R still holds after e has finished executing. Thus, via framing, separation logic allows one to focus on only the minimal part of the state that is affected by an operation. Everything that is separate stays unchanged by construction.

1.1 Modern Separation Logic

Separation logic was originally conceived as a verification technique for idealized imperative, pointer-manipulating code,[11] where the separating conjunction $P * Q$ separates disjoint pieces of memory (*e.g.,* two memory locations ℓ and r in $\ell \mapsto 42 * r \mapsto 0$). However, it has since far outgrown these roots.

[1] Jacobs et al., "VeriFast: A powerful, sound, predictable, fast verifier for C and Java", 2011 [Jac+11]; Appel, "Verified Software Toolchain", 2012 [App12]; Sammler et al., "RefinedC: Automating the foundational verification of C code with refined ownership types", 2021 [Sam+21].

[2] Astrauskas et al., "Leveraging Rust types for modular specification and verification", 2019 [Ast+19]; Gäher et al., "RefinedRust: A type system for high-assurance verification of Rust programs", 2024 [Gäh+24].

[3] Jensen, Benton, and Kennedy, "High-level separation logic for low-level code", 2013 [JBK13]; Chlipala, "Mostly-automated verification of low-level programs in computational separation logic", 2011 [Chl11].

[4] Parkinson and Bierman, "Separation logic, abstraction and inheritance", 2008 [PB08]; Jacobs et al., "VeriFast: A powerful, sound, predictable, fast verifier for C and Java", 2011 [Jac+11]; Müller, Schwerhoff, and Summers, "Viper: A verification infrastructure for permission-based reasoning", 2017 [MSS17].

[5] Eilers and Müller, "Nagini: A static verifier for Python", 2018 [EM18].

[6] Giarrusso et al., "Scala step-by-step: Soundness for DOT with step-indexed logical relations in Iris", 2020 [Gia+20].

[7] Chajed et al., "Verifying concurrent, crash-safe systems with Perennial", 2019 [Cha+19a]; Wolf et al., "Gobra: Modular specification and verification of Go programs", 2021 [Wol+21].

[8] Mével, Jourdan, and Pottier, "Cosmo: A concurrent separation logic for multicore OCaml", 2020 [MJP20]; Charguéraud, "Characteristic formulae for the verification of imperative programs", 2011 [Cha11].

[9] Reynolds, "Separation Logic: A logic for shared mutable data structures", 2002 [Rey02]; O'Hearn, Reynolds, and Yang, "Local reasoning about programs that alter data structures", 2001 [ORY01].

[10] Hoare, "An axiomatic basis for computer programming", 1969 [Hoa69].

[11] Reynolds, "Separation Logic: A logic for shared mutable data structures", 2002 [Rey02].

Modern formulations of separation logic often support more general resources, more complex programming languages, richer assertion languages, and more application domains. The core that they preserve is the separating conjunction $P * Q$—often for more interesting notions of separation—and its associated ownership reasoning principles such as framing.

Iris. The perhaps most successful example in this regard is the separation logic framework **Iris**[12] (which stands on the shoulders of many separation logic extensions before it, including CaReSL,[13] iCAP,[14] HOCAP,[15] Verifast,[16] CAP,[17] and VST[18]). It improves on the original separation logic[19] in four key aspects:

1. *Resource Algebras.* First, Iris supports a wide variety of resources that go far beyond the original points-to resources $\ell \mapsto v$ of separation logic (*e.g.,* invariants[20] and ghost programs[21]). At its foundation is a general model based on *resource algebras*, which re-interprets the separating conjunction $P * Q$ as resource composition and induces new logical connectives like the frame-preserving update modality $\Rrightarrow P$ and the persistency modality $\square P$.
2. *Step-Indexing.* Second, Iris integrates a technique called *step-indexing*[22] to support advanced features of modern programming languages such as higher-order state (*i.e.,* memory storing functions) and unrestricted recursive types. Step-indexing provides Iris with powerful recursive reasoning principles that go beyond traditional induction or co-induction.
3. *First-Class Weakest Precondition.* Third, Iris introduces a language-agnostic weakest precondition $\mathbf{wp}\ e\ \{Q\}$ as a first-class primitive in the logic. By instantiating it (or modifying its definition), users can derive new program logics for different languages (or application domains).
4. *Rocq Implementation.* Fourth, Iris is implemented inside the Rocq proof assistant and provides an interactive proof mode, the Iris Proof Mode,[23] for mechanizing proofs. The implementation allows others to build on Iris and it ensures trustworthiness of Iris's proofs.

By combining all four aspects, Iris has proven to be a versatile foundation for an unusually large number of use cases. There are, by now, over a hundred projects using and building on Iris, including for fine-grained concurrency [VB21; Car+22], time complexity reasoning [MJP19; Pot+24], automated verification [Sam+21; Gäh+24; MKG22], semantic models of languages [Jun+18a; Gia+20], weak memory models [Kai+17; Dan+20; Ham+24], multi-language verification [Sam+23; Gué+23], session types [HBK22; JHK23], probabilistic reasoning [TH19; Gre+24], relational reasoning [FKB18; Gäh+22; Tim+24a], and many more.

1.2 Contributions

But there are also limits to the reach of Iris. In this dissertation, we shine a light on what could—traditionally—*not* be done with Iris, and we revisit the foundations of Iris in order to lift these limitations.

Concretely, the goal of this dissertation is to make Iris more expressive (*i.e.,* to verify more advanced programs and properties) and to reduce its verification overhead (*i.e.,* to verify larger programs more easily). To this end, the dissertation focuses on two broader areas: *step-indexing* (§1.2.1) and *automation* (§1.2.2).

[12] Jung et al., "Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning", 2015 [Jun+15]; Jung et al., "Higher-order ghost state", 2016 [Jun+16]; Krebbers et al., "The essence of higher-order concurrent separation logic", 2017 [Kre+17]; Jung et al., "Iris from the ground up: A modular foundation for higher-order concurrent separation logic", 2018 [Jun+18b].

[13] Turon, Dreyer, and Birkedal, "Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency", 2013 [TDB13].

[14] Svendsen and Birkedal, "Impredicative concurrent abstract predicates", 2014 [SB14].

[15] Svendsen, Birkedal, and Parkinson, "Modular reasoning about separation of concurrent data structures", 2013 [SBP13].

[16] Jacobs et al., "VeriFast: A powerful, sound, predictable, fast verifier for C and Java", 2011 [Jac+11].

[17] Dinsdale-Young et al., "Concurrent abstract predicates", 2010 [Din+10].

[18] Appel, "Verified Software Toolchain", 2012 [App12]; Cao et al., "VST-Floyd: A separation logic tool to verify correctness of C programs", 2018 [Cao+18].

[19] Reynolds, "Separation Logic: A logic for shared mutable data structures", 2002 [Rey02]; O'Hearn, Reynolds, and Yang, "Local reasoning about programs that alter data structures", 2001 [ORY01].

[20] Jung et al., "Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning", 2015 [Jun+15].

[21] Turon, Dreyer, and Birkedal, "Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency", 2013 [TDB13]; Frumin, Krebbers, and Birkedal, "ReLoC: A mechanised relational logic for fine-grained concurrency", 2018 [FKB18].

[22] Appel and McAllester, "An indexed model of recursive types for foundational proof-carrying code", 2001 [AM01]; Ahmed et al., "Semantic foundations for typed assembly languages", 2010 [Ahm+10].

[23] Krebbers, Timany, and Birkedal, "Interactive proofs in higher-order concurrent separation logic", 2017 [KTB17].

For each area, it presents two projects. We briefly summarize their high-level contributions below, and we discuss their underlying technical contributions at the beginning of the corresponding part of the dissertation.

1.2.1 Generalizing Step-Indexing with Transfinite Iris and Later Credits

As mentioned above, Iris derives many of its recursive reasoning principles from a technique called "step-indexing". Step-indexing[24] is—alongside separation logic—one of the pivotal advances of the last two decades. It provides powerful recursive reasoning principles that—unlike co-induction and induction—support negative recursive occurrences. It has been essential for many of Iris's more advanced applications, including reasoning about languages with cyclic features such as recursive types[25] or higher-order state,[26] because the cycles that arise when modeling these features go beyond standard inductive or co-inductive reasoning.

Unfortunately, step-indexing comes at a cost. It was carefully designed for specific use cases (*e.g.,* proving safety of programs in languages with higher-order state[27]) and requires closely following the path forged by its inventors—leave the beaten path and wilderness awaits. In particular, at a fundamental level, step-indexing is designed for proving safety properties and, traditionally, does not apply to proving liveness properties. And for proving safety properties, the standard path requires a delicate alignment of recursive reasoning in the logic and program steps. If the alignment does not work out "as usual", perfectly natural proof strategies turn into dead ends.

To generalize step-indexing in the context of Iris, this dissertation presents two projects: **Transfinite Iris** (Part II) and **Later Credits** (Part III).

[24] Appel and McAllester, "An indexed model of recursive types for foundational proof-carrying code", 2001 [AM01]; Ahmed, Dreyer, and Rossberg, "State-dependent representation independence", 2009 [ADR09]; Ahmed et al., "Semantic foundations for typed assembly languages", 2010 [Ahm+10]; Dreyer, Ahmed, and Birkedal, "Logical step-indexed logical relations", 2011 [DAB11].

[25] Giarrusso et al., "Scala step-by-step: Soundness for DOT with step-indexed logical relations in Iris", 2020 [Gia+20].

[26] Jung et al., "RustBelt: Securing the foundations of the Rust programming language", 2018 [Jun+18a]; Guéneau et al., "Melocoton: A program logic for verified interoperability between OCaml and C", 2023 [Gué+23].

[27] Ahmed, "Semantics of types for mutable state", 2004 [Ahm04].

Liveness reasoning with Transfinite Iris. Transfinite Iris brings liveness reasoning to Iris by generalizing its *step-indexing* pillar. Traditionally, step-indexing captures only the *finitary behavior* of program executions, which works for safety properties (whose violations are finite execution prefixes) but falls short for liveness properties (whose violations are infinite executions). More specifically, with step-indexing, a property $\Phi(e)$ is typically stratified into a family of approximations $\Phi_i(e)$ for $i : \mathbb{N}$, where each approximation $\Phi_i(e)$ is fully determined by the first i steps of the program e. The number i is called the "*step-index*", and the original property $\Phi(e)$ is the intersection over the finite-prefix approximations for all step-indices (*i.e.,* $\Phi(e)$ *iff.* $\forall i : \mathbb{N}.\ \Phi_i(e)$). This approach works well for safety properties (because one considers all finite prefixes), but it falls short for liveness properties.

With Transfinite Iris, we fundamentally change the model of Iris: we move from finite step-indexing with natural numbers as the step-indices to transfinite step-indexing with ordinals as step-indices. We then identify a key property, the "existential property", as a high-level reasoning principle that makes liveness reasoning accessible to users of Transfinite Iris without diving into the details of transfinite step-indices. We demonstrate the effectiveness of Transfinite Iris by developing two program logics for sequential, higher-order stateful programs—one for termination and one for termination-preserving refinement.

Amortized step-indexing with Later Credits. Later Credits "amortize" step-indexing by marrying the *step-indexing* and *resource algebra* pillars of Iris. They provide a resource-based approach to step-indexing that complements Iris's existing step-indexing mechanism. That is, from a user perspective, the step-indexing in the model of Iris (*i.e.,* the step-indices i and the stratification described above) are encapsulated behind the so-called "*later modality*"[28] $\triangleright P$: propositions P are modeled as predicates over step-indices i, and $\triangleright P$ is defined to be true at step-index i if P is true at step-index $i - 1$. Traditionally, the later modality—a "logical step"—is tightly coupled with program steps (*i.e.,* one later modality occurs per step of computation) such that $\triangleright P$ means that P will hold after one step of computation. But in practice, this tight coupling is often too rigid: it complicates proofs, and it even prohibits some proofs entirely.

[28] Appel et al., "A very modal model of a modern, major, general type system", 2007 [App+07]; Dreyer, Ahmed, and Birkedal, "Logical step-indexed logical relations", 2011 [DAB11].

With Later Credits, we observe that the tight coupling between laters and program steps is unnecessary. To be sound, it suffices to "amortize" the step-index decreases over the execution of the program. We take advantage of this insight by leveraging the *resource algebra* pillar of Iris: we turn "the right to eliminate a later" (*i.e.,* to decrease the step-index) into an ownable resource, *a later credit* $£1$, which is subject to all the traditional modular reasoning principles of separation logic including framing. We demonstrate the usefulness of Later Credits by using them for several challenging examples and proof patterns which were previously not possible in Iris.

1.2.2 Improving Automation with Quiver and Daenerys

Iris has been very successful as a foundation for developing new program logics[29] and as a technique for building semantic models of languages.[30] It has been used less, ironically, for verifying programs. That is, projects building on Iris typically come with a handful of examples—intricate, hard-to-verify examples that go beyond the state-of-the-art—but often stop short of scaling to larger programs or code bases. A—if not *the*—limiting factor in this regard is *automation*: examples in Iris are almost always verified manually in Rocq using the Iris Proof Mode. Thus, alongside step-indexing, the second area that this dissertation focuses on is automation.

Recently, several projects have taken the first steps toward bringing more automation to Iris, including RefinedC[31] (automated C verification), RefinedRust[32] (automated Rust verification), and Diaframe[33] (automated verification of fine-grained concurrent programs). But their *verification overhead* remains considerable. In terms of *proof overhead*, they still fall short of more automated techniques like Verifast,[34] CN,[35] Viper,[36] and Verus,[37] which leverage SMT solvers to discharge a significant portion of the proof burden. And besides proof overhead, they also induce considerable *specification overhead*: Separation logic verification proceeds compositionally—function by function—and thus, typically, requires one to provide one specification per function. These specifications can be long, complex, or tedious to formulate—forcing the user to supply mundane side conditions about integer arithmetic, nontrivial preconditions about pointers, error cases, and conditionals over the return values.

To improve automation in the context of Iris, this dissertation presents two projects: **Quiver** (Part IV) and **Daenerys** (Part V).

[29] For example, see Timany et al., "Trillium: Higher-order concurrent and distributed separation logic for intensional refinement", 2024 [Tim+24a]; Gäher et al., "Simuliris: A separation logic framework for verifying concurrent program optimizations", 2022 [Gäh+22]; Frumin, Krebbers, and Birkedal, "ReLoC: A mechanised relational logic for fine-grained concurrency", 2018 [FKB18].

[30] For example, see Jung et al., "RustBelt: Securing the foundations of the Rust programming language", 2018 [Jun+18a]; Dang et al., "RustBelt meets relaxed memory", 2020 [Dan+20]; Guéneau et al., "Melocoton: A program logic for verified interoperability between OCaml and C", 2023 [Gué+23].

[31] Sammler et al., "RefinedC: Automating the foundational verification of C code with refined ownership types", 2021 [Sam+21].

[32] Gäher et al., "RefinedRust: A type system for high-assurance verification of Rust programs", 2024 [Gäh+24].

[33] Mulder, Krebbers, and Geuvers, "Diaframe: Automated verification of fine-grained concurrent programs in Iris", 2022 [MKG22].

[34] Jacobs et al., "VeriFast: A powerful, sound, predictable, fast verifier for C and Java", 2011 [Jac+11].

[35] Pulte et al., "CN: Verifying systems C code with separation-logic refinement types", 2023 [Pul+23].

[36] Müller, Schwerhoff, and Summers, "Viper: A verification infrastructure for permission-based reasoning", 2017 [MSS17].

[37] Lattuada et al., "Verus: Verifying Rust programs using linear ghost types", 2023 [Lat+23].

Specification inference with Quiver. Quiver reduces the specification overhead of separation logic by developing specification inference in Iris based on the *weakest precondition* pillar of Iris. Quiver is the first technique for inferring functional correctness specifications in separation logic while simultaneously verifying foundationally that they are correct. To guide Quiver towards the final specification, we take hints from the user in the form of *a specification sketch*, and then complete the sketch using inference. To do so, we introduce a new *abductive deductive verification* technique, centered around the weakest precondition $\mathbf{wp}\ e\ \{Q\}$. It integrates ideas from abductive inference (for specification inference) with deductive separation logic automation (for foundational verification). The result is that users provide some guidance, but significantly less than with traditional deductive separation logic verification techniques. We evaluate Quiver on a range of case studies, including code from popular open-source libraries.

Destabilizing Iris with Daenerys. Daenerys makes progress toward more proof automation by bringing *heap-dependent expression assertions* to Iris via a generalization of its *resource algebra* pillar. Heap-dependent expression assertions are logic-level assertions that contain program-level expressions. They originate from an alternative approach to ownership reasoning called *implicit dynamic frames*:[38] whereas the resources of separation logic combine ownership with memory values (*e.g.*, as in the points-to $\ell \mapsto v$), implicit dynamic frames separate ownership and memory values. As a result, (1) read-only program expressions can appear in specifications as heap-dependent expression assertions and (2) first-order logic can be used to reason about them. Both have been crucial to the success of the automated verifier family Viper.[39]

[38] Smans, Jacobs, and Piessens, "Implicit Dynamic Frames: Combining dynamic frames and separation logic", 2009 [SJP09].

[39] Müller, Schwerhoff, and Summers, "Viper: A verification infrastructure for permission-based reasoning", 2017 [MSS17]; Astrauskas et al., "Leveraging Rust types for modular specification and verification", 2019 [Ast+19]; Wolf et al., "Gobra: Modular specification and verification of Go programs", 2021 [Wol+21]; Eilers and Müller, "Nagini: A static verifier for Python", 2018 [EM18].

With Daenerys, we set out to relax the coupling between ownership and values in Iris to (1) define our own notion of heap-dependent expression assertions and (2) develop a technique for reasoning about them in first-order logic, laying the groundwork for SMT solver based automation. Unfortunately, this is easier said than done. At a fundamental level, heap-dependent expression assertions are incompatible with core reasoning principles of separation logic such as the central frame rule. To deal with this challenge, we shake up the foundations of Iris: we generalize its model of resources to encompass "unstable" resources (which do not survive framing) and define a *frame modality* $\boxplus P$ (which governs when an assertion is frameable). We then construct an unstable resource—the unstable points-to assertion $\ell \mapsto_{\mathrm{u}} v$—that serves as the foundation of our heap-dependent expression assertions. We apply Daenerys to several case studies, including some that go beyond what Viper and Iris can do individually and others that benefit from the connection to SMT.

1.3 Overview

The dissertation is divided into five parts. It starts with an introduction to Iris in Part I. Although Iris itself is *not a contribution* of this dissertation, it is the basis of all the actual contributions and thus important to review up front. Subsequently, the dissertation focuses first on *step-indexing* with **Transfinite Iris** (Part II) and **Later Credits** (Part III) and thereafter on *automation* with

Quiver (Part IV) and **Daenerys** (Part V). The dissertation concludes in §32 by summarizing the contributions and providing directions for future work.

1.4 Publications

This dissertation contains text and material from the following publications:

- Spies, Gäher, Gratzer, Tassarotti, Krebbers, Dreyer, and Birkedal. "Transfinite Iris: Resolving an existential dilemma of step-indexed separation logic" [Spi+21b]. 2021. Published in PLDI 2021.
- Spies, Gäher, Tassarotti, Jung, Krebbers, Birkedal, and Dreyer. "Later Credits: Resourceful reasoning for the later modality" [Spi+22a]. 2022. Published in ICFP 2022.
- Spies, Gäher, Sammler, and Dreyer. "Quiver: Guided abductive inference of separation logic specifications in Coq" [Spi+24a]. 2024. Published in PLDI 2024.
- Spies, Mück, Zeng, Sammler, Lattuada, Müller, and Dreyer. "Destabilizing Iris" [Spi+25a]. 2025. Published in PLDI 2025. This paper is the basis for Daenerys.

In addition, it contains text and material from the following notes:

- Dreyer, Spies, Gäher, Jung, Kaiser, Dang, Swasey, Menz, Mück, and Peters. *Semantics of type systems (lecture notes)* [Dre+25]. 2025. Unpublished lecture notes.

The dissertation reuses much of the text of these papers and notes, but it adapts them into a consistent presentation. More precisely, the provenance of the text in this dissertation is as follows:

- §1 is largely new text, reusing some material from the introductions of Transfinite Iris, Later Credits, Quiver, and Destabilizing Iris (aka Daenerys).
- Part I is an adaptation of the sections that I[40] authored for *Semantics of type systems (lecture notes)*. To obtain a self-contained presentation, the text is modified from the version in the notes and does not follow the same order.
- Part II is based on the Transfinite Iris paper and appendix.[41] The presentation is adapted to fit the introduction to Iris in Part I. Moreover, the text expands on several aspects that are kept brief in the original paper: the text in §7.3 is new, and the text in §7.1, §8.1, and §9.2 significantly expands on the original version. The section §8.3 has been integrated from the appendix.
- Part III is based on the Later Credits paper and appendix.[42] The presentation is adapted to fit the introduction to Iris in Part I. Moreover, the text in §12 contains a new example, and the text in §15.2 expands the discussion of transfinite step-indexing.
- Part IV is based on the Quiver paper and appendix.[43] The presentation is adapted to fit the introduction to Iris in Part I. Moreover, the text in §20.3 is new, and the discussion in §22.1 expands on the original version. The sections §20.1 and §20.2 are integrated from the appendix.

[40] Here and in §1.5, I use the first person singular to distinguish my contributions from those of my collaborators. The remainder of the dissertation uses the first person plural.

[41] Spies et al., *Transfinite Iris appendix and Rocq development*, 2021 [Spi+21a].

[42] Spies et al., *Later Credits Rocq development and technical documentation*, 2022 [Spi+22b].

[43] Spies et al., *Quiver: Guided abductive inference of separation logic specifications in Coq (Rocq development and appendix)*, 2024 [Spi+24b].

- Part V is based on the Daenerys paper and appendix.[44] The presentation is adapted to fit the introduction to Iris in Part I. The text in §26, §27, §29.1, and §30 is a synthesis of the corresponding parts in the paper and its appendix. The text also expands on the discussion in §27.

[44] Spies et al., *Destabilizing Iris (Rocq development and appendix)*, 2025 [Spi+25b].

- §32 is mostly new text, but reuses some material and text from the future work discussed in Transfinite Iris, Later Credits, Quiver, and Daenerys.

1.5 Collaborations

The contributions of this dissertation are the result of many collaborations. While they were all led by me, they would not have been possible without the work of my tremendous collaborators. In the following, I describe for each project my contributions and distinguish them from those of my collaborators.

For Transfinite Iris, I updated most of the model of Iris (except for the recursive domain equation), developed the program logics and the weakest precondition, generalized the existential property to large ordinals, led the Rocq mechanization effort, and led the writing. A first version of the existential property was suggested by Joseph Tassarotti for uncountable ordinals. I generalized it to larger ordinals leveraging the power of Rocq's universes. Together with Lennard Gäher, I mechanized the existential property, building on a set theory mechanization of Kirst and Smolka[45] for the definition of ordinals. Lennard Gäher and Daniel Gratzer solved and mechanized the recursive domain equation of Transfinite Iris; I contributed the insight that it suffices to solve the equation for functors from OFEs to COFEs instead of COFEs to COFEs, which was necessary for the model to be sound. Joseph Tassarotti mechanized the memorec case study, and I mechanized the remaining case studies.

[45] Kirst and Smolka, "Categoricity results and large model constructions for second-order ZF in dependent type theory", 2019 [KS19].

For Later Credits, I developed the definition of the later elimination update and the later credits, proved their soundness, integrated them into the weakest precondition, led the Rocq mechanization effort, and led the writing. Joseph Tassarotti contributed the logical relations for reordering refinements and reverse refinements. Ralf Jung and Joseph Tassarotti contributed the counter with a backup, and Ralf Jung removed make-laterable from logically atomic specifications. Lennard Gäher mechanized the promises example after I had sketched the key transition system invariant on paper.

For Quiver, I developed the abductive deductive verification technique, designed and implemented most of the abduction engine Argon and the type system Thorium, verified case studies including the Vector, led the Rocq mechanization, and led the writing. Lennard Gäher and Michael Sammler contributed several case studies. Lennard did the length version of the OpenSSL buffer, and Michael did the memcached buffer and most of the mutable Hashmap. Moreover, Lennard adapted the frontend of RefinedC[46] for Quiver, and he built a large part of the specification sketch machinery (after I had built the machinery for existential instantiation).

[46] Sammler et al., "RefinedC: Automating the foundational verification of C code with refined ownership types", 2021 [Sam+21].

For Daenerys, I developed the initial unstable heap points-to, the evaluation judgment, the extension of resource algebras, the program logic, the logical relation, and the first-order logic connection. Moreover, I led the Rocq mechanization effort and the writing. Haoyi Zeng and I jointly developed the resource algebra combinators for modeling the heap, and we adapted the Iris Proof

Mode.[47] Niklas Mück, Haoyi Zeng, Michael Sammler, and I developed the case studies. After joint discussions, Haoyi Zeng mechanized the Hashmap, Niklas Mück mechanized the SMT-based examples and ported most of the existing Iris examples, and Michael Sammler mechanized the iterative linked list verification. I mechanized the concurrent checksum example.

[47] Krebbers, Timany, and Birkedal, “Interactive proofs in higher-order concurrent separation logic”, 2017 [KTB17].

Lastly, while Iris is *not* a contribution of this dissertation, Part I gives an introduction to Iris. It is based on the second half of Derek’s *Semantics* course.[48] This half was developed while Lennard Gäher and I were teaching assistants for the course. I developed most of the text and structure for this half of the course, Lennard Gäher mechanized it in Rocq, and Derek presented it in class.

[48] Dreyer et al., *Semantics of type systems (lecture notes)*, 2025 [Dre+25].

Availability. All projects presented in this dissertation are open source and mechanized in Rocq. The Rocq implementations can be found at

1. **Transfinite Iris** (Part II):
   `https://gitlab.mpi-sws.org/iris/transfinite`
2. **Later Credits** (Part III):
   `https://gitlab.mpi-sws.org/iris/iris` for the latest version, which is integrated into the main Iris development.[49]
3. **Quiver** (Part IV):
   `https://gitlab.mpi-sws.org/iris/quiver`
4. **Daenerys** (Part V):
   `https://gitlab.mpi-sws.org/iris/daenerys`

[49] See `https://doi.org/10.5281/zenodo.6702804` for the original version including the applications and extensions.

Part I

An Iris Primer

Chapter 2

Separation Logic

As a foundation for the subsequent parts, we provide a gentle introduction to Iris. We start by introducing separation logic (in §2), continue with how Iris goes beyond traditional separation logic (in §3), and finally discuss the soundness of Iris (in §4). While the original presentation of separation logic uses an imperative language with local variables,[1] we will discuss a flavor of separation logic that is closer to the kinds of separation logics one typically constructs with Iris. Concretely, we focus on an ML-like functional language with references, called HeapLang. Readers familiar with separation logic can skip this chapter but may want to consult it later as a reference for the language.

[1] Reynolds, "Separation Logic: A logic for shared mutable data structures", 2002 [Rey02]; O'Hearn, Reynolds, and Yang, "Local reasoning about programs that alter data structures", 2001 [ORY01].

2.1 Purely Functional Programs

We start simple—with programs that do not even modify state. For our first example, we verify Euclid's algorithm.

Example 1 (Euclid)**.** Euclid's algorithm computes the greatest common divisor of two numbers a and b by repeatedly computing the modulus of the two:

$$\begin{aligned} \texttt{euclid}(a, b) &\triangleq \textsf{if}\ b == 0\ \textsf{then}\ a\ \textsf{else}\ \texttt{euclid}(b, \texttt{mod}(a, b)) \\ \texttt{mod}(a, b) &\triangleq a - (a\ \textsf{div}\ b) * b \end{aligned}$$

Our goal is to verify the correctness of the algorithm, which means we will prove the following specification:

$$\{a \geq 0 \land b \geq 0\}\ \texttt{euclid}(a, b)\ \{c.\ gcd(a, b, c)\}$$

It means that if a and b are nonnegative integers, then the result c is the greatest common divisor, defined by $gcd(a, b, c) \triangleq c|a \land c|b \land (\forall d.\ d|a \land d|b \Rightarrow d|c)$ where $n|m$ means "n divides m". ●

To verify Euclid's algorithm, we introduce a *program logic*.[2] Its main judgment will be the *Hoare triple* $\{P\}\ e\ \{v.\ Q(v)\}$, which expresses that if the *precondition* P is true, then the expression e will execute to a value v (if it terminates) satisfying the *postcondition* $Q(v)$. Importantly, we only prove partial correctness, meaning e is allowed to diverge and, *only if it terminates*, then the postcondition $Q(v)$ must be satisfied. The pre- and postconditions can contain the following propositions:

[2] To verify purely functional programs, one can usually use simpler solutions than a full-blown program logic (*e.g.,* reasoning directly about the operational semantics). We use pure examples here as an on-ramp to more interesting, stateful examples that warrant a program logic.

$$\text{Propositions} \quad P, Q, R \quad ::= \quad \phi \mid \exists x.\ P(x) \mid \forall x.\ P(x) \mid P \land Q \mid P \lor Q \mid \cdots$$

Here, ϕ is an arbitrary meta-level proposition such as True, False, $n < 0$, even(n), or $gcd(a, b, c)$. (In Iris, pure assertions ϕ correspond to Rocq-level propositions.) We will extend the propositions as we make the program logic more expressive.

The programming language. The programming language that we consider is called HeapLang. It is Iris's "default" programming language.[3] It is an ML-style functional language with right-to-left evaluation order and recursive functions $\mathsf{fix}\, f\, x.\, e$, sums, and pairs.

[3] Iris can be instantiated with different programming languages. HeapLang is a simple, ML-like language that is commonly used for simple examples. We use it as the main programming language in most of this dissertation.

$$\begin{array}{rrl}
\text{Expressions} & e & ::= x \mid v \mid \mathsf{fix}\, f\, x.\, e \mid e_1\, e_2 \mid e_1 \odot e_2 \mid \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 \\
 & & \mid\ \mathsf{match}\, e_1\, \mathsf{with}\, \mathsf{inj}_1\, x \Rightarrow e_2 \mid \mathsf{inj}_2\, y \Rightarrow e_3\, \mathsf{end} \\
 & & \mid\ (e_1, e_2) \mid \pi_1 e \mid \pi_2 e \mid \mathsf{inj}_1 e \mid \mathsf{inj}_2 e \mid \cdots \\
\text{Values} & v & ::= \mathsf{fix}\, f\, x.\, e \mid () \mid n \mid (v_1, v_2) \mid \mathsf{inj}_1 v \mid \mathsf{inj}_2 v \mid \mathsf{true} \mid \mathsf{false} \mid \cdots
\end{array}$$

Here, $\odot$ ranges over binary operators ($\texttt{+}, \texttt{-}, \texttt{*}, \texttt{div}, \texttt{==}, \ldots$), and we abbreviate $\lambda x.\, e \triangleq (\mathsf{fix}\, \_\, x.\, e)$, $\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \triangleq (\lambda x.\, e_2)\, e_1$, and $e_1; e_2 \triangleq \mathsf{let}\, \_ = e_1\, \mathsf{in}\, e_2$.

For example, we can define the functions `euclid` and `mod` as:

$$\begin{aligned}
\texttt{euclid} &\triangleq \mathsf{fix}\, \mathsf{euc}\, x.\, \mathsf{if}\, \pi_2\, x \,\texttt{==}\, 0\, \mathsf{then}\, \pi_1\, x\, \mathsf{else}\, \mathsf{euc}(\pi_2\, x, \texttt{mod}(x)) \\
\texttt{mod} &\triangleq \lambda x.\, (\pi_1\, x) \,\texttt{-}\, ((\pi_1\, x)\, \texttt{div}\, (\pi_2\, x)) \,\texttt{*}\, (\pi_2\, x)
\end{aligned}$$

However, for the sake of readability, we will often use notation as in Example 1: we use pattern matching notation for pairs as opposed to projections and omit the fixpoint/lambda operators. We also use record notation $\{\mathsf{f} \triangleq e_1, \mathsf{g} \triangleq e_2, \ldots\}$ for better readability, which the reader can think of as pairs with named projections, and we define options as $\mathsf{None} \triangleq \mathsf{inj}_1\, ()$ and $\mathsf{Some}(e) \triangleq \mathsf{inj}_2(e)$.

The program logic. To prove Hoare triples such as the one in Example 1, we use the following program logic rules:

Hoare Rules $\boxed{\{P\}\, e\, \{v.\, Q(v)\}}$

hoare-value
$$\{P(v)\}\, v\, \{w.\, P(w)\}$$

hoare-conseq
$$\frac{P \vdash P' \qquad \{P'\}\, e\, \{v.\, Q'(v)\} \qquad (\forall v.\, Q'(v) \vdash Q(v))}{\{P\}\, e\, \{v.\, Q(v)\}}$$

hoare-bind
$$\frac{\{P\}\, e\, \{v.\, Q(v)\} \qquad \forall v.\, \{Q(v)\}\, K[v]\, \{w.\, R(w)\}}{\{P\}\, K[e]\, \{w.\, R(w)\}}$$

hoare-exists
$$\frac{\forall x : X.\, \{P(x)\}\, e\, \{v.\, Q(v)\}}{\{\exists x : X.\, P(x)\}\, e\, \{v.\, Q(v)\}}$$

hoare-pure-step
$$\frac{e_1 \rightarrow_{\mathsf{pure}} e_2 \qquad \{P\}\, e_2\, \{v.\, Q(v)\}}{\{P\}\, e_1\, \{v.\, Q(v)\}}$$

hoare-pure
$$\frac{P \vdash \phi \qquad \phi \Rightarrow \{P\}\, e\, \{v.\, Q(v)\}}{\{P\}\, e\, \{v.\, Q(v)\}}$$

Figure 2.1: Program logic rules for purely functional programs.

The rule hoare-value says that values must satisfy the postcondition P. The rule hoare-conseq is the standard rule of consequence. It allows us to strengthen the precondition and weaken the postcondition using an entailment judgment $P \vdash Q$. The rule hoare-exists allows us to destruct existential quantifiers in the precondition. The rules hoare-pure, hoare-bind, and hoare-pure-step are somewhat non-standard. We present them in this form to match how rules in Iris are formulated. The rule hoare-pure allows us to use pure propositions from the precondition; its implication "$\phi \Rightarrow \cdots$" should be read as a "meta-level" implication. (The reader can think of it as an implication in Rocq.) The rule hoare-bind allows us to focus on a subexpression e. As we will see below, it generalizes the standard rule for sequential composition of Hoare logic.[4] The rule hoare-pure-step allows us to execute arbitrary pure reduction steps in a Hoare triple (*e.g.*, reducing if true then 42 else 0 to 42). We give the rules for pure steps[5] $e_1 \rightarrow_{\text{pure}} e_2$ and entailments $P \vdash Q$ below.

[4] Hoare, "An axiomatic basis for computer programming", 1969 [Hoa69].

[5] Here, $e[v/x]$ denotes the expression resulting from substituting v for x in e and $\text{trunc}(x)$ denotes the result of rounding x toward 0, meaning $\text{trunc}(x) = \lfloor x \rfloor$ for $x \geq 0$ and $\text{trunc}(x) = \lceil x \rceil$ for $x < 0$.

Pure Steps $\boxed{e_1 \rightarrow_{\text{pure}} e_2}$

$$\frac{e_1 \rightarrow_{\text{pure}} e_2}{K[e_1] \rightarrow_{\text{pure}} K[e_2]} \qquad (\text{fix}\ f\ x.\ e)\ v \rightarrow_{\text{pure}} e[\text{fix}\ f\ x.\ e/f, v/x]$$

$$\text{if true then } e_1 \text{ else } e_2 \rightarrow_{\text{pure}} e_1 \qquad \text{if false then } e_1 \text{ else } e_2 \rightarrow_{\text{pure}} e_2$$

$$n * m \rightarrow_{\text{pure}} (n \cdot m) \qquad n - m \rightarrow_{\text{pure}} (n - m) \qquad n + m \rightarrow_{\text{pure}} (n + m)$$

$$\frac{m \neq 0}{n \,\text{div}\, m \rightarrow_{\text{pure}} \text{trunc}(n/m)} \qquad \frac{n = m}{n == m \rightarrow_{\text{pure}} \text{true}} \qquad \frac{n \neq m}{n == m \rightarrow_{\text{pure}} \text{false}}$$

$$\pi_1(v_1, v_2) \rightarrow_{\text{pure}} v_1 \qquad \pi_2(v_1, v_2) \rightarrow_{\text{pure}} v_2$$

$$(\text{match inj}_1 v \text{ with inj}_1\ x \Rightarrow e_1 \mid \text{inj}_2\ y \Rightarrow e_2 \text{ end}) \rightarrow_{\text{pure}} e_1[v/x]$$

$$(\text{match inj}_2 v \text{ with inj}_1\ x \Rightarrow e_1 \mid \text{inj}_2\ y \Rightarrow e_2 \text{ end}) \rightarrow_{\text{pure}} e_2[v/y]$$

Entailments $\boxed{P \vdash Q}$

$$\text{ENT-REFL}\ \ P \vdash P \qquad \text{ENT-TRANS}\ \frac{P \vdash Q \quad Q \vdash R}{P \vdash R} \qquad \text{PURE-INTRO}\ \frac{\phi}{P \vdash \phi} \qquad \text{FROM-PURE}\ \frac{P \vdash \phi \quad \phi \Rightarrow (P \vdash Q)}{P \vdash Q}$$

$$\text{AND-ELIM-L}\ \ P \land Q \vdash P \qquad \text{AND-ELIM-R}\ \ P \land Q \vdash Q \qquad \text{AND-INTRO}\ \frac{P \vdash Q \quad P \vdash R}{P \vdash Q \land R} \qquad \text{OR-INTRO-L}\ \ P \vdash P \lor Q$$

$$\text{OR-INTRO-R}\ \ Q \vdash P \lor Q \qquad \text{OR-ELIM}\ \frac{P \vdash R \quad Q \vdash R}{P \lor Q \vdash R} \qquad \text{ALL-INTRO}\ \frac{\forall x : X.\ (P \vdash Q(x))}{P \vdash \forall x : X.\ Q(x)}$$

$$\text{ALL-ELIM}\ \frac{a : X}{(\forall x : X.\ P(x)) \vdash P(a)} \qquad \text{EXIST-INTRO}\ \frac{a : X \quad P \vdash Q(a)}{P \vdash \exists x : X.\ Q(x)} \qquad \text{EXIST-ELIM}\ \frac{\forall x : X.\ (P(x) \vdash Q)}{\exists x : X.\ P(x) \vdash Q}$$

Most of the rules are completely standard. Similar to the rule hoare-pure, they are formulated with respect to an ambient meta logic. For example, to prove a pure proposition (pure-intro), we prove it in the meta logic, and to instantiate a universal quantifier (all-elim), we pick a value $a : X$.

Evaluation contexts. As mentioned above, the rule HOARE-BIND allows us to focus on a subexpression e. More specifically, we can first verify e and then continue with the remainder $K[v]$ where v is the result of e. We can focus on any subexpression in an *evaluation context* K.

$$\begin{array}{lrl} \text{Eval. Contexts} & K ::= & \bullet \mid e_1\, K \mid K\, v_2 \mid e_1 \odot K \mid K \odot v_2 \mid \mathsf{if}\, K\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 \\ & \mid & \mathsf{match}\, K\, \mathsf{with}\, \mathsf{inj}_1\, x \Rightarrow e_2 \mid \mathsf{inj}_2\, y \Rightarrow e_3\, \mathsf{end} \\ & \mid & (e_1, K) \mid (K, v_2) \mid \pi_1 K \mid \pi_2 K \mid \mathsf{inj}_1 K \mid \mathsf{inj}_2 K \mid \cdots \end{array}$$

Evaluation contexts capture, with a hole $\bullet$, which subexpression is currently in evaluation position. For example, for an addition $e_1 + e_2$, we can first focus on the right side (because of right-to-left evaluation order) with $K_1 \triangleq e_1 + \bullet$ and, once that has been evaluated to a value, we can focus on the left side with $K_2 \triangleq \bullet + v_2$. We use evaluation contexts (and HOARE-BIND) to derive the standard sequential composition rule of Hoare-logic:

Lemma 2. *If* $\{P\}\, e_1\, \{\_.\, Q\}$ *and* $\{Q\}\, e_2\, \{w.\, R(w)\}$*, then* $\{P\}\, e_1; e_2\, \{w.\, R(w)\}$*.*

Proof. Using HOARE-BIND with $\{P\}\, e_1\, \{\_.\, Q\}$ and $K \triangleq \bullet; e_2$, it suffices to prove

$$\forall v.\ \{Q\}\, v; e_2\, \{w.\, R(w)\}$$

We execute one pure step with HOARE-PURE-STEP (reducing $v; e_2$ to e_2) and are left to prove the triple $\{Q\}\, e_2\, \{w.\, R(w)\}$, which is our second assumption. □

Euclid's algorithm. With the program logic rules in hand, we can return to verifying Euclid's algorithm. We first prove a helper lemma for the `mod` function, namely that it returns the remainder after dividing b by a:

Lemma 3. $\{a \geq 0 \land b > 0\}\, \texttt{mod}(a, b)\, \{v.\, \exists c, k \geq 0.\, v = c \land a = b \cdot k + c \land 0 \leq c < b\}$

Proof. Using the rule HOARE-PURE-STEP, it suffices to prove

$$\{a \geq 0 \land b > 0\}\, a - (a\, \mathsf{div}\, b) * b\, \{v.\, \exists c, k \geq 0.\, v = c \land a = b \cdot k + c \land 0 \leq c < b\}$$

Using HOARE-PURE, we can assume that $a \geq 0$ and $b > 0$, which ensures that $b \neq 0$. Thus, we can use HOARE-PURE-STEP to execute the division $a\, \mathsf{div}\, b$ and the remainder of the function. We arrive at

$$\{a \geq 0 \land b > 0\}\, n\, \{v.\, \exists c, k \geq 0.\, v = c \land a = b \cdot k + c \land 0 \leq c < b\}$$

for $n \triangleq a - \mathrm{trunc}(a/b) \cdot b$ (which is $a - \lfloor a/b \rfloor \cdot b$). We can prove the postcondition with HOARE-CONSEQ by picking $k \triangleq \lfloor a/b \rfloor$. □

In a similar fashion, we can then prove the correctness of `euclid`:

Lemma 4. $\{a \geq 0 \land b \geq 0\}\, \texttt{euclid}(a, b)\, \{v.\, \exists c.\, v = c \land gcd(a, b, c)\}$

Proof. By strong induction on b. We execute the `euclid` function using pure steps and, once we reach `mod`, apply Lemma 3. □

2.2 Ownership Reasoning

Let us now turn to stateful programs and the essence of separation logic, *ownership reasoning*. We extend our language with references:

$$\begin{array}{rcll} \text{Expressions} & e & ::= & \cdots \mid \mathsf{ref}(e) \mid e_1 := e_2 \mid\ !\, e \\ \text{Values} & v & ::= & \cdots \mid \ell \\ \text{Eval. Contexts} & K & ::= & \cdots \mid \mathsf{ref}(K) \mid e_1 := K \mid K := v_2 \mid\ !\, K \end{array}$$

The expression $\mathsf{ref}(v)$ allocates a new reference ℓ storing the value v, the expression $\ell := w$ updates the reference ℓ to store the value w, and the expression $!\,\ell$ retrieves the value of ℓ from memory.

Example 5 (Linked Lists)**.** As our next example, we consider a linked-list implementation. To represent list elements, the linked-list implementation uses optional references to pairs, and the empty list is represented by None.

$$\begin{aligned}
\mathtt{new}() &\triangleq \mathsf{None}\\
\mathtt{cons}(a,x) &\triangleq \mathsf{let}\ r = \mathsf{ref}(a,x)\ \mathsf{in}\ \mathsf{Some}(r)\\
\mathtt{head}(x) &\triangleq \mathsf{match}\ x\ \mathsf{with}\ \mathsf{None} \Rightarrow ()\mid \mathsf{Some}(r) \Rightarrow \mathsf{let}\ (a,x) = !r\ \mathsf{in}\ a\ \mathsf{end}\\
\mathtt{tail}(x) &\triangleq \mathsf{match}\ x\ \mathsf{with}\ \mathsf{None} \Rightarrow ()\mid \mathsf{Some}(r) \Rightarrow \mathsf{let}\ (a,x) = !r\ \mathsf{in}\ x\ \mathsf{end}\\
\mathtt{len}(x) &\triangleq \mathsf{match}\ x\ \mathsf{with}\ \mathsf{None} \Rightarrow 0\\
&\quad \mid \mathsf{Some}(r) \Rightarrow \mathsf{let}\ (a,x) = !\,r\ \mathsf{in}\ \mathtt{len}(x)+1\ \mathsf{end}\\
\mathtt{app}(x,y) &\triangleq \mathsf{match}\ x\ \mathsf{with}\ \mathsf{None} \Rightarrow y\\
&\quad \mid \mathsf{Some}(r) \Rightarrow \mathsf{let}\ (a,x) = !r\ \mathsf{in}\ r := (a, \mathtt{app}(x,y));\mathsf{Some}(r)\ \mathsf{end}\\
\mathtt{lookup}(x,i) &\triangleq \mathsf{match}\ x\ \mathsf{with}\ \mathsf{None} \Rightarrow \mathsf{None}\\
&\quad \mid \mathsf{Some}(r) \Rightarrow \mathsf{if}\ i{=}{=}0\ \mathsf{then}\ \mathsf{Some}(r)\ \mathsf{else}\ \mathtt{lookup}(\pi_2(!r), i{-}1)\ \mathsf{end}
\end{aligned}$$

●

Separation logic. To verify functions such as the linked-list implementation, we extend our program logic with reasoning principles for memory. We add the following separation logic assertions to describe the current values of memory.

$$\text{Propositions}\quad P,Q,R \quad ::= \quad \cdots \mid \ell \mapsto v \mid P * Q \mid P \mathrel{-\!\!*} Q$$

The proposition $\ell \mapsto v$ (read "ℓ points to v") asserts that the reference ℓ currently stores the value v. The proposition $P * Q$ (read "P star Q") is the name-giving feature of separation logic, the *separating conjunction*. Like conjunction, it asserts that both P and Q are true. But what makes it special is that it ensures P and Q are satisfied by *disjoint* parts of the heap. That is, the proposition $\ell \mapsto v * \ell \mapsto w$ is false (even if $v = w$), because both conjuncts refer to the same reference (*i.e.*, overlapping parts of the heap). The assertion $P \mathrel{-\!\!*} Q$ (read "P wand Q") is the so-called *magic wand*. It is the corresponding notion of implication for the separating conjunction.

We extend our entailment and Hoare triple rules. The extension of the entailment rules is depicted in Fig. 2.2, where we write $P \dashv\vdash Q$ as a shorthand for $P \vdash Q$ and $Q \vdash P$. The separating conjunction $P * Q$ is commutative (sep-comm), associative (sep-assoc), and distributes over existential quantification (sep-exists). It can be introduced if one side is True (sep-true) or by splitting a separating conjunction (sep-split). Moreover, the separation logic we consider is *affine*, which means we can drop conjuncts from the separating conjunction (sep-weaken).[6] The points-to assertion $\ell \mapsto v$ asserts *exclusive ownership* of ℓ such that we can never have two points-to assertions to the same location (points-to-sep). As mentioned above, the magic wand $P \mathrel{-\!\!*} Q$ is the corresponding notion of implication for the separating conjunction. This is reflected in its rules for introduction (wand-intro) and elimination (wand-elim).

[6] A *linear* separation logic disallows this rule. Linear separation logic (as opposed to affine separation logic) can be useful for languages like C where it is often considered to be a bug to silently drop memory resources (*i.e.*, a memory leak).

The extension of the Hoare triple rules is depicted in Fig. 2.3. The rule hoare-ref allocates a new reference resulting in a points-to assertion $\ell \mapsto v$ for the

Entailments $\boxed{P \vdash Q}$

Figure 2.2: Separation logic extension of the entailment rules.

sep-weaken
$$P * Q \vdash P$$

sep-true
$$P \vdash P * \mathsf{True}$$

sep-comm
$$P * Q \dashv\vdash Q * P$$

sep-split
$$\frac{P \vdash P' \qquad Q \vdash Q'}{P * Q \vdash P' * Q'}$$

sep-assoc
$$P * (Q * R) \dashv\vdash (P * Q) * R$$

sep-exists
$$(\exists x : X.\, P(x)) * Q \dashv\vdash (\exists x : X.\, P(x) * Q)$$

points-to-sep
$$\ell \mapsto v * \ell \mapsto w \vdash \mathsf{False}$$

wand-intro
$$\frac{P * Q \vdash R}{P \vdash Q \mathrel{-\!\!*} R}$$

wand-elim
$$\frac{P \vdash Q \mathrel{-\!\!*} R}{P * Q \vdash R}$$

Hoare Rules $\boxed{\{P\}\ e\ \{v.\, Q(v)\}}$

Figure 2.3: Separation logic rules for stateful programs.

hoare-frame
$$\frac{\{P\}\ e\ \{v.\, Q(v)\}}{\{P * R\}\ e\ \{v.\, Q(v) * R\}}$$

hoare-ref
$$\{\mathsf{True}\}\ \mathsf{ref}\ v\ \{w.\, \exists \ell.\, w = \ell * \ell \mapsto v\}$$

hoare-load
$$\{\ell \mapsto v\}\ \ell := w\ \{\_.\, \ell \mapsto w\}$$

hoare-store
$$\{\ell \mapsto v\}\ !\,\ell\ \{w.\, w = v * \ell \mapsto v\}$$

new reference ℓ in the postcondition, the rule hoare-load loads the value v from memory, and the rule hoare-store updates a reference ℓ. The rule hoare-frame is the characteristic "frame rule" of separation logic. If we know $\{P\}\ e\ \{v.\, Q(v)\}$, it allows us to frame on a separate assertion R, which is not affected by e.

Ownership reasoning. We have already touched on the ownership reasoning enabled by separation logic (and in particular the frame rule) in §1. Let us now take a closer look. The idea is that the assertion $\ell \mapsto v$ expresses ownership of the reference ℓ, and owning a reference ℓ means that no other program part can manipulate it. For example, in the triple $\{\ell \mapsto 0\}\ f\ (\ell, \ell')\ \{\_.\, \ell \mapsto 42\}$ the function f "owns" ℓ for the duration of the call, and it can be sure that no other program part (even in a concurrent setting) will interfere with ℓ. Moreover, from the triple, we also know f *only* needs the reference ℓ from the current heap—ownership of all other references can be framed around the function call. For example, we can verify the following program:

$$e_{\mathsf{own}} \triangleq \mathsf{let}\ x = \mathsf{ref}(0)\ \mathsf{in}\ \mathsf{let}\ y = \mathsf{ref}(42)\ \mathsf{in}\ f(x, y); \mathsf{assert}\ (!\,x == !\,y)$$

where $\mathsf{assert}\ (e) \triangleq \mathsf{if}\ e\ \mathsf{then}\ ()\ \mathsf{else}\ 0\ 0$ can only be verified if e evaluates to true. The reason why we can verify e_{own} is that the specification of f ensures that f sets the first argument to 42 and, implicitly, that it does not modify (or access) the second argument.

Lemma 6.
$$\{\mathsf{True}\}\ e_{\mathsf{own}}\ \{\_.\, \mathsf{True}\}$$

Proof Sketch. To sketch the proof, we give a proof outline that contains separation logic assertions (in ■ orange) for the intermediate program points.[7]

[7] For the sake of readability, we refer to locations by the variables that they are bound to in this outline following Jung et al. [Jun+18b].

■ $\{\mathsf{True}\}$

$\mathsf{let}\ x = \mathsf{ref}(0)\ \mathsf{in}$

■ $\{x \mapsto 0\}$

$\mathsf{let}\ y = \mathsf{ref}(42)\ \mathsf{in}$

■ $\{x \mapsto 0 * y \mapsto 42\}$

$f(x, y);$

■ $\{x \mapsto 42 * y \mapsto 42\}$

$\mathsf{assert}\ (!\,x == !\,y)$

■ $\{x \mapsto 42 * y \mapsto 42\}$

■ $\{\mathsf{True}\}$

Note that we use the HOARE-FRAME rule here multiple times: The first time, we frame the ownership of $x \mapsto 0$ around the allocation of y. The second time, we frame the ownership of $y \mapsto 42$ around the call to f. How do we know that y is not altered by f? The answer is that f only demands ownership of $x \mapsto 0$ in its precondition (even though it also takes y as an argument) and, hence, e_{own} can keep the ownership of $y \mapsto 42$ (*i.e.,* we can frame it around the function call). Since we can frame it, the final assert succeeds, because f has set x to 42 and y has not changed. □

Linked list. Let us now return to the linked list implementation (from Example 5). To reason about lists, we define recursively what it means for a value v to represent the list xs:

$$\begin{aligned}\mathsf{list}(v, \mathsf{nil}) &\triangleq v = \mathsf{None}\\ \mathsf{list}(v, x :: xr) &\triangleq \exists \ell, w.\, v = \mathsf{Some}(\ell) * \ell \mapsto (x, w) * \mathsf{list}(w, xr)\end{aligned}$$

The predicate list is a so-called *abstract predicate* or *representation predicate.* It ties the concrete program representation (here the value v) to an abstract mathematical representation (here the mathematical list xs). If the list is empty, then the value v is None. If the list contains a head x and a tail xr, then the value v is Some of a reference ℓ that stores a pair containing the head x and a value w for the tail of the list.

We can characterize the linked-list operations relative to this abstract list predicate. We focus on the append function app, which implements concatenation of the mathematical lists (written $xs + ys$).

Lemma 7. $\{\mathsf{list}(v, xs) * \mathsf{list}(w, ys)\}\ \mathsf{app}(v, w)\ \{u.\, \mathsf{list}(u, xs +\!\!+\, ys)\}$

Proof. By induction on xs. In the case $xs = \mathsf{nil}$, we have to show

$$\{v = \mathsf{None} * \mathsf{list}(w, ys)\}\ \mathsf{app}(v, w)\ \{u.\, \mathsf{list}(u, ys)\}$$

In this case, we execute the function with pure steps until the list w is returned.

In the case $xs = x :: xr$, we have to show

$$\begin{aligned}&\{(\exists \ell, u.\, v = \mathsf{Some}(\ell) * \ell \mapsto (x, u) * \mathsf{list}(u, xr)) * \mathsf{list}(w, ys)\}\\ &\quad \mathsf{app}(v, w)\\ &\{u.\, \mathsf{list}(u, x :: (xr +\!\!+\, ys))\}\end{aligned}$$

assuming $\forall v.\, \{\mathsf{list}(v, xr) * \mathsf{list}(w, ys)\}\ \mathsf{app}(v, w)\ \{u.\, \mathsf{list}(u, xr +\!\!+\, ys)\}$ by induction. After several pure steps and dereferencing ℓ, we are left to prove:

$$\begin{aligned}&\{\ell \mapsto (x, u) * \mathsf{list}(u, xr) * \mathsf{list}(w, ys)\}\\ &\quad \ell := (x, \mathsf{app}(u, w)); \mathsf{Some}(\ell)\\ &\{v'.\, \mathsf{list}(v', x :: (xr +\!\!+\, ys))\}\end{aligned}$$

Here, we frame the ownership of $\ell \mapsto (x, u)$ around the recursive call to app using the fact that $\mathsf{app}(u, w)$, by our inductive assumption, does not require ownership of ℓ. We are left to prove:

$$\{\ell \mapsto (x, u) * \mathrm{list}(v', xs +\!\!+ ys)\}\ \ell := (x, v'); \mathsf{Some}(\ell)\ \{v''.\ \mathrm{list}(v'', x :: (xr +\!\!+ ys))\}$$

where v' is the return value of the recursive call $\mathsf{app}(u, w)$. From here on, the proof is straightforward given the definition of the list predicate. □

The magic wand. The proofs for most of the other linked-list functions are analogous to Lemma 7. The only one that stands out is the function `lookup`. It returns a reference *into the linked-list* and changing this reference modifies the contents of the list. The challenge with `lookup` is that we need a way to simultaneously describe the ownership of the returned reference but also the ownership of the remaining list. Let us make this problem more precise. A naïve attempt to specify the function would be one of the following two options:

$$\{\mathrm{list}(v, xs) * 0 \le i < |xs|\}\ \mathsf{lookup}(v, i)\ \{w.\ \exists \ell, u. w = \mathsf{Some}(\ell) * \ell \mapsto (x, u)\}$$

$$\{\mathrm{list}(v, xs) * 0 \le i < |xs|\}\ \mathsf{lookup}(v, i)\ \{w.\ \exists \ell. w = \mathsf{Some}(\ell) * \mathrm{list}(v, xs)\}$$

where $|xs|$ denotes the length of the list xs. The first specification yields the ownership of the reference in the postcondition and the second one the ownership of the list. But neither one is satisfying. The first one drops the ownership of the rest of the list such that we can no longer operate on the list (*e.g.,* to compute the length of the list). The second one tells us nothing about the resulting reference, so we have no ownership to dereference or update it.

This is where we use the magic wand $P \mathrel{-\!\!*} Q$. As mentioned above, the magic wand is the notion of implication for the separating conjunction $P * Q$. What makes it interesting is that it can carry ownership itself. For example, consider the magic wand $r \mapsto 0 \mathrel{-\!\!*} (\ell \mapsto 42 * r \mapsto 0)$. It carries the ownership of $\ell \mapsto 42$. More specifically, we can create it by giving up the ownership of $\ell \mapsto 42$ (*i.e.,* $P \vdash Q \mathrel{-\!\!*} (P * Q)$ for $P \triangleq \ell \mapsto 42$ and $Q \triangleq r \mapsto 0$) and we can eliminate it by providing the ownership of $r \mapsto 0$, which then yields $\ell \mapsto 42 * r \mapsto 0$ (*i.e.,* $(Q \mathrel{-\!\!*} R) * Q \vdash R$ for $Q \triangleq r \mapsto 0$ and $R \triangleq \ell \mapsto 42 * r \mapsto 0$).

Using this ability of the magic wand to carry ownership, we can specify the `lookup` function as returning (1) the reference ℓ and (2) ownership of the remainder of the list except for the reference ℓ. We express the latter as a magic wand that, *given the ownership of the reference* ℓ, yields the ownership of the entire list again (where $xs[i]$ denotes the i-th element of xs and $xs[i \mapsto y]$ denotes the result of updating the i-th element to y):

Lemma 8.

$$\begin{array}{l}\{\mathrm{list}(v, xs) * 0 \le i < |xs|\} \\ \quad \mathsf{lookup}(v, i) \\ \{w.\ \exists \ell, u.\ w = \mathsf{Some}(\ell) * \ell \mapsto (xs[i], u) * (\forall y.\ \ell \mapsto (y, u) \mathrel{-\!\!*} \mathrm{list}(v, xs[i \mapsto y]))\}\end{array}$$

Proof. By induction on xs. The base case $xs = []$ is trivial. For $xs = x :: xr$, we know that $v = \mathsf{Some}(r)$ for some location r and we own $r \mapsto (x, w) * \mathrm{list}(w, xr)$. We distinguish two cases. If $i = 0$, then we are returning the current reference r. In this case, we have to prove

$$r \mapsto (x, w) * \mathrm{list}(w, xr) \vdash r \mapsto (x, w) * (\forall y.\ r \mapsto (y, w) \mathrel{-\!\!*} \mathrm{list}(\mathsf{Some}(r), y :: xr))$$

Using sep-split, this means we have to prove

$$\mathrm{list}(w, xr) \vdash (\forall y.\ r \mapsto (y, w) \mathrel{-\!\!*} \mathrm{list}(\mathrm{Some}(r), y :: xr))$$

Here, the magic wand gives us back the ownership of the location r that we need to prove the list predicate for $\mathrm{Some}(r)$.

If $i \neq 0$, we recurse. We frame the ownership of $r \mapsto (x, w)$ around the recursive call. We get from the recursive call the ownership of

$$\ell \mapsto (xr[i-1], u) * (\forall y.\ \ell \mapsto (y, u) \mathrel{-\!\!*} \mathrm{list}(w, xr[i-1 \mapsto y]))$$

and use it to prove

$$\ell \mapsto ((x :: xr)[i], u) * (\forall y.\ \ell \mapsto (y, u) \mathrel{-\!\!*} \mathrm{list}(\mathrm{Some}(r), (x :: xr)[i \mapsto y]))$$

by adding $r \mapsto (x, w)$ again to the ownership of the list. □

With this brief review, we have discussed the most important features of separation logic—adapted to a style that matches Iris. We are now equipped to explore how Iris goes beyond these features.

Chapter 3

The Modern Separation Logic Iris

Iris extends traditional separation logic in several dimensions, which we will discuss in this chapter: the weakest precondition (in §3.1), step-indexing (in §3.2), persistency (in §3.3), invariants (in §3.4), concurrency (in §3.5), and ghost state (in §3.6). To accommodate all of them, we will extend the separation logic from §2 with several additional connectives, explained one by one over the course of this chapter:

$$\text{Propositions}\quad P, Q, R ::= \cdots \mid \mathbf{wp}\ e\ \{v.\, Q(v)\} \mid \triangleright P \mid \square P \mid \boxed{P}^{\mathcal{N}} \mid \boxed{a}^{\gamma} \mid {\models\!\!\!\Rrightarrow} P$$

3.1 The Weakest Precondition

Following the lead of the original separation logic,[1] Iris—and projects building on it—are often presented as program logics in "Hoare triple style" (similar to the presentation in §2): their main judgment is a Hoare triple $\{P\}\ e\ \{v.\, Q(v)\}$ with precondition P and postcondition Q. Indeed this is also the style we will adopt for most of this dissertation. However, Hoare triples are not actually a primitive notion in Iris. Instead, they are defined using a *weakest precondition* $\mathbf{wp}\ e\ \{v.\, Q(v)\}$ (read "the weakest precondition of e for postcondition Q"). The idea is that to prove a Hoare triple $\{P\}\ e\ \{v.\, Q(v)\}$, one proves that the precondition P entails the weakest precondition of e for postcondition Q. Thus, for now,[2] we define Hoare triples as

[1] Reynolds, "Separation Logic: A logic for shared mutable data structures", 2002 [Rey02].

$$\{P\}\ e\ \{v.\, Q(v)\} \qquad \textit{iff} \qquad P \vdash \mathbf{wp}\ e\ \{v.\, Q(v)\}.$$

[2] We will revise this definition in §3.3 to express Hoare triples within the logic.

One of the main advantages of using weakest preconditions—especially in Rocq—is that most reasoning is reduced to proving entailments $P \vdash Q$, for which Iris provides excellent support via the Iris Proof Mode.[3]

[3] Krebbers, Timany, and Birkedal, "Interactive proofs in higher-order concurrent separation logic", 2017 [KTB17].

We reason about the weakest precondition using the proof rules in Fig. 3.1. The rule wp-value allows us to prove a weakest precondition for a value v by proving the postcondition Q, and the rule wp-wand allows us to weaken the postcondition. Importantly, wp-wand uses a *magic wand* $\forall v.\, Q(v) \twoheadrightarrow Q'(v)$, which allows us to derive the frame rule, as we will see below. The rule wp-bind can be used to focus on a subexpression in an evaluation context. It puts the weakest precondition for the remaining expression into the postcondition. The rule wp-pure-step can be used to take pure steps. The rule wp-ref allows us to allocate a new reference. When we apply it, we must prove the postcondition $Q(\ell)$ for a new location ℓ for which we obtain the ownership via the magic wand "$\ell \mapsto v \twoheadrightarrow$". The rule wp-load requires us to provide the ownership of $\ell \mapsto v$ and then allows us to assume it again for proving the postcondition $Q(v)$, Weakest Precondition Rules $\boxed{\mathbf{wp}\ e\ \{v.\, Q(v)\}}$

wp-value
$$Q(v) \vdash \mathbf{wp}\ v\ \{w.\, Q(w)\}$$

wp-pure-step
$$\frac{e \longrightarrow_{\text{pure}} e'}{\mathbf{wp}\ e'\ \{v.\, Q(v)\} \vdash \mathbf{wp}\ e\ \{v.\, Q(v)\}}$$

wp-wand
$$(\forall v.\, Q(v) \mathrel{-\!\!*} Q'(v)) * \mathbf{wp}\ e\ \{w.\, Q(w)\} \vdash \mathbf{wp}\ e\ \{w.\, Q'(w)\}$$

wp-bind
$$\mathbf{wp}\ e\ \big\{v.\, \mathbf{wp}\ K[v]\ \{w.\, Q(w)\}\big\} \vdash \mathbf{wp}\ K[e]\ \{w.\, Q(w)\}$$

wp-ref
$$(\forall \ell.\, \ell \mapsto v \mathrel{-\!\!*} Q(\ell)) \vdash \mathbf{wp}\ \mathsf{ref}(v)\ \{w.\, Q(w)\}$$

wp-load
$$\ell \mapsto v * (\ell \mapsto v \mathrel{-\!\!*} Q(v)) \vdash \mathbf{wp}\ !\,\ell\ \{w.\, Q(w)\}$$

wp-store
$$\ell \mapsto v * (\ell \mapsto w \mathrel{-\!\!*} Q()) \vdash \mathbf{wp}\ \ell := w\ \{u.\, Q(u)\}$$

Figure 3.1: Proof rules for the weakest precondition.

and the rule wp-store requires us to provide $\ell \mapsto v$ and then allows us to assume the updated version $\ell \mapsto w$ for the postcondition.

Let us now derive some of the Hoare rules from §2. We prove the rule for updating references, the rule of consequence, and the frame rule.

Lemma 9 (Hoare Store)**.**

$$\{\ell \mapsto v\}\ \ell := w\ \{\_.\, \ell \mapsto w\}$$

Proof. We apply the rule wp-store for the postcondition $Q(\_) \triangleq \ell \mapsto w$. We are left to prove $\ell \mapsto v \vdash \ell \mapsto v * (\ell \mapsto w \mathrel{-\!\!*} \ell \mapsto w)$. The entailment follows, because the wand is trivially true (*i.e.,* $\mathsf{True} \vdash \ell \mapsto w \mathrel{-\!\!*} \ell \mapsto w$). □

Lemma 10 (Hoare Consequence)**.**

$$\frac{P \vdash P' \qquad \{P'\}\ e\ \{v.\, Q'(v)\} \qquad (\forall v.\, Q'(v) \vdash Q(v))}{\{P\}\ e\ \{v.\, Q(v)\}}$$

Proof. Using the entailment $P \vdash P'$, we show $P' \vdash \mathbf{wp}\ e\ \{v.\, Q(v)\}$. We apply the rule wp-wand, leaving us to prove $P' \vdash (\forall v.\, Q'(v) \mathrel{-\!\!*} Q(v)) * \mathbf{wp}\ e\ \{v.\, Q'(v)\}$. Thus, with sep-split (in Fig. 2.2) and our assumption $\{P'\}\ e\ \{v.\, Q'(v)\}$, it suffices to prove $\mathsf{True} \vdash \forall v.\, Q'(v) \mathrel{-\!\!*} Q(v)$, which follows from $\forall v.\, Q'(v) \vdash Q(v)$. □

Lemma 11 (Hoare Frame)**.**

$$\frac{\{P\}\ e\ \{v.\, Q(v)\}}{\{P * R\}\ e\ \{v.\, Q(v) * R\}}$$

Proof. Unfolding the definition of Hoare triples, we have to prove (after trivial entailment transformations) that $R * \mathbf{wp}\ e\ \{v.\, Q(v)\} \vdash \mathbf{wp}\ e\ \{v.\, Q(v) * R\}$. Applying the rule wp-wand, it suffices to prove

$$R * \mathbf{wp}\ e\ \{v.\, Q(v)\} \vdash (\forall v.\, Q(v) \mathrel{-\!\!*} (Q(v) * R)) * \mathbf{wp}\ e\ \{v.\, Q(v)\}$$

To conclude the proof, we once again use the ability of the magic wand to carry ownership analogous to the lookup function in §2.2. Concretely, we conclude the proof by proving that $R \vdash \forall v.\, Q(v) \mathrel{-\!\!*} (Q(v) * R)$. $\square$

Proof Diagrams. One of the main strengths of the Rocq implementation of Iris is that it offers an interactive proof mode, the Iris Proof Mode (IPM).[4] It helps users of Iris to prove entailments $P \vdash Q$ by turning the premise P into a separation logic context and the conclusion Q into the current proof goal. For detailed Iris proofs, especially in this chapter, we use a similar approach on paper to improve the readability of our proofs with *proof diagrams*. In these diagrams, we distinguish between the current separation logic context (left) and the current goal (right), and we separate different proof steps by horizontal lines. For example, the proof of Lemma 11 as a proof diagram looks as follows:

[4] Krebbers, Timany, and Birkedal, "Interactive proofs in higher-order concurrent separation logic", 2017 [KTB17].

Lemma 12 (Hoare Frame with a Proof Diagram)**.**

$$\frac{\{P\}\ e\ \{v.\, Q(v)\}}{\{P * R\}\ e\ \{v.\, Q(v) * R\}}$$

Proof. Suppose $\{P\}\ e\ \{v.\, Q(v)\}$, so $P \vdash \mathbf{wp}\ e\ \{v.\, Q(v)\}$.

Context	Goal
$P * R$	$\mathbf{wp}\ e\ \{v.\, Q(v) * R\}$
Using the assumption and commutativity.	
$R * \mathbf{wp}\ e\ \{v.\, Q(v)\}$	$\mathbf{wp}\ e\ \{v.\, Q(v) * R\}$
Using wp-wand	
$R * \mathbf{wp}\ e\ \{v.\, Q(v)\}$	$(\forall v.\, Q(v) \mathrel{-\!\!*} (Q(v) * R)) * \mathbf{wp}\ e\ \{v.\, Q(v)\}$
Using sep-split, it suffices to prove	
R	$\forall v.\, Q(v) \mathrel{-\!\!*} (Q(v) * R)$
Introducing the universal quantifier and the wand, we are left to prove	
$R * Q(v)$	$Q(v) * R$
which is trivial.	$\square$

3.2 Step-Indexing

Next, let us turn to *step-indexing*.[5] As mentioned in the introduction (see §1), step-indexing is a powerful technique for recursive, seemingly cyclic reasoning. It enables users of Iris to reason about recursive functions, verify generic higher-order specifications, and build semantic models of languages with advanced features such as shared, mutable, higher-order state. In the following, we will focus first on step-indexing from a user perspective, especially on how it affects the program logic via the *later modality* $\triangleright P$ (read "later P"). We will then see how it affects the model of Iris and what is behind the later modality in §4.

As mentioned above, step-indexing enables different forms of recursive reasoning. The first one that we will focus on is how it enables us to verify recursive functions.[6] That is, up to now, all the examples that we have discussed are terminating. But recall (from §2) that we are only aiming for *partial correctness*, meaning programs are allowed to diverge. Step-indexing will help us to reflect this into our reasoning principles. To keep matters concrete, we focus on a specific example, a potentially non-terminating search procedure.

[5] Appel and McAllester, "An indexed model of recursive types for foundational proof-carrying code", 2001 [AM01]; Ahmed et al., "Semantic foundations for typed assembly languages", 2010 [Ahm+10].

[6] Step-indexing is not strictly necessary to verify recursive functions. For non-step-indexed partial correctness logics, there are other approaches that one could choose to verify potentially non-terminating functions such as co-induction. We use recursion here to provide a simple introduction to step-indexing. We consider more advanced applications such as impredicative invariants later in this chapter (in §3.4) and in subsequent parts of the dissertation.

Example 13 (The First Function)**.** The function `first` searches for the first natural number satisfying a predicate p. It does so by starting at n and then counting up (potentially diverging if p is never satisfied).

$$\texttt{first}\, p\, x \triangleq \mathsf{if}\, p\, x\, \mathsf{then}\, x\, \mathsf{else}\, \texttt{first}\, p\, (x+1)$$

Suppose that p computes a pure predicate ϕ. Our goal is to show that $\texttt{first}\, p\, n$ computes the first number greater than (or equal) n satisfying ϕ. Concretely, assuming $\{\mathsf{True}\}\, p\, n\, \{v.\, \exists b : \mathbb{B}.\, v = b * (b = \mathsf{true} \Leftrightarrow \phi(n))\}$, we want to show

$$\{\mathsf{True}\}\, \texttt{first}\, p\, n\, \{v.\, \exists m.\, v = m * \phi(m) * \forall k.\, n \le k < m \Rightarrow \neg\phi(k)\}$$

•

The later modality. The key to our recursive argument for `first` will be the *later modality* $\triangleright P$. Intuitively, it means that P will hold after the next step of computation. To reason about it, we use the following proof rules:

Later Rules $\boxed{\triangleright P}$

later-intro
$$P \vdash \triangleright P$$

later-mono
$$\frac{P \vdash Q}{\triangleright P \vdash \triangleright Q}$$

Löb
$$\frac{\triangleright P \vdash P}{\vdash P}$$

later-sep
$$\triangleright(P * Q) \dashv\vdash \triangleright P * \triangleright Q$$

later-exists
$$\frac{X \text{ non-empty}}{\triangleright(\exists x : X.\, P(x)) \dashv\vdash \exists x : X.\, \triangleright P(x)}$$

later-all
$$\triangleright(\forall x : X.\, P(x)) \dashv\vdash \forall x : X.\, \triangleright P(x)$$

wp-later-pure-step
$$\frac{e \longrightarrow_{\mathsf{pure}} e'}{\triangleright\, \mathbf{wp}\, e'\, \{v.\, P(v)\} \vdash \mathbf{wp}\, e\, \{v.\, P(v)\}}$$

wp-later-ref
$$\triangleright(\forall \ell.\, \ell \mapsto v \mathrel{-\!\!*} Q(\ell)) \vdash \mathbf{wp}\, \mathsf{ref}(v)\, \{w.\, Q(w)\}$$

wp-later-load
$$\ell \mapsto v * \triangleright(\ell \mapsto v \mathrel{-\!\!*} Q(v)) \vdash \mathbf{wp}\, !\,\ell\, \{w.\, Q(w)\}$$

wp-later-store
$$\ell \mapsto v * \triangleright(\ell \mapsto w \mathrel{-\!\!*} Q()) \vdash \mathbf{wp}\, \ell := w\, \{w.\, Q(w)\}$$

Figure 3.2: Proof rules for the later modality.

We can always introduce a later modality (later-intro); we can apply entailments underneath a later modality (later-mono); and it commutes with separating conjunction (later-sep), existential quantification (later-exists), and universal quantification (later-all). The later modality is connected to program steps by rules for the weakest precondition (wp-later-pure-step, wp-later-ref, wp-later-load, and wp-later-store). Each of these rules allows us to continue after the next program step (*e.g.*, after the next pure step in wp-later-pure-step) with the remaining goal *underneath a later modality*. They are strengthened versions of the weakest precondition rules from above. For example, we can derive the rule wp-pure-step from wp-later-pure-step as follows:

Lemma 14.

$$\frac{e \longrightarrow_{\text{pure}} e'}{\textbf{wp}\; e' \;\{v.\, Q(v)\} \vdash \textbf{wp}\; e \;\{v.\, Q(v)\}}$$

Proof. Suppose $e \longrightarrow_{\text{pure}} e'$.

CONTEXT	GOAL
$\textbf{wp}\; e' \;\{v.\, Q(v)\}$	$\textbf{wp}\; e \;\{v.\, Q(v)\}$

With LATER-INTRO, it suffices to prove

$\textbf{wp}\; e' \;\{v.\, Q(v)\}$	$\triangleright\, \textbf{wp}\; e \;\{v.\, Q(v)\}$

which follows immediatelly with WP-LATER-PURE-STEP. □

Löb induction. The key rule that we will use to justify recursive reasoning is LÖB. It means that if we want to prove P, then it suffices to prove P under the assumption that P already holds later. We put it to use for verifying `first`:

Lemma 15. *If* $\{\mathsf{True}\}\; p\; n\; \{v.\, \exists b : \mathbb{B}.\, v = b * (b = \mathsf{true} \Leftrightarrow \phi(n))\}$*, then*

$$\{\mathsf{True}\}\; \texttt{first}\; p\; n\; \{v.\, \exists m.\, v = m * \phi\,(m) * \forall k.\, n \le k < m \Rightarrow \neg\phi(k)\}$$

Proof. Suppose $\{\mathsf{True}\}\; p\; n\; \{v.\, \exists b : \mathbb{B}.\, v = b * (b = \mathsf{true} \Leftrightarrow \phi(n))\}$.

CONTEXT	GOAL
	$\forall n.\, \textbf{wp}\; \texttt{first}\; p\; n\; \{v.\, \Phi_n(v)\}$

where we abbreviate $\Phi_n(v) \triangleq \exists m.\, v = m * \phi\,(m) * \forall k.\, n \le k < m \Rightarrow \neg\phi(k)$.

We proceed by LÖB induction.

$\triangleright(\forall n.\, \textbf{wp}\; \texttt{first}\; p\; n\; \{v.\, \Phi_n(v)\})$	$\forall n.\, \textbf{wp}\; \texttt{first}\; p\; n\; \{v.\, \Phi_n(v)\}$

We introduce n and execute one pure step with WP-LATER-PURE-STEP. (Note that `first` is a curried function, so one pure step leaves us with a λ-abstraction.)

$\triangleright(\forall n.\, \textbf{wp}\; \texttt{first}\; p\; n\; \{v.\, \Phi_n(v)\})$

$\triangleright\, \textbf{wp}\; (\lambda x.\, \mathsf{if}\; p\; x\; \mathsf{then}\; x\; \mathsf{else}\; \texttt{first}\; p\; (x+1))\; n\; \{v.\, \Phi_n(v)\}$

We use LATER-MONO to strip the later from the context and the goal.

$\forall n.\, \textbf{wp}\; \texttt{first}\; p\; n\; \{v.\, \Phi_n(v)\}$

$\textbf{wp}\; (\lambda x.\, \mathsf{if}\; p\; x\; \mathsf{then}\; x\; \mathsf{else}\; \texttt{first}\; p\; (x+1))\; n\; \{v.\, \Phi_n(v)\}$

After another pure step, we focus on the execution of $p\; n$ with WP-BIND.

$\forall n.\, \textbf{wp}\; \texttt{first}\; p\; n\; \{v.\, \Phi_n(v)\}$

$\textbf{wp}\; p\; n\; \big\{w.\, \textbf{wp}\; \mathsf{if}\; w\; \mathsf{then}\; n\; \mathsf{else}\; \texttt{first}\; p\; (n+1)\; \{v.\, \Phi_n(v)\}\big\}$

Using our assumption about p, we obtain a Boolean b such that

$(\forall n.\, \textbf{wp}\; \texttt{first}\; p\; n\; \{v.\, \Phi_n(v)\}) * (b = \mathsf{true} \Leftrightarrow \phi(n))$

$\textbf{wp}\; (\mathsf{if}\; b\; \mathsf{then}\; n\; \mathsf{else}\; \texttt{first}\; p\; (n+1))\; \{v.\, \Phi_n(v)\}$

We proceed by case analysis on b.

Case $b = \mathsf{true}$.

Then we have to prove (after executing pure steps):

$(\forall n.\, \textbf{wp}\; \texttt{first}\; p\; n\; \{v.\, \Phi_n(v)\}) * \phi(n)$	$\Phi_n(n)$

which is trivially true (since $\phi(n)$ and there is no k such that $n \le k < n$).

Case $b = \mathsf{false}$.

Then we have to prove (after executing pure steps):

$(\forall n.\, \textbf{wp}\; \texttt{first}\; p\; n\; \{v.\, \Phi_n(v)\}) * \neg\phi(n)$	$\textbf{wp}\; \texttt{first}\; p\; (n+1)\; \{v.\, \Phi_n(v)\}$

We apply the inductive hypothesis for $n + 1$ using WP-WAND, leaving

$\neg\phi(n)$	$\forall v.\, \Phi_{n+1}(v) \mathrel{-\!\!*} \Phi_n(v)$

The claim follows trivially, since ϕ does not hold for n. □

More step-indexing. Löb induction is just the tip of the iceberg for step-indexing. We will introduce several more use cases of step-indexing in the remainder of this dissertation. For example, in Part II, we will use step-indexing to verify a higher-order stateful combinator (§8.1) and to define a logical relation for proving termination of a language with asynchronous channels (§8.3). In Part III, we will use step-indexing to define a binary logical relation for reasoning about reorderable expressions (§13.1), to verify a fine-grained concurrent data structure (§13.2), and to obtain fixpoints of recursive predicates (§14.2). In Part V, we will use it to verify a higher-order channel implementation (§29.1). (In many of these applications, the use of step-indexing will be *implicit*. It will be concealed behind Iris's *impredicative invariants*, which we will introduce in §3.4.1.)

3.3 Persistency

Next, let us turn to persistency. Most separation logic assertions cannot be duplicated because they carry ownership (*e.g.*, $\ell \mapsto v$ carries exclusive ownership of ℓ, so it cannot be duplicated). We will now introduce a class of propositions, *persistent propositions*, whose main characteristic is that they can be duplicated. Thus, just like propositions in first-order logic or Rocq, we can reuse them as often as we want. To motivate them, let us consider a concrete example:

Example 16 (Fibonacci)**.** We have seen above how to use Löb induction to reason about recursive functions. But now suppose we want to verify the Fibonacci-function (or any other function with multiple recursive occurrences):

$$\texttt{fib}(n) \triangleq \textsf{if}\; n \leq 1 \;\textsf{then}\; n \;\textsf{else}\; \texttt{fib}(n-1) + \texttt{fib}(n-2)$$

Then, without persistency, we will get stuck in the recursive case, because we can only discharge *one* of the two recursive occurrences. More specifically, suppose we want to prove the specification $\{\textsf{True}\}\ \texttt{fib}(n)\ \{v.\ v \in \mathbb{Z}\}$ (where, for simplicity, we only prove that the result is an integer). Analogously to the proof of `first` (in Lemma 15), we eventually reach the following proof state:

Context Goal

$\vdots$

Case $n \geq 2$.
$(\forall n.\ \textbf{wp}\ \texttt{fib}(n)\ \{v.\ v \in \mathbb{Z}\})$

$$\textbf{wp}\ \texttt{fib}(n-1) + \texttt{fib}(n-2)\ \{v.\ v \in \mathbb{Z}\}$$

We bind on $\texttt{fib}(n-2)$ and apply the inductive hypothesis with wp-wand.

$$\forall v.\ v \in \mathbb{Z} \mathrel{-\!\!*} \textbf{wp}\ \texttt{fib}(n-1) + v\ \{w.\ w \in \mathbb{Z}\}$$

But here we are stuck! We have used up the inductive hypothesis to evaluate $\texttt{fib}(n-2)$, so we cannot use it again to evaluate the *second* recursive call. To make this proof work, we must strengthen the inductive hypothesis to allow us to use it more than once. This is where persistency comes in.[7] •

[7] In general, a weakest precondition $\textbf{wp}\ e\ \{v.\ Q(v)\}$ cannot be duplicated, because it can carry ownership just like a magic wand. For example, we can prove $\ell \mapsto v \vdash \textbf{wp}\ !\ell\ \{w.\ w = v * \ell \mapsto v\}$. If we could duplicate the weakest precondition, then we could use two loads of ℓ to duplicate the ownership of $\ell \mapsto v$ (that we obtain from the postcondition).

The persistency modality. To characterize persistency, we introduce a new modality $\square P$ (read "persistently P"), and we call a proposition P *persistent* (written $\textsf{persistent}(P)$) if we can prove $P \vdash \square P$. We use the rules in Fig. 3.3 for reasoning about $\square P$. The persistency modality is monotone (pers-mono);

Persistency Rules $\boxed{\Box P}$

pers-mono
$$\frac{P \vdash Q}{\Box P \vdash \Box Q}$$

pers-idemp
$$\Box P \vdash \Box \Box P$$

pers-all
$$\forall x : X.\ \Box P(x) \vdash \Box \forall x : X.\ P(x)$$

pers-exists
$$\Box \exists x : X.\ P(x) \vdash \exists x : X.\ \Box P(x)$$

pers-later
$$\triangleright \Box P \dashv\vdash \Box \triangleright P$$

pers-pure
$$\phi \vdash \Box \phi$$

pers-elim
$$\Box P \vdash P$$

pers-dupl
$$\Box P \vdash (\Box P) * (\Box P)$$

pers-and-sep
$$(\Box P) \wedge Q \dashv\vdash (\Box P) * Q$$

Figure 3.3: Proof rules for the persistency modality $\Box P$.

idempotent (pers-idemp); and it commutes with universal quantification (pers-all), existential quantification (pers-exists), and the later modality (pers-later). Pure propositions are persistent (pers-pure) and we can always eliminate the persistency modality (pers-elim). The key property of persistent propositions is that we can *duplicate* them (pers-dupl), which can be expressed in a strengthened form as conjunction and separating conjunction coincide for persistent propositions (pers-and-sep).[8]

Note that there is no introduction rule for the persistency modality (other than for pure propositions with pers-pure). The typical way to introduce a persistency modality is with pers-mono. It requires the premise of the entailment to already be persistent. As a result, we can only use persistent assertions to prove persistent assertions.

[8] The ordinary conjunction $P \wedge Q$ is used rarely with separation logic resources. It means that P and Q are true at the same time, but if we want to use them (unless they are persistent), we must decide between one of the two conjuncts. In other words, we cannot turn a conjunction (in general) into a separating conjunction.

Multiple recursive occurrences. With the persistency modality in hand, let us return to the Fibonacci example. Instead of proving the entailment $\vdash \forall n.\ \mathbf{wp}\ \mathtt{fib}(n)\ \{v.\ v \in \mathbb{Z}\}$, we now prove $\vdash \Box \forall n.\ \mathbf{wp}\ \mathtt{fib}(n)\ \{v.\ v \in \mathbb{Z}\}$:

Lemma 17.
$$\vdash \Box\, \mathbf{wp}\ \mathtt{fib}(n)\ \{v.\ v \in \mathbb{Z}\}$$

Proof Sketch. We focus on how persistency allows us to continue where we would have been stuck before.

Context | Goal

$\vdots$

Case $n \geq 2$.

$\Box \forall n.\ \mathbf{wp}\ \mathtt{fib}(n)\ \{v.\ v \in \mathbb{Z}\}$

$\mathbf{wp}\ \mathtt{fib}(n-1) + \mathtt{fib}(n-2)\ \{v.\ v \in \mathbb{Z}\}$

Here, we benefit from persistency. We duplicate the inductive hypothesis with pers-dupl and eliminate one persistency modality with pers-elim.

$(\Box \forall n.\ \mathbf{wp}\ \mathtt{fib}(n)\ \{v.\ v \in \mathbb{Z}\}) * (\forall n.\ \mathbf{wp}\ \mathtt{fib}(n)\ \{v.\ v \in \mathbb{Z}\})$

$\mathbf{wp}\ \mathtt{fib}(n-1) + \mathtt{fib}(n-2)\ \{v.\ v \in \mathbb{Z}\}$

We bind on $\mathtt{fib}(n-2)$ and apply the inductive hypothesis with wp-wand.

$\Box \forall n.\ \mathbf{wp}\ \mathtt{fib}(n)\ \{w.\ w \in \mathbb{Z}\}$

$\forall v.\ v \in \mathbb{Z} \mathrel{-\!\!*} \mathbf{wp}\ \mathtt{fib}(n-1) + v\ \{w.\ w \in \mathbb{Z}\}$

After introducing $v \in \mathbb{Z}$, we are left to prove

$(\Box \forall n.\ \mathbf{wp}\ \mathtt{fib}(n)\ \{w.\ w \in \mathbb{Z}\}) * v \in \mathbb{Z}$ $\quad$ $\mathbf{wp}\ \mathtt{fib}(n-1) + v\ \{w.\ w \in \mathbb{Z}\}$

Now we can use the inductive hypothesis *again* to execute $\mathtt{fib}(n-1)$. The remainder of the proof is trivial. □

Internalizing Hoare triples. Verifying a weakest precondition underneath a persistency modality is a common pattern, because often we want to reuse the resulting specification multiple times. In fact, we can use this pattern to *internalize* the definition of Hoare triples to turn them into Iris propositions. The internalized Hoare triples are more powerful. As we will see shortly, they have two main benefits: (1) they can be used for specifying expressions whose resulting values are functions (because the new triples can be used in postconditions), and (2) they can use invariants (see §3.4).

To internalize Hoare triples, we define

$$\{P\}\ e\ \{v.\, Q(v)\} \triangleq \Box(P \mathrel{-\!\!*} \mathbf{wp}\ e\ \{v.\, Q(v)\})$$

Compared to the "old" Hoare triples $\{P\}\ e\ \{v.\, Q(v)\}_{\text{old}} \triangleq P \vdash \mathbf{wp}\ e\ \{v.\, Q(v)\}$ from §3.1, the new triples are no longer meta-level judgments (*i.e.,* they are no longer judgments of Rocq type *Prop*) but instead are Iris propositions (like $P * Q$). Thus, they can be used, for example, in postconditions. All Hoare triple results that we have proven for the "old" triples remain true, because

$$\vdash \{P\}\ e\ \{v.\, Q(v)\} \quad \textit{iff.} \quad P \vdash \mathbf{wp}\ e\ \{v.\, Q(v)\}$$

Our Hoare triples are trivially duplicable $\{P\}\ e\ \{v.\, Q(v)\} \vdash \{P\}\ e\ \{v.\, Q(v)\} * \{P\}\ e\ \{v.\, Q(v)\}$, and we can prove the following general reasoning principle for recursive functions:

Lemma 18.

$$\frac{(\forall u. \{P\, u\}(\mathsf{fix}\, f\, x.e)\, u\{w. Q(u,w)\}) \vdash \forall v. \{P\, v\} e[(\mathsf{fix}\, f\, x.e)/f, v/x]\{w.\, Q(v,w)\}}{P\, v \vdash \mathbf{wp}\ (\mathsf{fix}\, f\, x.e)\, v\ \{w.\, Q(v,w)\}}\ \textit{WP-REC}$$

Proof. By Löb induction analogous to the proofs of `first` and `fib`. □

3.4 Invariants

Next, let us turn to *invariants*. Invariants are one of the most important persistent assertions. They allow us to share ownership of resources (*e.g.,* points-to assertions $\ell \mapsto v$) between different parts of a program. To keep matters concrete, we start with an example.

Example 19 (MutBit)**.** Our goal is to verify the following simple data structure:

$$\texttt{MutBit} \triangleq \mathsf{let}\ x = \mathsf{ref}\ (0)\ \mathsf{in}\ \{\texttt{flip} \triangleq \lambda\_.\, x := 1 - !x,\ \texttt{get} \triangleq \lambda\_.\, !x\}$$

It allocates a new reference x that will alternate between 0 and 1 and provides two operations for accessing x: `flip` flips the internal value of the reference and `get` retrieves its value. Our goal will be to prove the following specification:

$$\{\mathsf{True}\}\ \texttt{MutBit}\ \{v.\, \mathsf{mutbit}(v)\} \quad \textit{where}$$

$\mathsf{mutbit}(v) \triangleq \{\mathsf{True}\}\ v.\texttt{flip}()\ \{w.\, w = ()\} * \{\mathsf{True}\}\ v.\texttt{get}()\ \{w.\, w = 0 \vee w = 1\}$. There is no precondition for `flip` and `get`, and `get` always returns 0 or 1. •

`MutBit` is the first example whose return value contains (a record of) functions. To give clients of `MutBit` the ability to use these functions, we put specifications for them into the postcondition—using the internalized version of Hoare triples. And since the reference x is captured internally by the two functions and not observably externally, we do not expose it via the specification. Instead, both `flip` and `get` have precondition True.

Implicit sharing. This specification simplifies the life of clients of `MutBit` in the sense that they do not need to thread around ownership of a reference, which they do not have access to. At the same time, it poses a challenge for verifying `MutBit`: the preconditions do not pass ownership of x to `flip` and `get`, so where do we get the ownership of x from to verify `flip` and `get`? More specifically, after allocating the reference x, we have to prove:

$$x \mapsto 0 \vdash \{\mathsf{True}\}\ bit.\texttt{flip}()\ \{w.\ w = ()\} * \{\mathsf{True}\}\ bit.\texttt{get}()\ \{w.\ w = 0 \vee w = 1\}$$
$$\text{where}\quad bit = \{\texttt{flip} \triangleq \lambda\_.\ x := 1 - {!}\,x,\quad \texttt{get} \triangleq \lambda\_.\ {!}\,x\}$$

How are we supposed to decide whether to give ownership of $x \mapsto 0$ to `flip` or `get`? Moreover, even if we pick one side, then we subsequently have to prove a persistent proposition (recall $\{P\}\ e\ \{v.\ Q(v)\} \triangleq \Box(P \mathrel{-\!\!*} \mathbf{wp}\ e\ \{v.\ Q(v)\})$). But the resource $x \mapsto 0$ is not persistent,[9] so we cannot keep it for proving anything underneath the persistency modality.

[9] $x \mapsto 0$ cannot be persistent, because we are not allowed to duplicate it (points-to-sep).

Here, invariants $\boxed{P}^{\mathcal{N}}$ come in. With invariants, we can make the ownership of x duplicable. The price we have to pay is that we have to agree on a "protocol" how x will be used. Concretely, in the case of `MutBit`, we know that x will always be either 0 or 1. That is what we choose as the invariant:

$$I_{\texttt{MutBit}} \triangleq \boxed{\exists n \in \{0,1\}.\ x \mapsto n}^{\mathcal{N}}$$

Invariants. To understand how invariants work and why they help us here, we consider their rules:

$$\textsc{inv-pers}\quad \boxed{R}^{\mathcal{N}} \vdash \Box\,\boxed{R}^{\mathcal{N}} \qquad\qquad \textsc{wp-inv-alloc}\quad \frac{P * \boxed{R}^{\mathcal{N}} \vdash \mathbf{wp}^{\mathcal{E}}\ e\ \{v.\ Q(v)\}}{P * R \vdash \mathbf{wp}^{\mathcal{E}}\ e\ \{v.\ Q(v)\}}$$

$$\textsc{wp-inv-open-timeless}\quad \frac{P * R \vdash \mathbf{wp}^{\mathcal{E}\setminus\mathcal{N}}\ e\ \{v.\ R * Q(v)\} \qquad \mathsf{atomic}(e) \qquad \mathcal{N} \subseteq \mathcal{E} \qquad \mathsf{timeless}(R)}{P * \boxed{R}^{\mathcal{N}} \vdash \mathbf{wp}^{\mathcal{E}}\ e\ \{v.\ Q(v)\}}$$

Figure 3.4: Proof rules for invariants.

The rule inv-pers says that invariants are persistent and, hence, we can duplicate them and share them between different parts of our program. The rule wp-inv-alloc says that we can allocate the invariant $\boxed{R}^{\mathcal{N}}$ if we give up ownership of R. Thereafter, we can freely share the ownership of R by sharing the invariant $\boxed{R}^{\mathcal{N}}$.

The rule wp-inv-open-timeless is quite a mouthful. Let us break it down. It says that if we own the invariant $\boxed{R}^{\mathcal{N}}$, then we can get access to R. In exchange, we have to prove R again in our postcondition. The reason why we need to re-establish R is that other program parts could rely on the invariant being true when they are executed. For example, both `flip` and `get` will rely on the invariant $I_{\texttt{MutBit}}$ and, hence, they have to ensure that it also holds again after their execution. Let us now turn to the side conditions of the rule: (1) The condition $\mathsf{atomic}(e)$ ensures that invariants can only be opened around *atomic expressions*, expressions that will reduce in one step to a value (*e.g.*, $!\,\ell$, $\ell := v, \ldots$). The reason, as we will see in §3.5, is to be compatible with adding concurrency to our language. (2) The condition $\mathcal{N} \subseteq \mathcal{E}$ uses "masks" $\mathcal{E}$ and "namespaces" $\mathcal{N}$ to ensure that invariants cannot be opened twice. We explain how both work below. (3) The condition $\mathsf{timeless}(R)$ is an artifact of step-indexing. We will come back to it in §3.4.1. For now, it suffices to know that our `MutBit`-invariant, $\exists n \in \{0,1\}.\ x \mapsto n$, is timeless.

Masks. To be sound, invariants introduce an additional piece of bookkeeping, *masks* $\mathcal{E}$ and *namespaces* $\mathcal{N}$. They ensure that we may not open the same invariant while it is already open. For example, if we have the invariant $I_{\texttt{MutBit}}$, then opening it twice would be fatal, because we could use points-to-sep to prove anything. To prevent this from happening, invariants $\boxed{R}^{\mathcal{N}}$ have a so-called namespace $\mathcal{N}$ associated with them and the weakest precondition has a mask $\mathcal{E}$. Whenever we open an invariant, we must make sure that it is not already opened by proving that the namespace $\mathcal{N}$ is still contained in the mask. Subsequently, the namespace is removed from the mask until the invariant is closed again (*i.e.,* in the postcondition). (Namespaces are more or less like names of invariants, and the masks track which invariants are currently closed.)

In practice, making sure that invariants are not opened twice is pretty straightforward. The masks and namespaces mechanism does, however, introduce a considerable amount of bookkeeping, especially on paper. Thus, to ease the presentation in the following, we will often omit masks, unless the interaction with invariants is particularly interesting. All the rules that we have seen so far for the weakest precondition also hold true parameterized by a mask (*e.g.,* $\mathbf{wp}^{\mathcal{E}}\ e\ \{v.\ \mathbf{wp}^{\mathcal{E}}\ K[v]\ \{w.\ Q(w)\}\} \vdash \mathbf{wp}^{\mathcal{E}}\ K[e]\ \{w.\ Q(w)\}$). We typically start with the full mask $\top$ in examples, but—other than in the `MutBit` example below—usually elide it.

Verifying MutBit. With invariants in hand, let us return to `MutBit`:

Lemma 20. $\{\mathsf{True}\}\ \texttt{MutBit}\ \{v.\ \mathsf{mutbit}(v)\}$

Proof.

Context	Goal
	$\mathbf{wp}^{\top}\ \texttt{MutBit}\ \{v.\ \mathsf{mutbit}(v)\}$
$x \mapsto 0$	$\mathbf{wp}^{\top}\ \{\texttt{flip} \triangleq \lambda\_.\ x := 1 - !x,\ \ \texttt{get} \triangleq \lambda\_.\ !x\}\ \{v.\ \mathsf{mutbit}(v)\}$
We allocate $\exists n \in \{0,1\}.\ x \mapsto n$ as an invariant with wp-inv-alloc.	
$I_{\texttt{MutBit}}$	$\mathbf{wp}^{\top}\ \{\texttt{flip} \triangleq \lambda\_.\ x := 1 - !x,\ \ \texttt{get} \triangleq \lambda\_.\ !x\}\ \{v.\ \mathsf{mutbit}(v)\}$
Let $bit \triangleq \{\texttt{flip} \triangleq \lambda\_.\ x := 1 - !x,\ \ \texttt{get} \triangleq \lambda\_.\ !x\}$.	
$I_{\texttt{MutBit}}$	$\mathsf{mutbit}(bit)$
We proceed with the two conjuncts, using the fact that $I_{\texttt{MutBit}}$ is duplicable.	
Case `get`.	
$I_{\texttt{MutBit}}$	$\mathbf{wp}^{\top}\ bit.\texttt{get}()\ \{w.\ w = 0 \lor w = 1\}$
$I_{\texttt{MutBit}}$	$\mathbf{wp}^{\top}\ !x\ \{w.\ w = 0 \lor w = 1\}$
We open the invariant with wp-inv-open-timeless. Let $\Phi(w) \triangleq w = 0 \lor w = 1$.	
$I_{\texttt{MutBit}} * (\exists n \in \{0,1\}.\ x \mapsto n)$	$\mathbf{wp}^{\top \setminus \mathcal{N}}\ !x\ \{w.\ (\exists n \in \{0,1\}.\ x \mapsto n) * \Phi(w)\}$
We use the ownership of x to justify the read with wp-load.	
$I_{\texttt{MutBit}} * n \in \{0,1\}$	$x \mapsto n \mathrel{-\!\!*} ((\exists n \in \{0,1\}.\ x \mapsto n) * \Phi(n))$
Thus, after giving back the ownership of x, we are left to prove	
$I_{\texttt{MutBit}} * n \in \{0,1\}$	$\Phi(n)$
which is trivial.	
Case `flip`.	
The case for `flip` is analogous, but we have to open the invariant *twice*: once for the load $!x$, where we obtain the current value $n \in \{0,1\}$ and once for the store $x := 1 - n$, where we use the fact that $1 - n \in \{0,1\}$ to close it. □	

3.4.1 Impredicative Invariants

Invariants in separation logic are nothing new. Even early versions of concurrent separation logic[10] already had invariants to share resources between threads. What is special about Iris's invariants (inherited from iCAP,[11] one of its predecessors) is that they are *impredicative*: they can contain any arbitrary Iris proposition, including other invariants and even weakest preconditions. Unfortunately, the price of their expressivity is that their model is cyclic—cyclic to the extent that naive models of $\boxed{R}^{\mathcal{N}}$ are not well-founded (*i.e.,* inductive or co-inductive definitions do not suffice).

[10] O'Hearn, "Resources, concurrency, and local reasoning", 2007 [OHe07].

[11] Svendsen and Birkedal, "Impredicative concurrent abstract predicates", 2014 [SB14].

We will discuss this point in more detail later on (see §3.4.1). For now, it suffices to know that the model of Iris's propositions roughly looks as follows:

$$\mathit{iProp} = \mathit{Inv} \to \mathit{Heap} \to \mathit{Prop} \qquad\qquad \mathit{Inv} = \mathbb{N} \xrightarrow{\text{fin}} \mathit{iProp}$$

To a first approximation, Iris propositions are predicates over invariants and program heaps.[12] Invariants, in turn, are finite maps of Iris propositions.

[12] As we will see in §4.3, Iris propositions are predicates over step-indices and resources. Both the heaps and the invariants are turned into resources.

Clearly, this is a cyclic dependency: to define Iris propositions, we need Iris propositions in the definition of invariants. Unfortunately, this definition is not only recursive, but also has a negative occurrence. In essence, we define *iProp* as predicates over *iProp*, which has no solution in set or type theory. Here, *step-indexing* comes to the rescue. Behind the scenes, in the model of Iris, step-indexing is used to define the type of Iris propositions and justify impredicative invariants.

To users of Iris, this use of step-indexing is concealed, but it becomes visible when interacting with invariants. In particular, Iris has a more general invariant opening rule than wp-inv-open-timeless, which is

$$\frac{\begin{array}{c}\text{wp-inv-open}\\ P * \triangleright R \vdash \mathbf{wp}^{\mathcal{E}\setminus\mathcal{N}}\, e\, \{v.\, \triangleright R * Q(v)\} \qquad \mathsf{atomic}(e) \qquad \mathcal{N} \subseteq \mathcal{E}\end{array}}{P * \boxed{R}^{\mathcal{N}} \vdash \mathbf{wp}^{\mathcal{E}}\, e\, \{v.\, Q(v)\}}$$

Compared to wp-inv-open-timeless, it removes the timelessness requirement. In exchange, the contents R of the invariant are now *guarded by a later modality* as an artifact of the step-indexed model. (When we re-establish the invariant, it also suffices to provide R underneath a later modality.)

Example 21. To see where the full power of impredicative invariants is useful, let us consider an example, *higher-order state* (*i.e.,* memory storing functions). In the following example, we create a reference l that always stores functions that produce an integer. We then return a record that contains two operations: one to update the function stored in the reference and one to execute it.

$$\texttt{FuncRef} \triangleq \mathsf{let}\, l = \mathsf{ref}\,(\lambda\_.\, 0)\ \mathsf{in}\ \{\texttt{store} \triangleq \lambda f.\, l := f,\ \texttt{exec} \triangleq \lambda x.\, (!\, l)(x)\}$$

Following the same style as `MutBit`, a natural specification for `FuncRef` is

$$\{\mathsf{True}\}\ \texttt{FuncRef}\ \{v.\, \mathsf{funcref}(v)\} \quad \textit{where}$$

$$\begin{aligned}\mathsf{funcref}(v) \triangleq\ & \{\{\mathsf{True}\}\, f\,()\, \{w.\, w \in \mathbb{Z}\}\}\ v.\texttt{store}(f)\ \{u.\, u = ()\}\\ & * \{\mathsf{True}\}\ v.\mathsf{exec}()\ \{w.\, w \in \mathbb{Z}\}\end{aligned}$$

The `store`-projection takes a function that will return some integer (and stores it internally in the reference). The `exec`-projection returns some integer (by executing the function currently stored in the reference). •

We will not discuss the proof of this example in detail, because it is analogous to the `MutBit`-example (Example 19). But let us focus on the key invariant that is needed to establish the proof:

$$I_{\mathsf{FuncRef}} \triangleq \boxed{\exists f.\, l \mapsto f * \{\mathsf{True}\}\, f()\, \{v.\, v \in \mathbb{Z}\}}^{\mathcal{N}}$$

It ensures that l always stores functions, which will return an integer. To define it, we make use of impredicativity: we put a Hoare triple into the invariant, because—as the reference is updated—we must update the triple for the function that is stored inside the reference.

Timelessness. Let us briefly discuss the timelessness constraint in wp-inv-open-timeless. In the `MutBit`-example, we are working with a simple invariant, which does not rely on impredicativity at all. For those cases, Iris has a class of propositions, the *timeless propositions*, which do not depend on step-indexing. That is, if a proposition P is timeless, then we can just eliminate a later modality in front of it when proving a weakest precondition. Specifically, timeless propositions satisfy the following rules:

$$\textsc{timeless-pure}\quad \mathsf{timeless}(\phi) \qquad \textsc{timeless-pers}\quad \frac{\mathsf{timeless}(P)}{\mathsf{timeless}(\Box\, P)} \qquad \textsc{timeless-sep}\quad \frac{\mathsf{timeless}(P) \qquad \mathsf{timeless}(Q)}{\mathsf{timeless}(P * Q)}$$

$$\textsc{timeless-wand}\quad \frac{\mathsf{timeless}(Q)}{\mathsf{timeless}(P \mathrel{-\!\!*} Q)} \qquad \textsc{timeless-or}\quad \frac{\mathsf{timeless}(P) \qquad \mathsf{timeless}(Q)}{\mathsf{timeless}(P \lor Q)}$$

$$\textsc{timeless-and}\quad \frac{\mathsf{timeless}(P) \qquad \mathsf{timeless}(Q)}{\mathsf{timeless}(P \land Q)} \qquad \textsc{timeless-all}\quad \frac{\forall x.\, \mathsf{timeless}(P(x))}{\mathsf{timeless}(\forall x.\, P(x))} \qquad \textsc{timeless-exists}\quad \frac{\forall x.\, \mathsf{timeless}(P(x))}{\mathsf{timeless}(\exists x.\, P(x))}$$

$$\textsc{timeless-points-to}\quad \mathsf{timeless}(\ell \mapsto v) \qquad \textsc{wp-timeless-strip}\quad \frac{\mathsf{timeless}(Q) \qquad P * Q \vdash \mathbf{wp}^{\mathcal{E}}\, e\, \{v.\, R(v)\}}{P * \triangleright Q \vdash \mathbf{wp}^{\mathcal{E}}\, e\, \{v.\, R(v)\}}$$

Figure 3.5: Proof rules for timeless propositions.

Timeless propositions include points-to assertions and pure assertions, and they are closed under most logical connectives. We can eliminate a later from them with wp-timeless-strip.

Using these rules, we can derive the timeless invariant opening rule that we have used for `MutBit`, wp-inv-open-timeless, from the general rule as follows:

Lemma 22.

$$\frac{P * R \vdash \mathbf{wp}^{\mathcal{E}\setminus\mathcal{N}}\, e\, \{v.\, R * Q(v)\} \qquad \mathsf{atomic}(e) \qquad \mathcal{N} \subseteq \mathcal{E} \qquad \mathsf{timeless}(R)}{P * \boxed{R}^{\mathcal{N}} \vdash \mathbf{wp}^{\mathcal{E}}\, e\, \{v.\, Q(v)\}}$$

Proof.

Context	Goal
$P * \boxed{R}^{\mathcal{N}}$	$\mathbf{wp}^{\mathcal{E}}\, e\, \{v.\, Q(v)\}$
Using wp-inv-open	
$P * \triangleright R$	$\mathbf{wp}^{\mathcal{E}\setminus\mathcal{N}}\, e\, \{v.\, \triangleright R * Q(v)\}$
Using wp-timeless-strip and later-intro	
$P * R$	$\mathbf{wp}^{\mathcal{E}\setminus\mathcal{N}}\, e\, \{v.\, R * Q(v)\}$

which follows by assumption. □

Hoare triple rules. We will use impredicative invariants for several applications throughout this dissertation (*e.g.*, in Part II and Part III, we will use them to develop logical relations). It will sometimes be more fitting to consider their rules in "Hoare triple style". To this end, we augment our notion of Hoare triples with a mask by defining $\{P\}\ e\ \{v.\ Q(v)\}_{\mathcal{E}} \triangleq \Box(P \mathrel{-\!\!*} \mathbf{wp}^{\mathcal{E}}\ e\ \{v.\ Q(v)\})$.

We can then trivially derive the following Hoare triple versions of the weakest precondition rules given above:

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

HOARE-INV-ALLOC
$$\frac{\{P * \boxed{R}^{\mathcal{N}}\}\ e\ \{v.\ Q(v)\}_{\mathcal{E}}}{\{P * R\}\ e\ \{v.\ Q(v)\}_{\mathcal{E}}}$$

HOARE-TIMELESS-STRIP
$$\frac{\{P * R\}\ e\ \{v.\ Q(v)\}_{\mathcal{E}} \qquad \mathsf{timeless}(R)}{\{P * \triangleright R\}\ e\ \{v.\ Q(v)\}_{\mathcal{E}}}$$

3.5 Concurrency

Let us now turn to *concurrency*. Up to this point, all the examples that we have considered were *sequential*: there was only ever a single thread of execution. We will now develop techniques to reason about *concurrent* programs. Support for concurrency is nothing Iris specific. Separation logic naturally extends to concurrency (*e.g.*, see O'Hearn; Brookes[13]). What Iris brings to the table are powerful impredicative invariants and expressive forms of resources (§3.6).

[13] O'Hearn, "Resources, concurrency, and local reasoning", 2007 [OHe07]; Brookes, "A semantics for concurrent separation logic", 2007 [Bro07].

But before we dive into the details of verifying concurrent programs, let us start with a simple example:

Example 23 (Coin Flip)**.** Consider the following code snippet, which "flips a coin"[14] by *concurrently* updating and reading from a reference r:

[14] Note that this is not a cryptographically secure technique for sampling random numbers.

$$e_{\mathsf{flip}} \triangleq \mathsf{let}\ r = \mathsf{ref}(0)\ \mathsf{in}\ \mathsf{fork}\{r := 1\};\, !\,r.$$

Concretely, the expression e_{flip} first creates a new reference r, then forks off a new thread that will execute $r := 1$ and then *concurrently* sets r to 1 and dereferences r. Executing two expressions e_1 and e_2 concurrently means that the execution of e_1 and e_2 are interleaved in an arbitrary order.[15] For example, the execution could proceed by first updating r to 1 (previously 0) and then reading the updated value 1. But it could also proceed by first reading the original value 0 and then updating r to 1. ●

[15] This is why the rule WP-INV-OPEN in §3.4.1 has the atomicity side condition. For soundness, it is crucial that invariants cannot be kept open for more than one step in a concurrent program. Otherwise, interleaved threads could observe violations of the invariant.

The fact that concurrent programs execute in an arbitrary interleaving makes them notoriously hard to reason about, because different interleavings can result in different outcomes. For example, the result of executing e_{flip} can be either 0 or 1, depending on the order in which we execute $r := 1$ and $!\,r$. Since we do not know upfront which interleaving will be chosen during the execution, we have to take *all interleavings* into account when we reason about the behavior of e_{flip}. For e_{flip}, this is simple, but once we consider larger programs, the number of interleavings can quickly get out of hand.

In this chapter, we will develop modular reasoning principles for concurrent programs based on separation logic. We will see how we can reason about the interleavings of e_{flip}, how to modularly compose concurrent programs, and how we can get back to more sequential reasoning by, for example, using locks.

3.5.1 A Concurrent Language

We extend the expressions of our language by three new constructs:

$$\begin{array}{rcll} \text{Expressions} & e & ::= & \cdots \mid \mathsf{CAS}(e_1, e_2, e_3) \mid \mathsf{FAA}(e_1, e_2) \mid \mathsf{fork}\{e\} \\ \text{Eval. Contexts} & K & ::= & \cdots \mid \mathsf{FAA}(e_1, K) \mid \mathsf{FAA}(K, v_2) \mid \mathsf{CAS}(e_1, e_2, K) \\ & & \mid & \mathsf{CAS}(e_1, K, v_3) \mid \mathsf{CAS}(K, v_2, v_3) \end{array}$$

The operation $\mathsf{CAS}(e_1, e_2, e_3)$ is a "compare-and-set". It evaluates e_1, e_2, and e_3 to a reference ℓ, a value v, and a value w. Afterwards, it replaces the value in reference ℓ by the value w if it is currently v, and it returns the current value stored in ℓ. The operation $\mathsf{FAA}(e_1, e_2)$ is a "fetch-and-add". It evaluates e_1 and e_2 to a reference ℓ storing an integer m and an integer n. Afterwards, it increments the integer m in reference ℓ by n and returns m. The compare-and-set and fetch-and-add operations are synchronization primitives, which we can use to communicate between threads. The operation $\mathsf{fork}\{e\}$ forks off a new thread executing e and immediately returns. For example, after executing $\mathsf{fork}\{e_1\}; e_2$ the expressions e_1 and e_2 are executing in parallel. The fork operation is analogous to, for example, spawn in C-like languages.

Synchronization and communication. From the perspective of someone who is used to programming in concurrent languages, the above extension to our sequential language may seem a bit strange: seemingly, there is no built-in way to *share data* (*e.g.,* via a lock or a channel). The reason is that we do not need to include these (more user-friendly) communication primitives, because we can derive them from our low-level primitives such as "compare-and-set". To illustrate this point, let us implement a "spin lock":

$$\begin{aligned} \mathtt{mklock}() &\triangleq \mathsf{ref}(\mathsf{false}) \\ \mathtt{lock}(l) &\triangleq \mathsf{if}\,\mathsf{CAS}(l, \mathsf{false}, \mathsf{true})\ \mathsf{then}\ ()\ \mathsf{else}\ \mathtt{lock}(l) \\ \mathtt{unlock}(l) &\triangleq l := \mathsf{false} \end{aligned}$$

We can create a spin lock with `mklock`, acquire a lock with `lock`, and release it again with `unlock`. Internally, the lock is just represented as a reference to a Boolean such that the reference is true while someone is holding the lock and false while the lock is available. To acquire the lock, we "spin" on the reference l until it becomes false (so until the lock becomes available).[16] To release the lock, we simply set the l to false.

[16] In many other languages, spinning on a location as done in `lock` would be considered a data race. Here, we assume a sequentially consistent memory model, which allows for racing on a location to enable synchronization.

For the lock implementation to be correct, we need to know that it is not possible for two threads to acquire the lock at the same time. That is where our synchronization primitive $\mathsf{CAS}(e_1, e_2, e_3)$ comes in: it checks whether the value stored at l is currently false and, in the same instruction, sets it to true so other lock attempts will not succeed. If $\mathsf{CAS}(e_1, e_2, e_3)$ was not a single instruction (*i.e.,* if it would take more than one step to execute), then we could end up in a similar situation where two threads first read and then write a new value without synchronization between them.

By taking the low-level route of including primitives such as compare-and-set instead of high-level primitives such as locks, we work closer to the instructions that are offered by modern processors. It also means that we can reason about the implementation of a spin lock, a ticket lock, or other fine-grained concurrent data structures.

3.5.2 A Concurrent Separation Logic

Now that we have concurrency in our language, how can we prove anything about concurrent programs? Separation logic is especially well equipped to reason about concurrent programs, because we already have built-in, fine-grained control over which data is *shared* (using invariants) and which data is *exclusively owned* (using ordinary points-to assertions). We extend our logic with the following rules for our new primitives:

wp-suc-cas

$$\frac{v_1 = v_2 \qquad v_1, v_2 \textit{ comparable}}{\ell \mapsto v_1 * \triangleright(\ell \mapsto w \twoheadrightarrow Q(\mathsf{true})) \vdash \mathbf{wp}^{\mathcal{E}}\, \mathsf{CAS}(\ell, v_2, w)\, \{v.\, Q(v)\}}$$

wp-fail-cas

$$\frac{v_1 \neq v_2 \qquad v_1, v_2 \textit{ comparable}}{\ell \mapsto v_1 * \triangleright(\ell \mapsto v_1 \twoheadrightarrow Q(\mathsf{false})) \vdash \mathbf{wp}^{\mathcal{E}}\, \mathsf{CAS}(\ell, v_2, w)\, \{v.\, Q(v)\}}$$

wp-faa

$$\ell \mapsto m * \triangleright(\ell \mapsto m + n \twoheadrightarrow Q(m)) \vdash \mathbf{wp}^{\mathcal{E}}\, \mathsf{FAA}(\ell, n)\, \{v.\, Q(v)\}$$

wp-fork

$$\mathbf{wp}^{\top}\, e\, \{\_.\, \mathsf{True}\} * \triangleright Q() \vdash \mathbf{wp}^{\mathcal{E}}\, \mathsf{fork}\{e\}\, \{v.\, Q(v)\}$$

Figure 3.6: Weakest precondition proof rules for the concurrency primitives.

The rules wp-suc-cas and wp-fail-cas can be used to show that a CAS will be successful (swapping out the value) or not (leaving the value unchanged). They impose the side condition v_1, v_2 *comparable*, which means that v_1 or v_2 should be "simple" in the sense that they can be compared in a single step of execution (because typically processors can only compare word-size data in a single instruction). The rule wp-faa increments the reference ℓ, and wp-fork forks a new thread. (The mask of the new thread is initially $\top$, because all invariants will be closed when e takes its first step.)

Verifying the coin flip. Let us verify the coin flip with the new rules.

Lemma 24. $\{\mathsf{True}\}\ e_{\mathsf{flip}}\ \{v.\, v = 0 \lor v = 1\}$

Proof.

Context	Goal
	$\mathbf{wp}\ e_{\mathsf{flip}}\ \{v.\, v = 0 \lor v = 1\}$
$r \mapsto 0$	$\mathbf{wp}\ \mathsf{fork}\{r := 1\}; !\,r\ \{v.\, v = 0 \lor v = 1\}$
We allocate the invariant:	
$\boxed{r \mapsto 0 \lor r \mapsto 1}^{\mathcal{N}}$	$\mathbf{wp}\ \mathsf{fork}\{r := 1\}; !\,r\ \{v.\, v = 0 \lor v = 1\}$
Using wp-bind and wp-fork (and some additional simplification).	
$\boxed{r \mapsto 0 \lor r \mapsto 1}^{\mathcal{N}}$	$\mathbf{wp}\ r := 1\ \{\_.\, \mathsf{True}\} * \mathbf{wp}\ !\,r\ \{v.\, v = 0 \lor v = 1\}$
First Goal	
$\boxed{r \mapsto 0 \lor r \mapsto 1}^{\mathcal{N}}$	$\mathbf{wp}\ r := 1\ \{\_.\, \mathsf{True}\}$
Using wp-inv-open-timeless	
$r \mapsto 0 \lor r \mapsto 1$	$\mathbf{wp}^{\top \setminus \mathcal{N}}\ r := 1\ \{\_.\, r \mapsto 0 \lor r \mapsto 1\}$
Follows trivially with wp-store.	
Second Goal	
$\boxed{r \mapsto 0 \lor r \mapsto 1}^{\mathcal{N}}$	$\mathbf{wp}\ !\,r\ \{v.\, v = 0 \lor v = 1\}$
Using wp-inv-open-timeless	
$r \mapsto 0 \lor r \mapsto 1$	$\mathbf{wp}^{\top \setminus \mathcal{N}}\ !\,r\ \{v.\, (v = 0 \lor v = 1) * (r \mapsto 0 \lor r \mapsto 1)\}$
Follows trivially with wp-load.	□

Example: Lock. For our next example, we return to the spin lock example. Typically, a lock is supposed to guard some exclusive piece of data (*e.g.,* a reference to a mutable list) which "becomes available" (*i.e.,* we may access it) upon acquiring the lock and has to be returned upon releasing the lock (*i.e.,* we may no longer access it). In Iris, we will specify a lock with the predicate $\mathsf{lock}(\ell, P)$, which means that ℓ is a lock guarding the resource P (an Iris assertion). For this notion of a lock, we then want to show the following specification witnessing the exchange of P between `lock` and `unlock`:

$$\{P\}\ \mathtt{mklock}()\ \{v.\ \exists \ell.\ v = \ell * \mathsf{lock}(\ell, P)\} \qquad \{\mathsf{lock}(\ell, P)\}\ \mathtt{lock}(\ell)\ \{\_.\ P\}$$

$$\{\mathsf{lock}(\ell, P) * P\}\ \mathtt{unlock}(\ell)\ \{\_.\ \mathsf{True}\}$$

One can think of a lock $\mathsf{lock}(\ell, P)$ as an invariant P that is tied to program code. We open it with `lock` and we, subsequently, close it again with `unlock`. While the lock is "open", we can freely use (and break) P, but at the time when we want to "close" it again, we have to return (and restore) P. In fact, that is exactly how we define $\mathsf{lock}(\ell, P)$:

$$\mathsf{lock}(\ell, P) \triangleq \boxed{\ell \mapsto \mathsf{true} \vee (\ell \mapsto \mathsf{false} * P)}^{\mathcal{N}}$$

Here, we benefit from the impredicativity of invariants. We can simply put an arbitrary Iris proposition P into the invariant—regardless of whether it mentions another invariant, a weakest precondition, or another lock.

Given the definition of $\mathsf{lock}(\ell, P)$, the verification of the lock operations is straightforward. We show the case for `lock`.

Lemma 25. $\{\mathsf{lock}(\ell, P)\}\ \mathtt{lock}(\ell)\ \{\_.\ P\}$.

Proof.

Context	Goal
$\mathsf{lock}(\ell, P)$	$\mathbf{wp}\ \mathtt{lock}(\ell)\ \{\_.\ P\}$
By Löb induction.	
$\mathsf{lock}(\ell, P) * \triangleright\, \mathbf{wp}\ \mathtt{lock}(\ell)\ \{\_.\ P\}$	$\mathbf{wp}\ \mathtt{lock}(\ell)\ \{\_.\ P\}$
Executing for one step.	
$\mathsf{lock}(\ell, P) * \mathbf{wp}\ \mathtt{lock}(\ell)\ \{\_.\ P\}$	
	$\mathbf{wp}\ \mathsf{if}\ \mathsf{CAS}(\ell, \mathsf{false}, \mathsf{true})\ \mathsf{then}\ ()\ \mathsf{else}\ \mathtt{lock}(\ell)\ \{\_.\ P\}$
By binding on CAS with $\Phi(v) \triangleq \mathbf{wp}\ \mathsf{if}\ v\ \mathsf{then}\ ()\ \mathsf{else}\ \mathtt{lock}(\ell)\ \{\_.\ P\}$.	
$\mathsf{lock}(\ell, P) * \mathbf{wp}\ \mathtt{lock}(\ell)\ \{\_.\ P\}$	$\mathbf{wp}\ \mathsf{CAS}(\ell, \mathsf{false}, \mathsf{true})\ \{v.\ \Phi(v)\}$
By opening the invariant (wp-inv-open) and commuting in the later modality.	
$(\ell \mapsto \mathsf{true} \vee \ell \mapsto \mathsf{false} * \triangleright P) * \mathbf{wp}\ \mathtt{lock}(\ell)\ \{\_.\ P\}$	
	$\mathbf{wp}\ \mathsf{CAS}(\ell, \mathsf{false}, \mathsf{true})\ \{v.\ (\ell \mapsto \mathsf{true} \vee \ell \mapsto \mathsf{false} * \triangleright P) * \Phi(v)\}$
Case $\ell \mapsto \mathsf{true}$.	
By wp-fail-cas, we are left to prove	
$\ell \mapsto \mathsf{true} * \mathbf{wp}\ \mathtt{lock}(\ell)\ \{\_.\ P\}$	
	$(\ell \mapsto \mathsf{true} \vee \ell \mapsto \mathsf{false} * \triangleright P) * \Phi(\mathsf{false})$
$\mathbf{wp}\ \mathtt{lock}(\ell)\ \{\_.\ P\}$	$\Phi(\mathsf{false})$
which is trivial using the recursive assumption $\mathbf{wp}\ \mathtt{lock}(\ell)\ \{\_.\ P\}$.	

Context	Goal
Case $\ell \mapsto \mathsf{false} * \triangleright P$.	
By wp-suc-cas, we are left to prove	
$\ell \mapsto \mathsf{true} * P * \mathbf{wp}\ \mathtt{lock}(\ell)\ \{\_.\, P\}$	
	$(\ell \mapsto \mathsf{true} \vee \ell \mapsto \mathsf{false} * \triangleright P) * \Phi(\mathsf{true})$
We do not have to return P and are left to prove	
$P * \mathbf{wp}\ \mathtt{lock}(\ell)\ \{\_.\, P\}$	$\Phi(\mathsf{true})$
which is trivial.	$\square$

3.6 Ghost State

Let us now turn to *ghost state* and *resources*. The only resource that we have considered so far is the points-to assertion $\ell \mapsto v$. However, Iris offers a much richer variety of resources. To understand how these resources work and where they are useful, we consider an example: a fine-grained concurrent, monotone counter (*i.e.,* a counter that does not use locking and only increases in value).

Example 26 (Concurrent Counter)**.** Our counter offers three methods:

$$\mathtt{mkcounter}\ () \triangleq \mathsf{ref}(0) \qquad \mathtt{get}(c) \triangleq\ !c \qquad \mathtt{inc}(c) \triangleq \mathsf{FAA}(c, 1)$$

We can create a counter with `mkcounter` (implemented as a reference internally), we can read its value with `get` (implemented by reading the reference), and we can increment its value with `inc` (implemented with fetch-and-add).

We will specify this counter with a predicate $\mathsf{counter}(c, n)$, which expresses that the value of counter c is currently at least n. Importantly, $\mathsf{counter}(c, n)$ only expresses that the counter value is "at least n" and not "exactly n", because we are considering a *concurrent* counter. That is, the counter can be shared between threads, and, after we have observed the counter value (*e.g.,* with a `get`), other threads can increment it, invalidating any assumptions about its *exact* value (but not about lower bounds). We will prove the specification:

$$\mathsf{counter}(c, n) \vdash \square\, \mathsf{counter}(c, n) \qquad \{\mathsf{True}\}\ \mathtt{mkcounter}\ ()\ \{c.\, \mathsf{counter}(c, 0)\}$$

$$\{\mathsf{counter}(c, n)\}\ \mathtt{get}(c)\ \{v.\, \exists m \geq n.\, v = m * \mathsf{counter}(c, m)\}$$

$$\{\mathsf{counter}(c, n)\}\ \mathtt{inc}(c)\ \{v.\, \exists m \geq n.\, v = m * \mathsf{counter}(c, m + 1)\}$$

The counter predicate is persistent (to be shared with other threads). We initially obtain a counter of at least 0. The operation `get` returns a value m that is at least the last observed value (and updates our bound), and the operation `inc` increases the counter. ●

How can we define the predicate $\mathsf{counter}(c, n)$? For it to be sharable between threads (*i.e.,* to be persistent), we must wrap the ownership of c in an invariant. But naïve attempts to do so such as:

$$\boxed{\exists m : \mathbb{N}.\, c \mapsto m}^{\mathcal{N}} \qquad \textit{or} \qquad \exists m : \mathbb{N}.\, \boxed{c \mapsto m}^{\mathcal{N}}$$

are destined to fail. The first invariant does not guarantee that m only increases (*i.e.,* it can become any natural number). The second invariant does not allow us to change m, so we could never increase it. We need a way to express "m can be changed, but it can only be increased". This is where *ghost state* comes in.

Ghost state. Ghost state is auxiliary logical state that is useful for the verification of a program.[17] In this particular case, for the verification of the counter, we use the ghost state of *monotonically growing natural numbers*. Monotonically growing natural numbers come in the form of two propositions, $\mathsf{mono}_\gamma(n)$ and $\mathsf{lb}_\gamma(n)$, which represent two pieces of ghost state connected by the name γ. The ghost state $\mathsf{mono}_\gamma(n)$ expresses that the current value of γ is n and that it can only grow over time. The ghost state $\mathsf{lb}_\gamma(n)$ is a lower bound on the value of γ. (It remains a lower bound on the value of γ, because $\mathsf{mono}_\gamma(n)$ can only grow.) Their relationship is captured by the following rules:

[17] In typical Iris fashion, we use the term "ghost state" liberally to refer to any construction based on resource algebras (§4.2). As we will see in §4.1, this includes ghost state constructions for the heap.

Monotonically growing natural numbers $\boxed{\mathsf{mono}_\gamma(n) \text{ and } \mathsf{lb}_\gamma(n)}$

make-bound
$$\mathsf{mono}_\gamma(n) \vdash \mathsf{mono}_\gamma(n) * \mathsf{lb}_\gamma(n)$$

use-bound
$$\mathsf{mono}_\gamma(n) * \mathsf{lb}_\gamma(m) \vdash n \geq m$$

bound-pers
$$\mathsf{lb}_\gamma(n) \vdash \square\, \mathsf{lb}_\gamma(n)$$

increase-val
$$\mathsf{mono}_\gamma(n) \vdash {\Rrightarrow}\, \mathsf{mono}_\gamma(n+1)$$

new-mono
$$\mathsf{True} \vdash {\Rrightarrow}\, \exists \gamma.\, \mathsf{mono}_\gamma(n)$$

mono-timeless
$$\mathsf{timeless}(\mathsf{mono}_\gamma(n))$$

bound-timeless
$$\mathsf{timeless}(\mathsf{lb}_\gamma(n))$$

The rule make-bound allows us to create a new lower bound $\mathsf{lb}_\gamma(n)$ from the current value $\mathsf{mono}_\gamma(n)$. The rule use-bound then later on allows us to show that any lower bound that we have created $\mathsf{lb}_\gamma(m)$ is smaller than the current value $\mathsf{mono}_\gamma(n)$. The rule bound-pers ensures that the lower bounds $\mathsf{lb}_\gamma(n)$ are persistent. The rules increase-val and new-mono use a new modality of Iris, the update modality ${\Rrightarrow} P$, which we will discuss shortly. Intuitively, increase-val says that we can always increase the current value $\mathsf{mono}_\gamma(n)$ by one with an update. The rule new-mono allows us to create a new monotonically growing natural number $\mathsf{mono}_\gamma(n)$ where the rule picks (a fresh) name γ for us. The rules mono-timeless and bound-timeless ensure that both new connectives are timeless and, hence, easy to use in invariants.

With this ghost state, we can define the counter predicate:

$$\mathsf{counter}(c, n) \triangleq \exists \gamma.\, \boxed{\exists m : \mathbb{N}.\, c \mapsto m * \mathsf{mono}_\gamma(m)}^{\mathcal{N}} * \mathsf{lb}_\gamma(n)$$

We tie the current value of the counter m to the monotonically growing ghost state γ in the invariant by asserting that the current value is m (with $\mathsf{mono}_\gamma(m)$). Outside of the invariant, we track n as a lower bound on the counter. Since the lower bound is persistent, the entire counter predicate is persistent.

The update modality. The update modality ${\Rrightarrow} P$ means that P holds after (possibly) performing some updates to the current ghost state. Besides ghost state specific rules like increase-val and new-mono, the modality has several structural rules, depicted in Fig. 3.7. With upd-return, we can always update the current state P to itself (a no-op). With upd-bind, we can compose two updates into a single update. (Together, the rules turn the update modality ${\Rrightarrow} P$ into a monad.) With wp-upd, we can execute an update at a weakest precondition.

The update modality $\boxed{\Rrightarrow P}$

Figure 3.7: Structural proof rules for the update modality.

upd-return
$$P \vdash \Rrightarrow P$$

upd-bind
$$(\Rrightarrow P) * (P \mathrel{-\!\!*} \Rrightarrow Q) \vdash \Rrightarrow Q$$

wp-upd
$$\Rrightarrow \mathbf{wp}\ e\ \{v.\, Q(v)\} \vdash \mathbf{wp}\ e\ \{v.\, Q(v)\}$$

From these basic rules, we can then derive several useful auxiliary rules:

upd-wand
$$(\Rrightarrow P) * (P \mathrel{-\!\!*} Q) \vdash \Rrightarrow Q$$

upd-trans
$$\Rrightarrow \Rrightarrow P \vdash \Rrightarrow P$$

upd-frame
$$P * \Rrightarrow Q \vdash \Rrightarrow (P * Q)$$

upd-mono
$$\frac{P \vdash Q}{\Rrightarrow P \vdash \Rrightarrow Q}$$

hoare-upd
$$\frac{\{P\}\ e\ \{v.\, Q(v)\}}{\{\Rrightarrow P\}\ e\ \{v.\, Q(v)\}}$$

With the update rules in hand, we will now look at an example of how one can update ghost state during the proof of a weakest precondition:

Lemma 27. $(\mathsf{mono}_\gamma(n{+}1) \mathrel{-\!\!*} \mathbf{wp}\ e\ \{v.\, Q(v)\}) \vdash (\mathsf{mono}_\gamma(n) \mathrel{-\!\!*} \mathbf{wp}\ e\ \{v.\, Q(v)\})$

Proof.

Context	Goal
$(\mathsf{mono}_\gamma(n{+}1) \mathrel{-\!\!*} \mathbf{wp}\ e\ \{v.\, Q(v)\}) * \mathsf{mono}_\gamma(n)$	$\mathbf{wp}\ e\ \{v.\, Q(v)\}$
By increase-val, we can update $\mathsf{mono}_\gamma(n)$ to $\mathsf{mono}_\gamma(n{+}1)$.	
$(\mathsf{mono}_\gamma(n{+}1) \mathrel{-\!\!*} \mathbf{wp}\ e\ \{v.\, Q(v)\}) * \Rrightarrow \mathsf{mono}_\gamma(n{+}1)$	$\mathbf{wp}\ e\ \{v.\, Q(v)\}$
By wp-upd, we can add an update in front of the weakest precondition.	
$(\mathsf{mono}_\gamma(n{+}1) \mathrel{-\!\!*} \mathbf{wp}\ e\ \{v.\, Q(v)\}) * \Rrightarrow \mathsf{mono}_\gamma(n{+}1)$	$\Rrightarrow \mathbf{wp}\ e\ \{v.\, Q(v)\}$
Follows by upd-wand.	□

Verifying the counter implementation. Let us now return to actually verifying the counter implementation. We focus only on `mkcounter` and `inc`. The proof for `get` is analogous.

Lemma 28. $\{\mathsf{True}\}$ `mkcounter` $()\ \{c.\, \mathsf{counter}(c, 0)\}$

Proof.

Context	Goal
	$\mathbf{wp}$ `mkcounter` $()\ \{c.\, \mathsf{counter}(c, 0)\}$
$c \mapsto 0$	$\mathbf{wp}\ c\ \{c.\, \mathsf{counter}(c, 0)\}$
We allocate a new resource with new-mono and wp-upd.	
$c \mapsto 0 * \mathsf{mono}_\gamma(0)$	$\mathbf{wp}\ c\ \{c.\, \mathsf{counter}(c, 0)\}$
We create a lower bound with make-bound.	
$c \mapsto 0 * \mathsf{mono}_\gamma(0) * \mathsf{lb}_\gamma(0)$	$\mathbf{wp}\ c\ \{c.\, \mathsf{counter}(c, 0)\}$
We allocate the counter invariant with wp-inv-alloc.	
$\boxed{\exists m : \mathbb{N}.\, c \mapsto m * \mathsf{mono}_\gamma(m)}^{\mathcal{N}} * \mathsf{lb}_\gamma(0)$	$\mathbf{wp}\ c\ \{c.\, \mathsf{counter}(c, 0)\}$
The rest of the proof is trivial.	□

Lemma 29. $\{\mathsf{counter}(c,n)\}\ \mathtt{inc}(c)\ \{v.\ \exists m \geq n.\ v = m * \mathsf{counter}(c, m+1)\}$

Proof.

Context　　Goal

$\mathsf{counter}(c,n)$　　$\mathbf{wp}\ \mathtt{inc}(c)\ \{v.\ \exists m \geq n.\ v = m * \mathsf{counter}(c, m+1)\}$

$\boxed{\exists m : \mathbb{N}.\ c \mapsto m * \mathsf{mono}_\gamma(m)}^{\mathcal{N}} * \mathsf{lb}_\gamma(n)$

$\mathbf{wp}\ \mathsf{FAA}(c,1)\ \{v.\ \exists m \geq n.\ v = m * \mathsf{counter}(c, m+1)\}$

Using wp-inv-open-timeless with $\Phi(v) \triangleq \exists m \geq n.\ v = m * \mathsf{counter}(c, m+1)$.

$\boxed{\exists m : \mathbb{N}.\ c \mapsto m * \mathsf{mono}_\gamma(m)}^{\mathcal{N}} * \mathsf{lb}_\gamma(n)$

$* (\exists m : \mathbb{N}.\ c \mapsto m * \mathsf{mono}_\gamma(m))$

$\mathbf{wp}^{\top\setminus\mathcal{N}}\ \mathsf{FAA}(c,1)\ \big\{v.\ (\exists m : \mathbb{N}.\ c \mapsto m * \mathsf{mono}_\gamma(m)) * \Phi(v)\big\}$

Using use-bound

$\boxed{\exists m : \mathbb{N}.\ c \mapsto m * \mathsf{mono}_\gamma(m)}^{\mathcal{N}}$

$* c \mapsto m * \mathsf{mono}_\gamma(m) * n \leq m$

$\mathbf{wp}^{\top\setminus\mathcal{N}}\ \mathsf{FAA}(c,1)\ \big\{v.\ (\exists m : \mathbb{N}.\ c \mapsto m * \mathsf{mono}_\gamma(m)) * \Phi(v)\big\}$

We update $\mathsf{mono}_\gamma(m)$ with Lemma 27 and obtain a lower bound with make-bound.

$\boxed{\exists m : \mathbb{N}.\ c \mapsto m * \mathsf{mono}_\gamma(m)}^{\mathcal{N}}$

$* c \mapsto m * \mathsf{mono}_\gamma(m+1) * \mathsf{lb}_\gamma(m+1) * n \leq m$

$\mathbf{wp}^{\top\setminus\mathcal{N}}\ \mathsf{FAA}(c,1)\ \big\{v.\ (\exists m : \mathbb{N}.\ c \mapsto m * \mathsf{mono}_\gamma(m)) * \Phi(v)\big\}$

Using wp-faa, we increment c.

$\boxed{\exists m : \mathbb{N}.\ c \mapsto m * \mathsf{mono}_\gamma(m)}^{\mathcal{N}}$

$* c \mapsto m + 1 * \mathsf{mono}_\gamma(m+1) * \mathsf{lb}_\gamma(m+1) * n \leq m$

$(\exists m : \mathbb{N}.\ c \mapsto m * \mathsf{mono}_\gamma(m)) * \Phi(m)$

We give up ownership of the counter reference and $\mathsf{mono}_\gamma(m+1)$.

$\boxed{\exists m : \mathbb{N}.\ c \mapsto m * \mathsf{mono}_\gamma(m)}^{\mathcal{N}} * \mathsf{lb}_\gamma(m+1) * n \leq m$

$\Phi(m)$

We can now assemble the predicate $\mathsf{counter}(c, m+1)$.

$\mathsf{counter}(c, m+1) * n \leq m$

$\Phi(m)$

The rest of the proof is trivial. □

More ghost state. Monotonically growing natural numbers are only one of Iris's many different forms of ghost state. To support different forms of ghost state, Iris has a generic mechanism based on so-called "*resource algebras*". We will discuss resource algebras and how one can derive ghost theories from them in §4.2. For now, we conclude our tour of the key features of Iris and turn to the model of Iris.

Chapter 4

The Model of Iris

After spending the last chapter discussing the different features of Iris, let us now take a closer look at the model of Iris (*i.e.,* we discuss how Iris's propositions and connectives are defined). We focus mainly on a simplified model without impredicative invariants, which suffices to explain the contributions of the dissertation in the subsequent parts. The full model of Iris is described in "Iris from the ground up: A modular foundation for higher-order concurrent separation logic",[1] and we briefly discuss what is necessary to extend the simplified model to the full model at the end of this chapter.

[1] Jung et al., "Iris from the ground up: A modular foundation for higher-order concurrent separation logic", 2018 [Jun+18b].

The model of Iris consists mainly of two parts: the *program logic* and the *base logic*. The program logic is concerned with programs, heaps, and the weakest precondition. The base logic is agnostic about all of those and is merely concerned with resources, resource management, and step-indexing. It forms a simple foundation upon which the program logic is built. We start with the program logic (§4.1) and then work our way down to the base logic (§4.3). Along the way, we introduce resources as a foundation for ghost state (§4.2). We close our tour of the simplified model with a discussion of Iris's soundness (§4.4) and a sketch of how to extend it with impredicative invariants (§4.5).

4.1 The Program Logic

The centerpiece of the program logic is the weakest precondition. It is defined internally inside of Iris by tying the *operational semantics* of a language[2] to separation logic resources. Let us write $(e, h) \rightsquigarrow (e', h')$ for a single, sequential step in the operational semantics. Then the weakest precondition is defined as follows (omitting masks $\mathcal{E}$, support for invariants, and concurrency)[3]:

[2] In Iris, the program logic is defined parametrically over the programming language at hand. In the following, we will mostly focus on HeapLang as the instantiation of the language.

[3] We will discuss concurrency shortly, and masks and invariants are discussed in §4.5.1.

Sequential Weakest Precondition $\boxed{\textbf{wp}\ e\ \{v.\ Q(v)\}}$

$$\begin{aligned}
&\textbf{wp}\ v\ \{w.\ Q(w)\} \triangleq {\Rrightarrow} Q(v) \\
&\textbf{wp}\ e\ \{w.\ Q(w)\} \triangleq \forall h.\ \mathrm{SI}(h) \mathrel{-\!\!*} {\Rrightarrow} \mathrm{progress}(e, h) && \text{if } e \notin \mathit{Val} \\
&\qquad\qquad * \ (\forall e', h'.\ (e, h) \rightsquigarrow (e', h') \mathrel{-\!\!*} \triangleright {\Rrightarrow} (\mathrm{SI}(h') * \textbf{wp}\ e'\ \{w.\ Q(w)\}))
\end{aligned}$$

The definition has two cases. In the first case, the value case, one has to prove the postcondition $Q(v)$ after potentially updating the ghost state with an update "${\Rrightarrow}$". Otherwise, if e is a proper expression, one gets to assume the state interpretation $\mathrm{SI}(h)$ (which ties the physical state h to the logical state; explained below) and has to show two conditions: (1) the current expression e can make progress in the heap h where $\mathrm{progress}(e, h) \triangleq \exists e', h'.\ (e, h) \rightsquigarrow (e', h')$

and (2) for any successor expression e' and heap h', one can re-establish the state interpretation $\mathrm{SI}(h')$ and a weakest precondition for e'.

The use of the modalities in this definition is essential. The update modality "$\Rrightarrow$" (in red) is needed to allow (1) updating auxiliary ghost state (*e.g.,* for the concurrent counter from Example 26) and (2) updating the resources to the new state h' (*e.g.,* in case e has updated a location in the heap h). The later modality "$\triangleright$" (in blue) ties the program steps to step-indexing, because every program step is now accompanied by a later modality. This later is responsible for the intuition: *"$\triangleright P$ means P will hold after the next step of computation"* (see §3.2). That is, as we have seen in the verification of `first` (see Example 13), when we take a step (*e.g.,* with wp-later-pure-step), the goal is put underneath a later. We can then use later-mono to remove guarding laters from hypotheses (*e.g.,* for ones obtained from a Löb induction).

The state interpretation. With the state interpretation $\mathrm{SI}(h)$, we can control which heaps h—and more generally which *program state*—we have to consider in the weakest precondition for the next program step. It is like an invariant that we maintain throughout the execution of e: it holds initially and the weakest precondition preserves it for every step. We use it to tie the resource assertions of the program logic to the concrete program state that is used in the operational semantics.

For example, in the particular case of HeapLang, we use the state interpretation $\mathrm{SI}(h)$ to tie the heap h in the weakest precondition to the points-to assertions $\ell \mapsto v$ of the program logic. We do so via ghost state. More concretely, we define $\mathrm{SI}(h) \triangleq \mathsf{heap}(h)$ for the following *heap ghost theory*:[4]

pts-excl
$$\ell \mapsto v * \ell \mapsto w \vdash \mathsf{False}$$

heap-lookup
$$\ell \mapsto v * \mathsf{heap}(h) \vdash h(\ell) = v$$

heap-alloc
$$\ell \notin \mathsf{dom}\, h * \mathsf{heap}(h) \vdash \Rrightarrow \mathsf{heap}(h[\ell \mapsto v]) * \ell \mapsto v$$

heap-update
$$\ell \mapsto v * \mathsf{heap}(h) \vdash \Rrightarrow \ell \mapsto w * \mathsf{heap}(h[\ell \mapsto w])$$

The assertion $\mathsf{heap}(h)$ keeps track of the current state of the entire heap h and the points-to assertion $\ell \mapsto v$ carries the ownership of one individual location ℓ. The points-to assertion cannot be duplicated (pts-excl), and it determines the value of ℓ in h (heap-lookup). We can allocate fresh locations $\ell \notin \mathsf{dom}\, h$ with heap-alloc, and we can update a location in the heap with heap-update. We will discuss shortly how this ghost theory can be defined in the base logic of Iris (see §4.2.2).

Recursion. Above, we have already discussed the primary purpose of the later modality "$\triangleright$" in the definition of the weakest precondition $\mathbf{wp}\ e\ \{v.\, Q(v)\}$: to enable step-indexed reasoning such as verifying recursive functions via Löb induction. However, it also serves a secondary purpose: it resolves the cycle in the definition of the weakest precondition. That is, note that the weakest precondition is defined recursively in terms of itself (in the second case). Step-indexing resolves this cycle[5] by offering a *guarded fixpoint combinator* $\mu\, f\, x.\, P\, f\, x$. It provides a solution to a recursive definition if every recursive

[4] In this ghost theory the points-to assertions $\ell \mapsto v$ gives *exclusive* access to the location ℓ. In Part V, we will also make use of a *fractional* version $\ell \mapsto_q v$ where for $0 < q < 1$, the points-to allows reading but not writing, and for $q = 1$, it coincides with the exclusive $\ell \mapsto v$.

[5] Alternatively, since the weakest precondition occurs in a positive position, we could also use a least- or greatest fixpoint here to obtain a well-founded definition. We will see such a definition in Part II.

occurrence occurs underneath at least one later modality $\triangleright P$. Formally, we have $(\mu\, f x.\, P\, f\, x)\, y \dashv\vdash P(\mu\, f x.\, P\, f\, x)\, y$ if every occurrence of f in P occurs underneath a later modality. We will define the combinator using step-indexing in the model in §4.3.

Concurrency. Let us now turn to concurrency, which is a surprisingly simple extension. Let us write $(e, h) \rightsquigarrow (e', h', es)$ for a step in the operational semantics of a concurrent language, where es is a (potentially empty) list of additional threads that have been forked-off in this step (*e.g.*, $(\mathsf{fork}\{e\}, h) \rightsquigarrow ((), h, [e])$ in the case of HeapLang). Then we extend the definition of the weakest precondition as follows (with additions highlighted in orange):

Concurrent Weakest Precondition $\boxed{\textbf{wp}\ e\ \{v.\, Q(v)\}}$

$$\begin{aligned}
\textbf{wp}\ v\ \{w.\, Q(w)\} &\triangleq \Rrightarrow Q(v) \\
\textbf{wp}\ e\ \{w.\, Q(w)\} &\triangleq \forall h.\, \mathrm{SI}(h) \mathrel{-\!\!*} \Rrightarrow \mathrm{progress}(e, h) \qquad\qquad \text{if } e \notin Val \\
&\qquad * \ \forall e', h', es.\, (e, h) \rightsquigarrow (e', h', es) \mathrel{-\!\!*} \\
&\qquad\qquad \triangleright \Rrightarrow (\mathrm{SI}(h') * \textbf{wp}\ e'\ \{w.\, Q(w)\} * \textstyle\mathop{\Large *}_{e'' \in es} \textbf{wp}\ e''\ \{\_.\, \mathsf{True}\})
\end{aligned}$$

where now $\mathrm{progress}(e, h) \triangleq \exists e', h', es.\, (e, h) \rightsquigarrow (e', h', es)$. Besides the recursive occurrence of the weakest precondition of the successor expression e', we additionally add one recursive occurrence $\textbf{wp}\ e''\ \{\_.\, \mathsf{True}\}$ per forked-off thread e''. All recursive occurrences are still guarded by a later modality, so the weakest precondition can still be defined as a guarded fixpoint.

Soundness. With the definition of the weakest precondition in hand, one can prove the soundness of the rules of the program logic depicted in Fig. 4.1.[6] For example, let us sketch the proof for the rule wp-later-store:

[6] Notably, these rules do not contain the rules for invariants wp-inv-open and wp-inv-alloc and the rule for timelessness wp-timeless-strip. We will discuss invariants and timelessness in §4.5.

Lemma 30. $\ell \mapsto v * \triangleright(\ell \mapsto w \mathrel{-\!\!*} Q()) \vdash \textbf{wp}\ \ell := w\ \{u.\, Q(u)\}$

Proof Sketch. In the operational semantics of HeapLang, $\ell := w$ reduces to $()$ if and only if (1) the original heap h already contains ℓ, (2) the new heap h' is the same as h except that ℓ now stores w, and (3) no threads are forked off. Using this fact, let us sketch the proof. The definition of the weakest precondition allows us to assume $\mathrm{SI}(h)$ for some heap h. We can then use our points-to assertion $\ell \mapsto v$ to deduce $h(\ell) = v$ using heap-lookup. Thus, we know that $\ell := w$ can make progress in h. Next, let us assume an arbitrary step $(\ell := w, h) \rightsquigarrow (e', h', es)$. By the operational semantics of $\ell := w$, we know that it can only step to the expression $e' = ()$ and the heap $h' = h[\ell \mapsto w]$. Moreover, we know es must be empty. We can use heap-update to update the state interpretation from h to h', which also gives us the ownership of $\ell \mapsto w$. We are left to prove $\ell \mapsto w * (\ell \mapsto w \mathrel{-\!\!*} Q()) \vdash \textbf{wp}\ ()\ \{u.\, Q\, u\}$, which follows by the value case of the weakest precondition. □

Like wp-later-store, the rules wp-later-ref, wp-later-load, wp-suc-cas, wp-fail-cas, wp-faa, and wp-fork are all clearly specific to HeapLang. They depend on the choice of the state interpretation and the constructs of the language. The other rules on the other hand (*i.e.*, wp-value, wp-wand, wp-bind, wp-upd, and wp-later-pure-step) are quite generic. They do not mention any concrete language construct and also do not depend on the particular state interpretation $\mathrm{SI}(h)$

Language Generic Rules

Figure 4.1: Weakest precondition rules.

wp-value
$$Q(v) \vdash \mathbf{wp}\; v\; \{w.\, Q(w)\}$$

wp-wand
$$(\forall v.\, Q(v) \mathrel{-\!\!*} Q'(v)) * \mathbf{wp}\; e\; \{w.\, Q(w)\} \vdash \mathbf{wp}\; e\; \{w.\, Q'(w)\}$$

wp-bind
$$\mathbf{wp}\; e\; \big\{v.\, \mathbf{wp}\; K[v]\; \{w.\, Q(w)\}\big\} \vdash \mathbf{wp}\; K[e]\; \{w.\, Q(w)\}$$

wp-upd
$$\Rrightarrow \mathbf{wp}\; e\; \{v.\, Q(v)\} \vdash \mathbf{wp}\; e\; \{v.\, Q(v)\}$$

wp-later-pure-step
$$\frac{e \longrightarrow_{\mathsf{pure}} e'}{\triangleright\, \mathbf{wp}\; e'\; \{v.\, Q(v)\} \vdash \mathbf{wp}\; e\; \{v.\, Q(v)\}}$$

Language Specific Rules

wp-later-ref
$$\triangleright(\forall \ell.\, \ell \mapsto v \mathrel{-\!\!*} Q(\ell)) \vdash \mathbf{wp}\; \mathsf{ref}(v)\; \{w.\, Q(w)\}$$

wp-later-load
$$\ell \mapsto v * \triangleright(\ell \mapsto v \mathrel{-\!\!*} Q(v)) \vdash \mathbf{wp}\; !\,\ell\; \{w.\, Q(w)\}$$

wp-later-store
$$\ell \mapsto v * \triangleright(\ell \mapsto w \mathrel{-\!\!*} Q()) \vdash \mathbf{wp}\; \ell := w\; \{w.\, Q(w)\}$$

wp-suc-cas
$$\frac{v_1 = v_2 \qquad v_1, v_2\ \mathit{comparable}}{\ell \mapsto v_1 * \triangleright(\ell \mapsto w \mathrel{-\!\!*} Q(\mathsf{true})) \vdash \mathbf{wp}\; \mathsf{CAS}(\ell, v_2, w)\; \{v.\, Q(v)\}}$$

wp-fail-cas
$$\frac{v_1 \neq v_2 \qquad v_1, v_2\ \mathit{comparable}}{\ell \mapsto v_1 * \triangleright(\ell \mapsto v_1 \mathrel{-\!\!*} Q(\mathsf{false})) \vdash \mathbf{wp}\; \mathsf{CAS}(\ell, v_2, w)\; \{v.\, Q(v)\}}$$

wp-faa
$$\ell \mapsto m * \triangleright(\ell \mapsto m + n \mathrel{-\!\!*} Q(m)) \vdash \mathbf{wp}\; \mathsf{FAA}(\ell, n)\; \{v.\, Q(v)\}$$

wp-fork
$$\mathbf{wp}\; e\; \{\_.\, \mathsf{True}\} * \triangleright Q() \vdash \mathbf{wp}\; \mathsf{fork}\{e\}\; \{v.\, Q(v)\}$$

that has been chosen. As such, they are proven in Iris in a language-generic fashion (for suitable generalizations of expressions, values, evaluation contexts, and state). And in general, the notion of a pure step $e \longrightarrow_{\mathsf{pure}} e'$ is defined as:

$$\begin{aligned} e \longrightarrow_{\mathsf{pure}} e' \triangleq{} & \mathrm{progress}(e, h) \\ & \wedge (\forall h, h'', e'', es.\, (e, h) \rightsquigarrow (e'', h'', es) \Rightarrow h'' = h \wedge e'' = e' \wedge es = []) \end{aligned}$$

The first part ensures progress and the second part that there are no possible steps to expressions which are not e'.

Having discussed the soundness of the weakest precondition rules, we can now continue further on our journey through the model of Iris. Next up are *resources* and *resource algebras.* They will allow us to define ghost theories like the heap predicate $\mathsf{heap}(h)$ for the state interpretation $\mathrm{SI}(h)$ above (and the corresponding points-to assertions $\ell \mapsto v$).

4.2 Resource Algebras

Instead of a few select ghost theories (*e.g.,* $\mathsf{mono}_\gamma(n)$, $\mathsf{lb}_\gamma(m)$, $\dots$), Iris supports an extensible mechanism for defining and manipulating ghost state. At the heart of this mechanism is the ghost state connective $\boxed{a}^{\gamma}$, which expresses ownership of the *resource* a with name γ. From it different forms of ghost state can be derived by choosing the different kinds of resources.

The resource a is drawn from a *resource algebra* $M = (\mathcal{A}, \cdot, \mathcal{V}, |\_|_{\mathrm{pcore}})$. The resource algebra determines the resulting ghost theory for a. It consists of four parts: (1) a carrier type $\mathcal{A}$, (2) a composition operation $\cdot$, (3) a validity predicate $\mathcal{V}$, and (4) a core projection $|\_|_{\mathrm{pcore}}$. In short, the carrier type together with the binary operation $(\cdot)$ forms a partial commutative monoid that governs how separating conjunction behaves on the resources of the algebra. The (meta-level) predicate $\mathcal{V}$ encapsulates what it means to own a resource—it must be a "valid" resource, meaning $a \in \mathcal{V}$. Finally, the partial core $|\_|_{\mathrm{pcore}}$ maps resources to their duplicable part (*i.e.,* $|a|_{\mathrm{pcore}}$ is obtained from a by stripping off non-duplicable parts), which is used for persistency.

We will discuss the *axioms* that the components of a resource algebra must satisfy shortly. First, let us discuss the logic-level rules that the ghost state connective $\boxed{a}^{\gamma}$ satisfies:

Ghost State Rules $\boxed{\boxed{a}^{\gamma}}$

$$\textsc{res-alloc}\ \frac{a \in \mathcal{V}}{\vdash\ \Rrightarrow \exists \gamma.\, \boxed{a}^{\gamma}} \qquad \textsc{res-upd}\ \frac{a \rightsquigarrow B}{\boxed{a}^{\gamma} \vdash\ \Rrightarrow \exists b \in B.\, \boxed{b}^{\gamma}} \qquad \textsc{res-pers}\ \frac{|a|_{\mathrm{pcore}} \neq \bot}{\boxed{a}^{\gamma} \vdash \Box\, \boxed{|a|_{\mathrm{pcore}}}^{\gamma}}$$

$$\textsc{res-sep}\ \ \boxed{a}^{\gamma} * \boxed{b}^{\gamma} \dashv\vdash \boxed{a \cdot b}^{\gamma} \qquad \textsc{res-valid}\ \ \boxed{a}^{\gamma} \vdash a \in \mathcal{V}$$

First, we can allocate any valid resource a (res-alloc). When we do so, we get $\boxed{a}^{\gamma}$ for some new ghost name γ. Second, we can update resources according to the update relation $a \rightsquigarrow B$, which we will introduce below (res-upd). Third, if we own a resource a, then the core of a, the duplicable part, is persistent (res-pers). Fourth, we can split and combine resources using the composition operation $a \cdot b$ (res-sep). Fifth, if we own a resource a, then it is valid (res-valid).[7]

For these rules to be sound, a resource algebra cannot be any quadruple $(\mathcal{A}, \cdot, \mathcal{V}, |\_|_{\mathrm{pcore}})$. Instead, it needs to obey several axioms:[8]

Definition 31 (Resource Algebra)**.** *A resource algebra is a quadruple* $(\mathcal{A}, (\cdot) : \mathcal{A} \times \mathcal{A} \to \mathcal{A}, \mathcal{V} : \mathcal{A} \to \mathit{Prop}, |\_|_{\mathrm{pcore}} : \mathcal{A} \to \mathcal{A}^?)$ *satisfying:*

$$(a \cdot b) \cdot c = a \cdot (b \cdot c) \qquad (\textsc{ra-assoc})$$
$$a \cdot b = b \cdot a \qquad (\textsc{ra-comm})$$
$$|a|_{\mathrm{pcore}} \in \mathcal{A} \Rightarrow |a|_{\mathrm{pcore}} \cdot a = a \qquad (\textsc{ra-core-id})$$
$$|a|_{\mathrm{pcore}} \in \mathcal{A} \Rightarrow ||a|_{\mathrm{pcore}}|_{\mathrm{pcore}} = |a|_{\mathrm{pcore}} \qquad (\textsc{ra-core-idem})$$
$$|a|_{\mathrm{pcore}} \in \mathcal{A} \land a \preccurlyeq b \Rightarrow |b|_{\mathrm{pcore}} \in \mathcal{A} \land |a|_{\mathrm{pcore}} \preccurlyeq |b|_{\mathrm{pcore}} \qquad (\textsc{ra-core-mono})$$
$$a \cdot b \in \mathcal{V} \Rightarrow a \in \mathcal{V} \qquad (\textsc{ra-valid-op})$$

where $\mathcal{A}^? \triangleq \mathcal{A} \uplus \{\bot\} \qquad x \cdot \bot \triangleq \bot \cdot x \triangleq x \qquad a \preccurlyeq b \triangleq \exists c \in \mathcal{A}.\, b = a \cdot c$

[7] In this form, the rule res-valid only holds for non-step-indexed resources. For step-indexed resources (see §4.5.2), it holds for a step-indexed version of validity internalized into Iris.

[8] Iris Team, *The Iris 4.3 Reference*, 2024 [Iri24].

The composition—since it corresponds to separating conjunction—should be associative and commutative (ra-assoc, ra-comm). The core $|a|_{\mathrm{pcore}}$—since it corresponds to persistency—should be a part of a that can be duplicated (ra-core-id) and it should be idempotent (ra-core-idem). Moreover, it is required to be monotone (ra-core-mono). And, finally, the validity predicate $\mathcal{V}$, which holds whenever we own a resource, must also hold for smaller resources (ra-valid-op), because we can always discard resources with sep-weaken (in Fig. 2.2).

Unital resource algebras. Sometimes, it will be useful to work with resource algebras with a unit $\varepsilon : \mathcal{A}$, so-called *unital resource algebras*, where (1) $\varepsilon \cdot a = a$, (2) $\varepsilon \in \mathcal{V}$, and (3) $|\varepsilon|_{\mathrm{pcore}} = \varepsilon$. For them, the partial core $|a|_{\mathrm{pcore}}$ can always be completed to a total function $|a|_{\mathrm{core}}$ by picking ε in cases where $|a|_{\mathrm{pcore}} = \bot$. We write $M = (\mathcal{A}, (\cdot), \mathcal{V}, |\_|_{\mathrm{core}})$ for unital resource algebras.

Resource updates. The components of a resource algebra characterize how ghost state interacts with the separating conjunction $P * Q$, the persistency modality $\square P$, and what it means to own a resource. But when can we update a piece of ghost state to another (*i.e.,* when does $a \rightsquigarrow B$ hold)? As the definition of resource algebras shows, we do not get to choose an arbitrary update relation $a \rightsquigarrow B$ when we define a resource algebra. Instead, the updates of a resource algebra are already determined by the choice of validity $\mathcal{V}$ and composition $(\cdot)$.

To understand why, we have to take a closer look at validity. Validity $\mathcal{V}$ characterizes what it means to be a *valid* element of the resource algebra. For example, we will later see (1) that in the resource algebra of monotonically growing natural numbers the resource given by $\mathsf{mono}_\gamma(n) * \mathsf{lb}_\gamma(m)$ is valid iff $n \geq m$ and (2) that the resource given by $\mathsf{mono}_\gamma(n) * \mathsf{mono}_\gamma(n')$ is just invalid regardless of the choice of n and n' (because $\mathsf{mono}_\gamma(n)$ is exclusive like $\ell \mapsto v$, so there can only ever be one). Validity is implicitly maintained throughout proofs by Iris: we initially choose a valid resource (with res-alloc), we maintain validity (implicitly) when we update resources (with res-upd), and ownership of a resource entails it is valid (with res-valid).

Thus, one might think that we can update ownership of a resource a to an arbitrary other valid resource a'. But this is not the case. To understand why, we consider an example. Suppose we own the ghost state $\mathsf{mono}_\gamma(42)$. What prevents us from updating it to $\mathsf{mono}_\gamma(2)$, so what prevents us from violating the monotonicity baked into the ghost state? The resource behind $\mathsf{mono}_\gamma(2)$ is certainly valid, so we do not violate validity by going from $\mathsf{mono}_\gamma(42)$ to $\mathsf{mono}_\gamma(2)$. But in doing so, we ignore that there are potential frames of the ghost state $\mathsf{mono}_\gamma(42)$ which we would violate. For instance, suppose another part of the program initially owns $\mathsf{lb}_\gamma(41)$, so initially $\mathsf{mono}_\gamma(42) * \mathsf{lb}_\gamma(41)$ would be valid. If we now were to update $\mathsf{mono}_\gamma(42)$ to $\mathsf{mono}_\gamma(2)$, then the ghost state named γ suddenly would become invalid (since $2 \not\geq 41$).

To remedy this predicament, we account for the existence of a potential frame in the definition of *frame preserving updates* $a \rightsquigarrow B$:

Definition 32 (Frame Preserving Updates)**.**
It is possible to do a frame-preserving update from $a \in \mathcal{A}$ *to* $B \subseteq \mathcal{A}$*, written* $a \rightsquigarrow B$*, if* $\forall x_{\mathrm{f}} \in \mathcal{A}^?.\ a \cdot x_{\mathrm{f}} \in \mathcal{V} \Rightarrow \exists b \in B.\ b \cdot x_{\mathrm{f}} \in \mathcal{V}$*. We define* $a \rightsquigarrow b \triangleq a \rightsquigarrow \{b\}$*.*

Note that x_{f} could be $\bot$, so the frame-preserving update can also be applied to elements that have *no* frame. Those elements are called *exclusive resources.*[9]

[9] In the use of the term “exclusive”, we follow the Iris terminology here, where resources are exclusive if they have no frame [Jun+18b, Page 19]. A common alternative usage of the term is for resources that cannot be composed with *themselves* (*e.g.,* for points-to resources of individual locations).

4.2.1 Common Resource Algebras

Let us now fill the definition of resource algebras with life by instantiating it with different monoids. Many of the instances below are not very useful on their own, but they will be useful building blocks for deriving composite resource algebras with useful ghost theories such as the monotonically growing natural numbers or heaps (see §4.2.2). (We will not need all of these combinators and resulting ghost theories right away, but they are all used at some point in this dissertation, so we collect them here.)

Natural Number Addition $\boxed{(\mathbb{N}, +)}$

The monoid $(\mathbb{N}, +)$ forms a unital resource algebra:

$$m \cdot n \triangleq m + n \qquad m \in \mathcal{V} \triangleq \mathsf{True} \qquad |n|_{\mathrm{core}} \triangleq 0 \qquad \varepsilon \triangleq 0$$

$$\forall m, n.\, m \rightsquigarrow n \qquad m \preccurlyeq n \iff m \leq n$$

Natural Number Maximum $\boxed{(\mathbb{N}, \max)}$

The monoid $(\mathbb{N}, \max)$ forms a unital resource algebra:

$$m \cdot n \triangleq \max(m, n) \qquad m \in \mathcal{V} \triangleq \mathsf{True} \qquad |n|_{\mathrm{core}} \triangleq n \qquad \varepsilon \triangleq 0$$

$$\forall m, n.\, m \rightsquigarrow n \qquad m \preccurlyeq n \iff m \leq n$$

Fractions $\boxed{((0,1], +)}$

Just like for natural numbers, addition on the positive rational numbers $(\mathbb{Q}^+, +)$ forms a resource algebra (without a unit). We obtain a particularly useful resource algebra if we restrict to the interval $(0, 1]$:

$$q_1 \cdot q_2 \triangleq q_1 + q_2 \qquad q \in \mathcal{V} \triangleq q \leq 1 \qquad |q|_{\mathrm{pcore}} \triangleq \bot$$

$$0 < q_2 \leq q_1 \Rightarrow q_1 \rightsquigarrow q_2 \qquad q_1 \preccurlyeq q_2 \iff q_1 < q_2$$

Fractions do not have a core (*i.e.*, $|q|_{\mathrm{pcore}} = \bot$) since addition on positive rational numbers is never idempotent and, similarly, they do not have a unit.

Pairs $\boxed{M_1 \times M_2}$

The pair of two resource algebras M_1 and M_2 forms a resource algebra where

$$(x_1, x_2) \cdot (y_1, y_2) \triangleq (x_1 \cdot y_1, x_2 \cdot y_2) \qquad (x_1, x_2) \in \mathcal{V} \triangleq x_1 \in \mathcal{V} \land x_2 \in \mathcal{V}$$

$$\begin{array}{ll} |(x_1, x_2)|_{\mathrm{pcore}} \triangleq (|x_1|_{\mathrm{pcore}}, |x_2|_{\mathrm{pcore}}) & \textit{if } |x_1|_{\mathrm{pcore}} \neq \bot \textit{ and } |x_2|_{\mathrm{pcore}} \neq \bot \\ |(x_1, x_2)|_{\mathrm{pcore}} \triangleq \bot & \textit{otherwise} \end{array}$$

$$(x_1, x_2) \preccurlyeq (y_1, y_2) \iff x_1 \preccurlyeq y_1 \land x_2 \preccurlyeq y_2$$

$$x_1 \rightsquigarrow Y_1 \land x_2 \rightsquigarrow Y_2 \Rightarrow (x_1, x_2) \rightsquigarrow \{(y_1, y_2) \mid y_1 \in Y_1, y_2 \in Y_2\}$$

The partial core is only defined if the partial cores of both elements are defined. If both M_1 and M_2 have a unit, then the unit of the pair resource algebra is given by $\varepsilon \triangleq (\varepsilon_1, \varepsilon_2)$.

Options $\boxed{\text{option}(M)}$

The option resource algebra $\text{option}(M) \triangleq \text{Some}(a : \mathcal{A}_M) \mid \text{None}$ extends the resource algebra M with a unit. We typically write ε instead of None, and we commonly omit the Some injection.

$$\varepsilon \triangleq \text{None} \qquad \text{None} \cdot o \triangleq o \cdot \text{None} \triangleq o \qquad \text{Some}(a) \cdot \text{Some}(b) \triangleq \text{Some}(a \cdot b)$$

$$\text{None} \in \mathcal{V} \qquad \text{Some}(a) \in \mathcal{V} \Leftrightarrow a \in \mathcal{V}_M$$

$$\begin{aligned} |\text{None}|_{\text{core}} &\triangleq \text{None} & \\ |\text{Some}(a)|_{\text{core}} &\triangleq \text{None} & \textit{if } |a|_{\text{pcore}} = \bot \\ |\text{Some}(a)|_{\text{core}} &\triangleq \text{Some}(|a|_{\text{pcore}}) & \textit{otherwise} \end{aligned}$$

Exclusive Resource Algebra $\boxed{Ex(X)}$

The *exclusive resource algebra* $Ex(X) \triangleq \text{ex}(x : X) \mid ↯$ consists of an exclusive element $\text{ex}(x : X)$ and an invalid element $↯$. Its operations are pretty simple:

$$a \cdot b \triangleq ↯ \qquad a \in \mathcal{V} \triangleq a \neq ↯ \qquad |a|_{\text{pcore}} \triangleq \bot$$

$$\forall x, y.\ \text{ex}(x : X) \rightsquigarrow \text{ex}(y : X) \qquad a \preccurlyeq b \iff b = ↯$$

As the name indicates, the exclusive resource algebra ensures that there can always be only one, exclusive resource $\text{ex}(x : X)$. It gives us the right to update it freely (without violating the assumptions about the current ghost state of other program parts). It is similar to a points-to assertion $\ell \mapsto v$, which conveys the exclusive ownership to update ℓ.

Agreement Resource Algebra $\boxed{Ag(X)}$

The carrier type of the *agreement algebra* are finite, non-empty sets over X:

$$Ag(X) \triangleq \left\{A \in \mathcal{P}^{\text{fin}}(X) \,\middle|\, A \text{ non-empty}\right\} \qquad \text{ag}(x) \triangleq \{x\}$$

The operations on the elements of the resource algebra are given by:

$$A \cdot B \triangleq A \cup B \qquad A \in \mathcal{V} \triangleq \exists x : X.\, A = \{x\} \qquad |A|_{\text{pcore}} \triangleq A$$

$$\text{ag}(x : X) \rightsquigarrow \text{ag}(y : X) \iff x = y \qquad A \preccurlyeq B \iff A \subseteq B$$

The agreement resource algebra is in some sense the opposite of the exclusive resource algebra: its elements can be freely duplicated, but in exchange we can never update them.

$$\text{ag}(x) \cdot \text{ag}(x) = \text{ag}(x) \qquad \text{ag}(x) \cdot \text{ag}(y) \in \mathcal{V} \iff x = y$$

$$\text{ag}(x) \preccurlyeq \text{ag}(y) \iff x = y$$

The elements of the agreement resource algebra are similar to invariants $\boxed{P}^{\mathcal{N}}$ in that they can be freely duplicated, but once allocated, no one can change the statement P of the invariant $\mathcal{N}$.

Finite Functions $\boxed{K \xrightharpoonup{\text{fin}} M}$

For a countably infinite set K and a resource algebra M, the resource algebra of finite functions $K \xrightharpoonup{\text{fin}} M$ lifts the resource algebra structure of M to finite maps.

This resource algebra is used, for example, to obtain a ghost state version of heaps. The operations on the resource algebra are given by:

$$\begin{aligned}
f_1 \cdot f_2 &\triangleq [k \mapsto a \mid k \mapsto a \in f_1,\ k \notin \mathrm{dom}\, f_2] \\
&\quad \cup [k \mapsto a \mid k \mapsto a \in f_2,\ k \notin \mathrm{dom}\, f_1] \\
&\quad \cup [k \mapsto a \cdot b \mid k \mapsto a \in f_1,\ k \mapsto b \in f_2] \\
f \in \mathcal{V} &\triangleq \forall k \in \mathrm{dom}\, f.\, f(k) \in \mathcal{V}_M \\
|f|_{\mathrm{pcore}} &\triangleq \left[k \mapsto |a|_{\mathrm{pcore}} \,\middle|\, k \mapsto a \in f,\ |a|_{\mathrm{pcore}} \neq \bot\right] \\
\varepsilon &\triangleq \emptyset
\end{aligned}$$

and thus

$$\textsc{fin-fun-alloc}\quad \frac{G \text{ infinite} \qquad a \in \mathcal{V}}{\emptyset \rightsquigarrow \{[k \mapsto a] \mid k \in G\}} \qquad\qquad \textsc{fin-fun-update}\quad \frac{a \rightsquigarrow B}{f[k \mapsto a] \rightsquigarrow \{f[k \mapsto b] \mid b \in B\}}$$

Note that the update rule fin-fun-alloc is the first rule that truly makes use of the fact that frame preserving updates $a \rightsquigarrow B$ go from an element of the resource algebra a to *a set of elements* of the resource algebra B. We need a set of elements, because there is no single key k such that $\emptyset \rightsquigarrow [k \mapsto a]$, since k could always be used as part of the frame. In other words, since updates need to be frame preserving, picking specific keys is impossible, because we cannot ensure that they have not already been picked by some frame. We can, however, pick a set of elements: For every potential frame g, there exists some fresh key k in the *infinite set* G that is not contained in the *finite domain* of the frame g.

The authoritative resource algebra. Let us now turn to one of the most widely used resource algebras of Iris: the *authoritative resource algebra Auth*(M). The idea of this resource algebra is that it allows us to relate a global view of the entire resource with locally owned fragment resources. More specifically, for a unital resource algebra M, the elements of the resource algebra $Auth(M)$ are either the authoritative element $\bullet a$ or fragments $\circ b$. The relationship between the two kinds of elements is that, at any given point, all fragments $\circ b$ are *included* in the authoritative element $\bullet a$ (*i.e.,* $b \preccurlyeq a$). Moreover, fragments $\circ b$ can only be updated if the corresponding part of the authoritative element $\bullet a$ is also updated.

Authoritative Resource Algebra $\boxed{Auth(M)}$

The carrier type, its elements, and its operations are given by:

$$Auth(M) \triangleq \mathrm{option}(Ex(M)) \times M \qquad \bullet a \triangleq (\mathrm{ex}(a), \varepsilon_M) \qquad \circ b \triangleq (\varepsilon, b)$$

$$(x, a) \cdot (y, b) \triangleq (x \cdot y, a \cdot b)$$

$$\mathcal{V} \triangleq \{(\varepsilon, b) \mid b \in \mathcal{V}_M\} \cup \{(\mathrm{ex}(a), b) \mid b \preccurlyeq_M a \wedge a \in \mathcal{V}_M\} \qquad \varepsilon \triangleq (\varepsilon, \varepsilon_M)$$

$$|(x, a)|_{\mathrm{core}} \triangleq (\varepsilon, |a|_{\mathrm{core}})$$

The definition of $Auth(M)$ is not exactly trivial. It does, however, allow us to derive the properties depicted in Fig. 4.2. The fragment rules (frag-op, frag-core, frag-val, frag-unit, frag-incl) show that the resource algebra M embeds into the resource algebra $Auth(M)$ via the fragment connective $\circ$ (preserving all the *fragment rules*

FRAG-OP
$\circ(a \cdot b) = \circ a \cdot \circ b$

FRAG-CORE
$|\circ a|_{\mathrm{core}} = \circ |a|_{\mathrm{core}}$

FRAG-VAL
$\circ a \in \mathcal{V} \iff a \in \mathcal{V}_M$

FRAG-UNIT
$\circ \varepsilon_M = \varepsilon$

FRAG-INCL
$\circ a \preccurlyeq \circ b \iff a \preccurlyeq b$

authoritative element rules

AUTH-VAL
$\bullet a \in \mathcal{V} \iff a \in \mathcal{V}_M$

AUTH-EXCL
$\bullet a \cdot \bullet b \in \mathcal{V} \iff \mathsf{False}$

interaction rules

BOTH-VALID
$\bullet a \cdot \circ b \in \mathcal{V} \iff b \preccurlyeq_M a \wedge a \in \mathcal{V}_M$

BOTH-UPDATE
$(a, b) \rightsquigarrow_{\mathsf{L}} (a', b') \Rightarrow \bullet a \cdot \circ b \rightsquigarrow \bullet a' \cdot \circ b'$

Figure 4.2: Resource algebra rules of the authoritative resource algebra $Auth(M)$.

properties of the original algebra). The rules for the authoritative element (AUTH-VAL, AUTH-EXCL) show that the authoritative element connective $\bullet$ embeds the elements of M as exclusive elements (*i.e.,* there can never be two authoritative elements). The interaction rules are the most interesting rules. The rule BOTH-VALID says that, as explained above, every fragment must be included in the authoritative element. The rule BOTH-UPDATE states a condition on when it is possible to update a fragment inside and the authoritative element, the so-called *local update.*

Local Updates $\boxed{(a, b) \rightsquigarrow_{\mathsf{L}} (a', b')}$

It is possible to update a fragment and its corresponding part in the authoritative element whenever we can prove a *local update*:

$$(a, b) \rightsquigarrow_{\mathsf{L}} (a', b') \triangleq \forall x \in \mathcal{A}^?.\ a \in \mathcal{V}_M \wedge a = b \cdot x \Rightarrow a' \in \mathcal{V}_M \wedge a' = b' \cdot x$$

Observe that while ordinary updates $a \rightsquigarrow B$ just refer to validity and are hence trivial for some of the resource algebras we have seen (*e.g.,* for $(\mathbb{N}, +)$), the local updates also impose requirements that do not involve validity, requirements on the composition of the elements. Thus, we obtain some interesting local update rules for the resource algebras that we have discussed already:

$$\frac{n + m' = n' + m}{(n, m) \rightsquigarrow_{\mathsf{L}} (n', m')}(\mathbb{N}, +) \qquad \frac{n \le k}{(n, m) \rightsquigarrow_{\mathsf{L}} (k, k)}(\mathbb{N}, \max)$$

4.2.2 Common Ghost Theories

We now have all pieces together to define the ghost theories that we have encountered so far, monotonically growing numbers (§3.6) and heaps (§4.1). We additionally define two very useful ghost theories that we will use in the subsequent chapters.

Monotonically growing natural numbers. For the ghost theory of monotonically growing natural numbers, we use the resource algebra $\mathit{Auth}(\mathbb{N}, \max)$. We define:[10]

$$\mathsf{mono}_\gamma(n) \triangleq \boxed{\bullet n}^{\gamma} * \boxed{\circ n}^{\gamma} \qquad\qquad \mathsf{lb}_\gamma(n) \triangleq \boxed{\circ n}^{\gamma}$$

[10] It is important that the definition of $\mathsf{mono}_\gamma(n)$ includes the fragment $\boxed{\circ n}^{\gamma}$, because otherwise one could not prove the rule MAKE-BOUND as is, *i.e.,* $\mathsf{mono}_\gamma(n) \vdash \mathsf{mono}_\gamma(n) * \mathsf{lb}_\gamma(n)$. Without $\boxed{\circ n}^{\gamma}$, MAKE-BOUND would require an update to obtain the lower bound $\mathsf{lb}_\gamma(n)$ since $\circ n$ is not always included in $\bullet n$ (*i.e.,* $\circ n \not\preccurlyeq \bullet n$ for $n > 0$).

As an example, let us show the following two rules of their ghost theory:

Lemma 33.

UseBound
$$\mathsf{mono}_\gamma(n) * \mathsf{lb}_\gamma(m) \vdash n \geq m$$

BoundPers
$$\mathsf{lb}_\gamma(n) \vdash \Box\, \mathsf{lb}_\gamma(n)$$

Proof. UseBound. Observe that $\mathsf{mono}_\gamma(n) * \mathsf{lb}_\gamma(m) \vdash \boxed{\bullet n \cdot \circ n \cdot \circ m}^{\gamma}$. Thus $\bullet n \cdot \circ n \cdot \circ m \in \mathcal{V}$ and, hence, $\bullet n \cdot \circ m \in \mathcal{V}$. By definition of validity of the authoritative resource algebra, we obtain $m \preccurlyeq_{(\mathbb{N},\max)} n$ and $m \in \mathcal{V}_{(\mathbb{N},\max)}$. Thus, we obtain $m \leq n$, since $m \preccurlyeq_{(\mathbb{N},\max)} n \iff m \leq n$.

BoundPers. Observe that $|\circ n|_{\text{core}} = \circ|n|_{\text{core}} = \circ n$ by the definition of the core in the authoritative resource algebra and the $(\mathbb{N}, \max)$ resource algebra. □

Heaps. For the heap ghost theory, we use the resource algebra $\mathit{Heap}(K, V) \triangleq \mathit{Auth}(K \xrightarrow{\text{fin}} \mathit{Ex}(V))$. We define:

$$\mathsf{heap}^{\gamma}(h) \triangleq \boxed{\bullet[k \mapsto \mathsf{ex}(v) \mid k \mapsto v \in h]}^{\gamma} \qquad k \mapsto^{\gamma} v \triangleq \boxed{\circ[k \mapsto \mathsf{ex}(v)]}^{\gamma}$$

where $[k \mapsto \mathsf{ex}(v)]$ is a singleton map. We obtain the interaction rules:

$$\mathsf{True} \vdash \,\dot{\Rrightarrow}\, \exists \gamma.\, \mathsf{heap}^{\gamma}(\emptyset) \qquad k \mapsto^{\gamma} v * k \mapsto^{\gamma} w \vdash \mathsf{False}$$

$$\mathsf{heap}^{\gamma}(h) * k \mapsto^{\gamma} v \vdash \,\dot{\Rrightarrow}\, \mathsf{heap}^{\gamma}(h[k \mapsto w]) * k \mapsto^{\gamma} w$$

$$k \notin \operatorname{dom} h * \mathsf{heap}^{\gamma}(h) \vdash \,\dot{\Rrightarrow}\, \mathsf{heap}^{\gamma}(h[k \mapsto v]) * k \mapsto^{\gamma} v$$

$$\mathsf{heap}^{\gamma}(h) * k \mapsto^{\gamma} v \vdash h(k) = v$$

For the heap ghost theory in the state interpretation $\mathrm{SI}(h)$ (see §4.1), the ghost name γ is fixed globally to be some γ_{heap}, which we omit on paper when writing assertions such as $\ell \mapsto 42$.

Ghost variables. Ghost variables provide a points-to assertion $\gamma \Mapsto_q x$ akin to the regular points-to $\ell \mapsto v$, but which is *purely ghost* (*i.e.,* it is not directly tied to any part of the program such as the heap). Ghost variables are *fractional:*[11] they carry a fraction $0 < q \leq 1$ where $q = 1$ gives the right to arbitrarily update the ghost variable, and for $0 < q < 1$, two ghost points-to assertions must agree. Ghost variables satisfy the following rules:

[11] Analogously to ghost variables, one can obtain a fractional version of the heap resource algebra by choosing $((0, 1], +) \times \mathit{Ag}(\mathit{Val})$ in place of $\mathit{Ex}(\mathit{Val})$ for the construction of the heap. We will discuss a fractional heap in Part V (see §26.3.2).

$$\mathsf{True} \vdash \exists \gamma.\, \gamma \Mapsto_1 x \qquad \gamma \Mapsto_1 x \vdash \,\dot{\Rrightarrow}\, \gamma \Mapsto_1 y \qquad \gamma \Mapsto_q x * \gamma \Mapsto_{q'} y \vdash x = y$$

$$\gamma \Mapsto_q x * \gamma \Mapsto_{q'} x \dashv\vdash \gamma \Mapsto_{q+q'} x \quad \text{for } q, q' \in (0, 1]$$

For a co-domain X, the ghost points-to assertions are defined by picking the resource algebra $((0, 1], +) \times \mathit{Ag}(X)$ and defining:

$$\gamma \Mapsto_q x \triangleq \boxed{(q, \mathsf{ag}(x))}^{\gamma}$$

Credits. Another common resource algebra construction are *credits*. They consist of two pieces of ghost state: the actual credits $\mathsf{cred}_\gamma(n)$ and the full supply of credits $\mathsf{supp}_\gamma(m)$. Together, they obey the following rules:

$$\mathsf{cred}_\gamma(n_1) * \mathsf{cred}_\gamma(n_2) \dashv\vdash \mathsf{cred}_\gamma(n_1 + n_2) \qquad \mathsf{cred}_\gamma(n) * \mathsf{supp}_\gamma(m) \vdash m \geq n$$

$$\mathsf{True} \vdash \exists \gamma.\, \mathsf{supp}_\gamma(0) \qquad \frac{m, n \in \mathbb{N}}{\mathsf{supp}_\gamma(m) \vdash \mathsf{supp}_\gamma(m+n) * \mathsf{cred}_\gamma(n)}$$

$$\frac{m, n \in \mathbb{N}}{\mathsf{supp}_\gamma(m+n) * \mathsf{cred}_\gamma(n) \vdash {\mid\!\Rrightarrow} \mathsf{supp}_\gamma(m+n') * \mathsf{cred}_\gamma(n')}$$

To obtain the ghost theory, we pick the resource algebra $\mathit{Auth}(\mathbb{N}, +)$ and define:

$$\mathsf{cred}_\gamma(n) \triangleq \boxed{\circ n}^{\gamma} \qquad \mathsf{supp}_\gamma(m) \triangleq \boxed{\bullet m}^{\gamma}$$

4.3 The Base Logic

Equipped with resources, we can now move on to *the base logic*. It is the foundation upon which—as we have seen in the previous sections—everything can be built, including the weakest precondition (§4.1) and Hoare triples (§3.3). It consists of the following propositions and connectives:

$$\begin{array}{rcl} P, Q, R & ::= & \phi \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid \forall x : X.\, P(x) \mid \exists x : X.\, P(x) \\ & \mid & P * Q \mid P \mathrel{-\!\!*} Q \mid \square P \mid \triangleright P \mid \mu f x.\, P\, f\, x \mid {\mid\!\Rrightarrow} P \mid \mathsf{Own}\,(a) \end{array}$$

We have already encountered all of these connectives, except for the assertion $\mathsf{Own}\,(a)$. It is, as we will discuss shortly, a more primitive version of $\boxed{a}^{\gamma}$. We will now model each of these assertions by defining the type of *uniform predicates over a unital resource algebra M*, written $\mathit{UPred}(M)$.

Uniform Predicates $\boxed{\mathit{UPred}(M)}$

For a unital resource algebra $M = (\mathcal{A}, (\cdot), \mathcal{V}, |\_|_{\mathrm{core}})$, the type $\mathit{UPred}(M)$ consists of predicates over step-indices and M-resources that are down-closed in the step-index and up-closed in the resource:

$$\mathit{UPred}(M) \triangleq \{P \in \mathbb{P}(\mathbb{N} \times \mathcal{A}) \mid \forall (n, a) \in P.\, \forall m, b.\, m \leq n \Rightarrow a \preccurlyeq b \Rightarrow (m, b) \in P\}$$

The entailment relation $P \vdash Q$ is given by

$$P \vdash Q \triangleq \forall n, a.\, a \in \mathcal{V} \Rightarrow (n, a) \in P \Rightarrow (n, a) \in Q$$

and we define the logical connectives as depicted in Fig. 4.3. Many of these definitions are self-explanatory, since they merely lift the meta-level logical operation to $\mathit{UPred}(M)$-assertions (*e.g.*, ϕ and $P \wedge Q$). For implication $P \Rightarrow Q$, we ensure the closure properties hold by quantifying over smaller step-indices and larger resources. For the separating conjunction $P * Q$, we use the resource composition operation $(\cdot)$ to separate the resource a into two parts a_1 and a_2 for P and Q. For the magic wand $P \mathrel{-\!\!*} Q$, we add a resource b satisfying P to prove Q. For persistency $\square P$, we use the core operation $|a|_{\mathrm{core}}$. For the update modality ${\mid\!\Rrightarrow} P$, we effectively perform a frame-preserving update on the current resource a, and for the ownership connective $\mathsf{Own}\,(a)$, we say that the current resource b is (at least) a.

$$
\begin{aligned}
\phi &\triangleq \{(n,a) \mid \phi\}\\
P \wedge Q &\triangleq \{(n,a) \mid (n,a) \in P \wedge (n,a) \in Q\}\\
P \vee Q &\triangleq \{(n,a) \mid (n,a) \in P \vee (n,a) \in Q\}\\
P \Rightarrow Q &\triangleq \{(n,a) \mid \forall m \leq n. \forall b \succcurlyeq a.\, b \in \mathcal{V} \Rightarrow (m,b) \in P \Rightarrow (m,b) \in Q\}\\
\forall x : X.\, P(x) &\triangleq \{(n,a) \mid \forall x : X.\, (n,a) \in P(x)\}\\
\exists x : X.\, P(x) &\triangleq \{(n,a) \mid \exists x : X.\, (n,a) \in P(x)\}\\
P * Q &\triangleq \{(n,a) \mid \exists a_1, a_2.\, a = a_1 \cdot a_2 \wedge (n,a_1) \in P \wedge (n,a_2) \in Q\}\\
P \mathrel{-\!\!*} Q &\triangleq \{(n,a) \mid \forall m \leq n. \forall b.\, a \cdot b \in \mathcal{V} \Rightarrow (m,b) \in P \Rightarrow (m, a \cdot b) \in Q\}\\
\square P &\triangleq \{(n,a) \mid (n, |a|_{\mathrm{core}}) \in P\}\\
\triangleright P &\triangleq \{(n,a) \mid \forall m < n.\, (m,a) \in P\}\\
\mathrel{|\!\!\Rrightarrow} P &\triangleq \left\{(n,a) \,\middle|\, \forall m \leq n. \forall a_f.\, a \cdot a_f \in \mathcal{V} \Rightarrow \exists b.\, b \cdot a_f \in \mathcal{V} \wedge (m,b) \in P\right\}\\
\mathsf{Own}\,(a) &\triangleq \{(n,b) \mid a \preccurlyeq b\}
\end{aligned}
$$

Figure 4.3: Connectives of the base logic.

Soundness. Equipped with the definition of uniform predicates, one can prove the entailment rules, which we have encountered in previous sections. Concretely, one can prove the rules depicted in Fig. 4.4. As an example, let us consider Löb induction:

Lemma 34. *If* $\triangleright P \vdash P$*, then* $\vdash P$.

Proof. Let $a \in \mathcal{V}$. We have to show $\forall n.\, (n,a) \in P$. We proceed by strong induction on n, which gives us $\forall m < n.\, (m,a) \in P$. Thus, we know $(n,a) \in \triangleright P$ given the definition of $\triangleright P$. Since $\triangleright P \vdash P$, we conclude $(n,a) \in P$. □

Ghost state. Notably, none of the rules in Fig. 4.4 mention ghost state or resources in any form. Let us have a closer look at resources in *UPred*(M). Instead of the ghost state connective $\boxed{a}^{\gamma}$, the connectives in Fig. 4.3 include a more primitive ownership connective, $\mathsf{Own}\,(a)$, for resources a from a fixed resource algebra M (from which we will derive $\boxed{a}^{\gamma}$ shortly). It obeys the following rules:

own-empty
$$\mathsf{True} \vdash \mathsf{Own}\,(\varepsilon)$$

own-upd
$$\frac{a \rightsquigarrow B}{\mathsf{Own}\,(a) \vdash \mathrel{|\!\!\Rrightarrow} \exists b \in B.\, \mathsf{Own}\,(b)}$$

own-valid
$$\mathsf{Own}\,(a) \vdash a \in \mathcal{V}$$

own-pers
$$\mathsf{Own}\,(a) \vdash \square\, \mathsf{Own}\,(|a|_{\mathrm{core}})$$

own-sep
$$\mathsf{Own}\,(a) * \mathsf{Own}\,(b) \dashv\vdash \mathsf{Own}\,(a \cdot b)$$

We always own the empty resource ε (own-empty), and we can use a frame preserving update $a \rightsquigarrow B$ to update a resource a that we own (own-upd). If we own a resource a, then it is valid[12] (own-valid) and we persistently own its core (own-pers). Lastly, we can use the resource composition of M to split and combine resources (own-sep).

[12] In this form, the rule own-valid only holds for non-step-indexed resources. For step-indexed resources (see §4.5.2), it holds for a step-indexed version of validity internalized into Iris.

Note that there is no rule for timelessness of $\mathsf{Own}\,(a)$ among these rules. When is a resource timeless? The intuitive answer is that most “normal resources” (*e.g.*, heaps, credits, ghost variables, monotonically growing numbers,

ent-refl
$$P \vdash P$$

ent-trans
$$\dfrac{P \vdash Q \qquad Q \vdash R}{P \vdash R}$$

pure-intro
$$\dfrac{\phi}{P \vdash \phi}$$

from-pure
$$\dfrac{P \vdash \phi \qquad \phi \Rightarrow (P \vdash Q)}{P \vdash Q}$$

Figure 4.4: Rules of the base logic without the ownership assertion.

and-elim-l
$$P \wedge Q \vdash P$$

and-elim-r
$$P \wedge Q \vdash Q$$

and-intro
$$\dfrac{P \vdash Q \qquad P \vdash R}{P \vdash Q \wedge R}$$

or-intro-l
$$P \vdash P \vee Q$$

or-intro-r
$$Q \vdash P \vee Q$$

or-elim
$$\dfrac{P \vdash R \qquad Q \vdash R}{P \vee Q \vdash R}$$

all-intro
$$\dfrac{\forall x : X.\, (P \vdash Q(x))}{P \vdash \forall x : X.\, Q(x)}$$

all-elim
$$\dfrac{a : X}{(\forall x : X.\ P(x)) \vdash P(a)}$$

exist-intro
$$\dfrac{a : X \qquad P \vdash Q(a)}{P \vdash \exists x : X.\, Q(x)}$$

exist-elim
$$\dfrac{\forall x : X.\, (P(x) \vdash Q)}{\exists x : X.\, P(x) \vdash Q}$$

wand-intro
$$\dfrac{P * Q \vdash R}{P \vdash Q \mathrel{-\!\!*} R}$$

wand-elim
$$\dfrac{P \vdash Q \mathrel{-\!\!*} R}{P * Q \vdash R}$$

sep-weaken
$$P * Q \vdash P$$

sep-true
$$P \vdash P * \mathsf{True}$$

sep-comm
$$P * Q \vdash Q * P$$

sep-split
$$\dfrac{P \vdash P' \qquad Q \vdash Q'}{P * Q \vdash P' * Q'}$$

sep-assoc
$$P * (Q * R) \dashv\vdash (P * Q) * R$$

pers-elim
$$\Box P \vdash P$$

pers-mono
$$\dfrac{P \vdash Q}{\Box P \vdash \Box Q}$$

pers-pure
$$\phi \vdash \Box \phi$$

pers-all
$$\forall x : X.\ \Box P(x) \vdash \Box \forall x : X.\, P(x)$$

pers-exists
$$\Box \exists x : X.\, P(x) \vdash \exists x : X.\ \Box P(x)$$

pers-and-sep
$$(\Box P) \wedge Q \vdash (\Box P) * Q$$

pers-idemp
$$\Box P \vdash \Box \Box P$$

later-intro
$$P \vdash \triangleright P$$

later-mono
$$\dfrac{P \vdash Q}{\triangleright P \vdash \triangleright Q}$$

Löb
$$\dfrac{\triangleright P \vdash P}{\vdash P}$$

later-sep
$$\triangleright(P * Q) \dashv\vdash \triangleright P * \triangleright Q$$

later-exists
$$\dfrac{X \text{ non-empty}}{\triangleright(\exists x : X.\, P(x)) \dashv\vdash \exists x : X.\ \triangleright P(x)}$$

later-all
$$\triangleright(\forall x : X.\, P(x)) \dashv\vdash \forall x : X.\ \triangleright P(x)$$

later-pers
$$\triangleright \Box P \dashv\vdash \Box \triangleright P$$

upd-return
$$P \vdash \mathrel{|\!\!\Rrightarrow} P$$

upd-bind
$$\mathrel{|\!\!\Rrightarrow} P * (P \mathrel{-\!\!*} \mathrel{|\!\!\Rrightarrow} Q) \vdash \mathrel{|\!\!\Rrightarrow} Q$$

etc.) are timeless. The formal answer requires a non-trivial amount of background knowledge about the step-indexed types and resources of Iris (see §4.5.2 for a brief introduction, and Jung et al.[13] for a thorough discussion). In a nutshell, a resource construction becomes non-timeless when we define the resource itself using step-indexing (as with the invariants in §4.5.2).

[13] Jung et al., "Iris from the ground up: A modular foundation for higher-order concurrent separation logic", 2018 [Jun+18b].

Iris Propositions $\boxed{iProp}$

From the general construction of $UPred(M)$, we obtain the type of Iris propositions *iProp* and the ghost state ownership connective $\boxed{a}^{\gamma}$ by picking specific resource algebra $\mathcal{M}$ that combines the resource algebras that we want to use:

$$iProp \triangleq UPred(\mathcal{M})$$

Concretely, suppose we want to use the resource algebras $M_1, M_2, \ldots, M_n$ (*e.g.,* heaps, ghost variables, ...).[14] Then, we pick the resource algebra

$$\mathcal{M} \triangleq Map(M_1) \times Map(M_2) \times \cdots \times Map(M_n) \quad \text{where} \quad Map(M) \triangleq \mathbb{N} \xrightarrow{\text{fin}} M$$

[14] To support modularity in the choice of resource algebras, the Rocq implementation uses a definition of $\mathcal{M}$ based on dependent types and type classes. We simplify this construction here by assuming there is a specific collection of resource algebras $M_1, M_2, \ldots, M_n$ that we wish to use.

Its elements are tuples of finite maps $Map(M)$ to the elements of the resource algebras $M_1, M_2, \ldots, M_n$. The reader may wonder why we are using finite maps here instead of directly the tuple $M_1 \times M_2 \times \cdots \times M_n$. The answer is that, by using finite maps, we can associate ghost state with a name $\gamma : \mathbb{N}$ and have multiple instances of the same ghost state (*e.g.,* $\mathsf{mono}_{\gamma_1}(n_1)$, $\mathsf{mono}_{\gamma_2}(n_2)$, ...).

We define the ghost ownership connective $\boxed{a}^{\gamma}$ as

$$\boxed{a : M_i}^{\gamma} \triangleq \mathsf{Own}\,(\emptyset, \ldots, \emptyset, [\gamma \mapsto a], \emptyset, \ldots, \emptyset)\,,$$

meaning as a large tuple of empty maps $\emptyset$ where the i-th component[15] is set to the singleton map $[\gamma \mapsto a]$. We obtain the standard ghost state rules:

[15] Note that the connective $\boxed{a}^{\gamma}$ needs to know the position i of the resource algebra of a to know *which* component of the tuple to set. For this reason, we write $\boxed{a : M_i}^{\gamma}$ here. In Rocq, type classes are used to maintain this information. In the remainder of this dissertation, we gloss over this detail and simply write $\boxed{a}^{\gamma}$.

RES-ALLOC
$$\frac{a \in \mathcal{V}_{M_i}}{\vdash\, \Rrightarrow \exists \gamma.\, \boxed{a : M_i}^{\gamma}}$$

RES-SEP
$$\boxed{a : M_i}^{\gamma} * \boxed{b : M_i}^{\gamma} \dashv\vdash \boxed{a \cdot b : M_i}^{\gamma}$$

RES-VALID
$$\boxed{a : M_i}^{\gamma} \vdash a \in \mathcal{V}_{M_i}$$

RES-UPD
$$\frac{a \rightsquigarrow_{M_i} B}{\boxed{a : M_i}^{\gamma} \vdash\, \Rrightarrow \exists b \in B.\, \boxed{b : M_i}^{\gamma}}$$

RES-PERS
$$\frac{|a|_{\mathrm{pcore}} \neq \bot}{\boxed{a : M_i}^{\gamma} \vdash \Box\, \boxed{|a|_{\mathrm{pcore}} : M_i}^{\gamma}}$$

Guarded Fixpoints. The last missing piece of the base logic are *guarded fixpoints* $\mu\, f x.\, P\, f\, x$. Let us sketch their construction for predicates of type $X \to UPred(M)$. For a function $P : (X \to UPred(M)) \to (X \to UPred(M))$, we define $(\mu\, f y.\, P\, f\, y)(x) \triangleq \{(n, a) \mid (n, a) \in f_n(x)\}$ where:

$$f_0(x) \triangleq P(\lambda\_.\, \mathsf{True})(x) \qquad\qquad f_{n+1}(x) \triangleq P(f_n)(x)$$

If every recursive occurrence of f in P is guarded by a later, then—at step-index n—one cannot distinguish between f_n and $P(f_n)$. Thus, with $\mu\, f y.\, P\, f\, y$, we obtain a fixpoint of P in the following sense:

Lemma 35. *Let every occurrence of f in P be guarded by a later modality. Then $\mu\, f y.\, P\, f\, y$ is a fixpoint, meaning $\forall x.\, (\mu\, f y.\, P\, f\, y)(x) \dashv\vdash P(\mu\, f y.\, P\, f\, y)(x)$.*

4.4 Adequacy

Now that we have seen the model of Iris propositions and the model of the weakest precondition, one interesting question remains: "What do we prove when we prove weakest preconditions in Iris?" This question is answered by the adequacy results of Iris. They can be used to lift results proven in Iris to results about programs at the meta level (*i.e.,* outside of Iris).[16] We keep the discussion of adequacy brief here and discuss it in more depth in Part III (see §14).

The main adequacy theorem of Iris is the following:[17]

Theorem 36 (Adequacy). *If* $\vdash \{\mathsf{True}\}\ e\ \{v.\, \phi(v)\}$, *then* e *is safe to execute in any heap* h *and all possible return values* v *satisfy the pure postcondition* $\phi(v)$.

It ensures that when we prove a Hoare triple over a HeapLang program e in Iris, then e is safe to execute and only terminates in values satisfying the postcondition. The theorem is based on two language-generic results, adequacy of the program logic (Lemma 37) and soundness of the base logic (Lemma 38). The adequacy theorem of the program logic states states that a weakest precondition for e ensures that it is safe to execute e in the current heap h for n steps. It is instantiated for HeapLang to obtain the language-specific Theorem 36. The soundness theorem for the base logic states that pure propositions (under a potential nesting of later modalities and update modalities) are true at the meta level. It is used by Lemma 37 to extract the postcondition (and other pure propositions) from the interleaving of updates and later modalities in the weakest precondition (see §14.1).

Lemma 37 (Adequacy of the Program Logic).
If $(e, h) \rightsquigarrow^n (e', h')$ *and* $\vdash \Rrightarrow \mathit{SI}(h) * \mathbf{wp}\ e\ \{v.\, \phi(v)\}$, *then* (e', h') *is either progressive or* e' *is a value* v *and* $\phi(v)$ *holds.*

Lemma 38 (Soundness of the Base Logic). *If* $\vdash (\Rrightarrow \triangleright)^n \phi$, *then* ϕ *holds.*

[16] Technically, Theorem 36 and Lemma 37 need to quantify over the ghost name γ_{heap} that is chosen for the heap. We have omitted it here for simplicity. In general, for resource algebras where we only want to have a single, global instance like for the heap, obtaining a dedicated ghost name γ is not a problem.

[17] This adequacy theorem holds for HeapLang specifically. Using the language-generic lemmas Lemma 37 and Lemma 38 below, it is also possible to obtain similar adequacy results for other languages.

4.5 Impredicative Invariants and Fancy Updates

Notably, what is missing from our discussion above is how invariants (and timelessness; see §4.5.1) are integrated into the program logic. In §3.4.1, we have—as a first approximation—sketched the model of Iris as:

$$\mathit{iProp} = \mathit{Inv} \to \mathit{Heap} \to \mathit{Prop} \qquad\qquad \mathit{Inv} = \mathbb{N} \xrightarrow{\text{fin}} \mathit{iProp}$$

to highlight the circular dependency between invariants and propositions in Iris. But—as we have just seen—that is not actually how *iProp* is defined. Instead, the heaps are a resource algebra in Iris. And—as we will see now—so are the invariants! That is, since version 2 of Iris, invariants are expressed as resources via an elaborate resource construction described by Jung et al.[18] To explain how "invariants as resources" works, we will recall the most important parts of this construction below (§4.5.1). Our primary goal, however, will be to shed some light on how Iris uses step-indexing to break the circularity between invariants and propositions (§4.5.2). Readers not interested in the construction can skip (§4.5.1) and proceed directly to (§4.5.2).

[18] Jung et al., "Higher-order ghost state", 2016 [Jun+16].

wp-inv-alloc
$$\frac{P * \boxed{R}^{\mathcal{N}} \vdash \mathbf{wp}^{\mathcal{E}}\ e\ \{v.\ Q(v)\}}{P * R \vdash \mathbf{wp}^{\mathcal{E}}\ e\ \{v.\ Q(v)\}}$$

wp-inv-open
$$\frac{P * \triangleright R \vdash \mathbf{wp}^{\mathcal{E}\setminus\mathcal{N}}\ e\ \{v.\ \triangleright R * Q(v)\} \qquad \mathsf{atomic}(e) \qquad \mathcal{N} \subseteq \mathcal{E}}{P * \boxed{R}^{\mathcal{N}} \vdash \mathbf{wp}^{\mathcal{E}}\ e\ \{v.\ Q(v)\}}$$

wp-timeless-strip
$$\frac{\mathsf{timeless}(Q) \qquad P * Q \vdash \mathbf{wp}^{\mathcal{E}}\ e\ \{v.\ R(v)\}}{P * \triangleright Q \vdash \mathbf{wp}^{\mathcal{E}}\ e\ \{v.\ R(v)\}}$$

Figure 4.5: Program logic rules for invariants and timelessness.

fancy-inv
$$\frac{\mathcal{N} \subseteq \mathcal{E}}{\boxed{P}^{\mathcal{N}} \vdash {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}\setminus\mathcal{N}}\ \triangleright P * (\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}{\Rrightarrow}^{\mathcal{E}}\ \mathsf{True})}$$

fancy-inv-alloc
$$P \vdash {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}}\ \boxed{P}^{\mathcal{N}}$$

fancy-return
$$P \vdash {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}}\ P$$

fancy-bind
$${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}\ P * (P \mathrel{-\!\!*} {}^{\mathcal{E}_2}{\Rrightarrow}^{\mathcal{E}_3}\ Q) \vdash {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_3}\ Q$$

fancy-upd
$${\Rrightarrow} P \vdash {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}}\ P$$

fancy-timeless-later-elim
$$\frac{\mathsf{timeless}(P)}{\triangleright P \vdash {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}}\ P}$$

wp-fancy
$${}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}}\ \mathbf{wp}^{\mathcal{E}}\ e\ \{v.\ Q(v)\} \vdash \mathbf{wp}^{\mathcal{E}}\ e\ \{v.\ Q(v)\}$$

wp-fancy-atomic
$$\frac{\mathsf{atomic}(e)}{{}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}\ \mathbf{wp}^{\mathcal{E}_2}\ e\ \left\{v.\ {}^{\mathcal{E}_2}{\Rrightarrow}^{\mathcal{E}_1}\ Q(v)\right\} \vdash \mathbf{wp}^{\mathcal{E}_1}\ e\ \{v.\ Q(v)\}}$$

Figure 4.6: Proof rules for the fancy update ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}\ P$.

4.5.1 Fancy Updates

Recall the rules of the program logic for interacting with invariants and timeless propositions, depicted in Fig. 4.5: we allocate invariants with wp-inv-alloc, we open invariants with wp-inv-open, and we eliminate laters from timeless propositions with wp-timeless-strip. They all rest on the central piece of Iris's invariants mechanism: the *fancy update* ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}\ P$. The fancy update ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}\ P$ is an update modality like ${\Rrightarrow} P$ for ghost state updates, but it is—well—a little more fancy: it supports accessing invariants and interacting with timeless propositions.

Intuitively, a fancy update ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}\ P$ means that P holds after *opening* all the invariants which are in $\mathcal{E}_1$, performing some ghost state updates, and then *closing* all the invariants that are in $\mathcal{E}_2$. To reason about fancy updates, we use the rules depicted in Fig. 4.6. The most important rule is fancy-inv. It allows one to open the invariant $\mathcal{N}$ (as long as the namespace $\mathcal{N}$ is contained in the mask $\mathcal{E}$). When we use fancy-inv, we get $\triangleright P$ and, additionally, a magic wand $(\triangleright P \mathrel{-\!\!*} {}^{\mathcal{E}\setminus\mathcal{N}}{\Rrightarrow}^{\mathcal{E}}\ \mathsf{True})$ to close the invariant again (*i.e.*, to restore the mask to $\mathcal{E}$

again). Together with the other rules, we can use this rule to open, modify, and then close invariants again all as part of proving a fancy update ${}^{\mathcal{E}}{\mid\!\Rrightarrow}^{\mathcal{E}} Q$.

The rules fancy-return and fancy-bind mirror those of the update modality ${\mid\!\Rrightarrow} P$ and enable similar compositionality (*e.g.*, we can transitively compose fancy updates). The rule fancy-timeless-later-elim allows us to eliminate a later from a timeless proposition, and the rule wp-fancy allows us to execute a fancy update at a weakest precondition. Taken together, both rules yield the timeless stripping rule wp-timeless-strip. The rule wp-fancy-atomic allows us to change the mask of the weakest precondition around an atomic expression e. It, together with fancy-inv, can be used to derive the rule wp-inv-open. The rule fancy-inv-alloc allows us to allocate invariants, underlying wp-inv-alloc.

To obtain the rules in Fig. 4.6 (and hence Fig. 4.5), all one has to do is replace the update modality "${\mid\!\Rrightarrow}$" in the definition of the weakest precondition (see §4.1) with fancy updates "${}^{\mathcal{E}_1}{\mid\!\Rrightarrow}^{\mathcal{E}_2}$". The only subtlety is the choice of the masks. Here, we refer the reader to Jung et al.[19] for a careful discussion of which mask $\mathcal{E}$ should be used where in the definition of the weakest precondition.

[19] Jung et al., "Iris from the ground up: A modular foundation for higher-order concurrent separation logic", 2018 [Jun+18b].

Model of the fancy update. To get to the circularity between invariants and propositions, we focus on the model of the fancy update. The fancy update ${}^{\mathcal{E}_1}{\mid\!\Rrightarrow}^{\mathcal{E}_2} P$ is defined as follows:

$$ {}^{\mathcal{E}_1}{\mid\!\Rrightarrow}^{\mathcal{E}_2} P \triangleq \mathsf{wsat} * \boxed{\mathcal{E}_1}^{\gamma_{\mathrm{en}}} \mathrel{-\!\!*} {\mid\!\Rrightarrow} \diamond (\mathsf{wsat} * \boxed{\mathcal{E}_2}^{\gamma_{\mathrm{en}}} * P) $$

Let us unpack this definition step by step. First, it uses a proposition called *world satisfaction* wsat (discussed below) that will store the invariants. We assume it initially and then restore it again at the end of the fancy update.[20] Alongside world satisfaction wsat, the fancy update maintains a resource $\boxed{\mathcal{E}}^{\gamma_{\mathrm{en}}}$. This piece of ghost state tracks which invariants are currently closed. We will use it, together with an additional piece of ghost state $\boxed{\mathcal{E}}^{\gamma_{\mathrm{dis}}}$ in the definition of wsat to distinguish between open and closed invariants.

[20] This pattern of assuming world satisfaction and then re-establishing it again after an update is similar to how the state interpretation $\mathrm{SI}(h)$ is treated in the definition of the weakest precondition (see §4.1). It gives one temporary access to wsat when proving a fancy update before it must be given back.

Apart from world satisfaction and the ghost state for closed invariants, the fancy update is *almost* just a regular update ${\mid\!\Rrightarrow} P$. The only extra addition is a new modality $\diamond P$, which is for timelessness. We first explain the timelessness modality and then return to world satisfaction and invariants.

Timelessness. Intuitively, a proposition P is timeless if its complete behavior is determined by its behavior at step-index 0. In terms of the base logic, we can express this property as:[21]

$$ \mathsf{timeless}(P) \triangleq \triangleright P \vdash \diamond P \quad \textit{where} \quad \diamond P \triangleq \triangleright \mathsf{False} \lor P $$

[21] An alternative, more direct definition in the model would be $\mathsf{timeless}(P) \triangleq \forall n.\, a \in \mathcal{V} \Rightarrow (0, a) \in P \Rightarrow (n, a) \in P$. This definition works for the non-step-indexed resources. For step-indexed resources, discussed in §4.5.2, one can define a similar property directly in the model.

The definition of $\diamond P$ means that the step-index is 0 (the left branch) or P holds currently (the right branch). Thus, a proposition P is timeless if $\triangleright P$ means the step-index is 0 or P holds already at the current step-index.

The key rules that we get for timelessness—and the reason why the modality $\diamond P$ is integrated into the definition of a fancy update—are:

later-timeless-modality
$$ \frac{\mathsf{timeless}(P)}{\triangleright P \vdash \diamond P} $$

timeless-upd-elim
$$ \diamond P \vdash {}^{\mathcal{E}}{\mid\!\Rrightarrow}^{\mathcal{E}} P $$

The rule later-timeless-modality allows us one to turn a later modality into a timeless modality for timeless propositions. The rule timeless-upd-elim allows one to eliminate a timeless modality at a fancy update. Both together yield the timelessness rule for fancy updates, fancy-timeless-later-elim (in Fig. 4.6).

The world satisfaction ghost theory. Let us now turn to world satisfaction. We start with the ghost theory that ties invariants to world satisfaction wsat. In this ghost theory, invariants will have a *name* $\iota : \mathbb{N}$ instead of a namespace $\mathcal{N}$. The namespace invariants are obtained as $\boxed{P}^{\mathcal{N}} \triangleq \exists \iota \in \mathcal{N}.\, \boxed{P}^{\iota}$.

The ghost theory of world satisfaction has three interesting rules:

inv-alloc-wsat
$$\mathsf{wsat} * (\triangleright P) * \mathcal{E}\ \mathsf{infinite} \vdash \Rrightarrow \exists \iota \in \mathcal{E}.\, \boxed{P}^{\iota} * \mathsf{wsat}$$

inv-open-wsat
$$\boxed{P}^{\iota} * \mathsf{wsat} * \boxed{\{\iota\}}^{\gamma_{\mathrm{en}}} \vdash \Rrightarrow (\triangleright P) * \mathsf{wsat} * \boxed{\{\iota\}}^{\gamma_{\mathrm{dis}}}$$

inv-close-wsat
$$\boxed{P}^{\iota} * \mathsf{wsat} * (\triangleright P) * \boxed{\{\iota\}}^{\gamma_{\mathrm{dis}}} \vdash \Rrightarrow \mathsf{wsat} * \boxed{\{\iota\}}^{\gamma_{\mathrm{en}}}$$

The rule inv-alloc-wsat allows us to allocate a new invariant P if we own $\triangleright P$. The rule inv-open-wsat allows us to open the invariant ι and obtain $\triangleright P$. The rule inv-close-wsat allows us the close the invariant ι by returning $\triangleright P$. To use inv-open-wsat, we need to know that ι is currently enabled and we get back a token $\boxed{\{\iota\}}^{\gamma_{\mathrm{dis}}}$ witnessing that ι is now disabled. To use inv-close-wsat, we need to know that ι is currently disabled and we get back a token $\boxed{\{\iota\}}^{\gamma_{\mathrm{en}}}$ witnessing that ι is now enabled.

The ghost state for the enabled and disabled invariants are sets of invariant names $\mathcal{E}$ with $\uplus$ as the monoid operation. More concretely, for the enabled invariants γ_{en}, we pick the resource algebra $(\mathbb{P}(\mathbb{N}), \uplus)$ (*i.e.,* sets of natural numbers with disjoint union as the composition) and for the disabled invariants, we pick the resource algebra $(\mathbb{P}^{\mathrm{fin}}(\mathbb{N}), \uplus)$ (*i.e., finite* sets of natural numbers with disjoint union as the composition). At any given point, we can have infinitely many enabled tokens (*e.g.,* with $\boxed{\top}^{\gamma_{\mathrm{en}}}$), but there can only ever be finitely many disabled tokens (*i.e.,* $\boxed{\{\iota\}}^{\gamma_{\mathrm{dis}}}$), which is sufficient because there will only ever be a finite number of allocated invariants at any given time.

An attempt at world satisfaction. Given the ghost theory of world satisfaction, let us now attempt to define wsat and the invariants $\boxed{P}^{\iota}$:

$$\begin{aligned}
\boxed{P}^{\iota} &\triangleq \boxed{\circ[\iota \mapsto \mathsf{ag}(P)]}^{\gamma_{\mathrm{inv}}} \\
\mathsf{wsat} &\triangleq \exists I : \mathbb{N} \xrightarrow{\mathrm{fin}} \mathit{iProp}.\, \boxed{\bullet[\iota \mapsto \mathsf{ag}(P) \mid \iota \mapsto P \in I]}^{\gamma_{\mathrm{inv}}} \\
&\qquad\qquad * \mathop{\Large *}_{\iota \mapsto P \in I} (\triangleright P * \boxed{\{\iota\}}^{\gamma_{\mathrm{dis}}}) \vee \boxed{\{\iota\}}^{\gamma_{\mathrm{en}}}
\end{aligned}$$

In this definition, we introduce an additional piece of ghost state γ_{inv} which connects the invariant connective $\boxed{P}^{\iota}$ to world satisfaction. The idea is that we use the *agreement resource algebra Ag* (see §4.2) to synchronize the P in $\boxed{P}^{\iota}$ with the propositions P that we work with in the definition of world satisfaction (*i.e.,* those for which there is a ι such that $\iota \mapsto P \in I$). And for every invariant $\iota \mapsto P \in I$ (*i.e.,* every currently allocated invariant), we are in one of two states: either (1) the invariant is currently closed, which means we store $\triangleright P$ and $\boxed{\{\iota\}}^{\gamma_{\mathrm{dis}}}$ in world satisfaction, or (2) the invariant is currently open, which means we only store the token $\boxed{\{\iota\}}^{\gamma_{\mathrm{en}}}$ in world satisfaction. In either case, we can facilitate the token exchange witnessed by inv-open-wsat and inv-close-wsat.

Sadly, this model of world satisfaction and invariants is broken!

4.5.2 Step-Indexed Types and Resources

While it is not necessarily apparent at first glance, the naïve definition of invariants described above contains a circularity. To see where it is, we have to take a closer look at the resource algebras involved in the construction:

$$(\mathbb{P}(\mathbb{N}), \uplus) \qquad (\mathbb{P}^{\text{fin}}(\mathbb{N}), \uplus) \qquad \mathit{Auth}(\mathbb{N} \xrightarrow{\text{fin}} \mathit{Ag}(\mathit{iProp}))$$

The first two resource algebras are (1) sets of natural numbers with disjoint union as the composition and (2) finite sets of natural numbers with disjoint union as the composition. Neither of these pose any kind of trouble. The third one, sometimes called an "agreement map", is a construction similar to the heap (in §4.2), but it does not allow changing the per-location points-to assertions once they have been allocated. Instead, in this resource algebra, all points-to assertions are persistent, allowing us to share them freely in the logic once they have been allocated.

This resource algebra is the problem. Recall (from §4.3) that the model of Iris propositions *iProp* is $\mathit{UPred}(\mathcal{M})$ for a specific resource algebra $\mathcal{M}$. If we now refer to *iProp* in the resource algebra $\mathcal{M}$, then we have constructed a cycle:

$$\begin{aligned}&\mathit{iProp} \triangleq \mathit{UPred}(\mathcal{M})\\ &\quad \mathcal{M} \triangleq \mathit{Map}(\mathbb{P}(\mathbb{N}), \uplus) \times \mathit{Map}(\mathbb{P}^{\text{fin}}(\mathbb{N}), \uplus) \times \mathit{Map}(\mathit{Auth}(\mathbb{N} \xrightarrow{\text{fin}} \mathit{Ag}(\mathit{iProp}))) \times \cdots\end{aligned}$$

Through $\mathcal{M}$, the type *iProp* refers to itself. And what is even worse, this form of a cycle neither has an inductive nor a coinductive solution, because $\mathit{UPred}(M)$ uses M in a *negative occurrence*. In fact, strongly simplified, $\mathit{UPred}(M)$ consists of *sets of elements of* M, which has a strictly larger cardinality than M—and *iProp* cannot have a larger cardinality than itself.

Step-indexed types and resources. So what now? How can we build a model of impredicative invariants nonetheless? The answer to these questions is *step-indexing*. That is, beyond using step-indexing in the definition of $\mathit{UPred}(M)$, Iris uses a form of step-indexing in its types and resources. It requires generalizing types to "*(complete) ordered families of equivalences*", functions to "*non-expansive functions*", and resource algebras to "*cameras*".[22] The rough idea is that equality $x = y$, validity $\mathcal{V}$, and other definitions of the model become parametric in a step-index (*i.e.*, $x \overset{n}{=} y$, $(n, x) \in \mathcal{V}, \ldots$).

[22] Jung et al., "Higher-order ghost state", 2016 [Jun+16].

Once the generalization to step-indexed types and resources is completed, one can introduce a new type former $\blacktriangleright A$, which roughly does the same as the later modality $\triangleright P$ on *iProp*. And just like there are guarded fixpoints $\mu\, f x.\, P\, f\, x$ of propositions where recursive occurrences are guarded by a later $\triangleright P$, one can show that there are guarded fixpoints of those types whose recursive occurrences are guarded by a type-level later $\blacktriangleright A$. For Iris, the resulting equation that can be solved in the step-indexed world is:

$$\begin{aligned}&\mathit{iProp} \triangleq \mathit{UPred}(\mathcal{M})\\ &\quad \mathcal{M} \triangleq \mathit{Map}(\mathbb{P}(\mathbb{N}), \uplus) \times \mathit{Map}(\mathbb{P}^{\text{fin}}(\mathbb{N}), \uplus) \times \mathit{Map}(\mathit{Auth}(\mathbb{N} \xrightarrow{\text{fin}} \mathit{Ag}(\blacktriangleright \mathit{iProp}))) \times \cdots\end{aligned}$$

Invariants and laters. The price one has to pay for the step-indexed types is that later modalities arise when working with invariants. That is, recall from the rule wp-inv-open (from §3.4.1) that when we open an invariant, we do not immediately get access to its contents R. Instead, we get access to R underneath a guarding later modality (*i.e.,* we get access to $\triangleright R$). This later is an artifact of the use of step-indexed types. More specifically, it is a direct consequence of the later modality $\triangleright P$ in the definition of wsat, and it is needed to make the step-indexed definition with $\blacktriangleright iProp$ work.[23] For a more extensive discussion of step-indexed types and the full model of Iris, we refer the reader to Jung et al.[24]

[23] At first glance, once might think that $\triangleright P$ is simply of type $\blacktriangleright iProp$, but this is not the case. Instead, the later arises, because in the proof of INV-OPEN-WSAT, one obtains a step-indexed equality between the P from $\boxed{P}^{\iota}$ and the map entry $I(\iota)$—and the equality only holds for step-index $n - 1$ while the current step-index is n.

[24] Jung et al., "Iris from the ground up: A modular foundation for higher-order concurrent separation logic", 2018 [Jun+18b].

Outlook. With this primer on both Iris and its model, we are now well prepared for the remainder of this dissertation. In Part II, we will change the step-indexing underlying Iris to use ordinals instead of natural numbers. In Part III, we will develop a new resource (using the resource constructions from §4.2) that can be spent to eliminate later modalities from the context. In Part IV, we will develop a specification inference technique centered around the weakest precondition. And in Part V, we will extend the notion of resources to support unstable resources, which break the central frame rule of Iris.

Part II

Transfinite Iris

Chapter 5

Introduction

All prior *step-indexed* separation logics[1]—separation logics like Iris which incorporate step-indexing into their model—suffer from a shared Achilles heel: they can prove *safety* properties ("bad things never happen"), but not *liveness* properties ("good things eventually happen"). There is a simple and intuitive explanation for this limitation: the whole idea of step-indexing is to give semantics to a program based only on its finitary behavior (*i.e.,* the finite prefixes of its traces), and safety properties are precisely those properties of a program that can be determined from examining its finitary behavior. In contrast, determining whether a program satisfies a liveness property fundamentally requires examination of its infinite traces.

[1] Jung et al., "Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning", 2015 [Jun+15]; Svendsen and Birkedal, "Impredicative concurrent abstract predicates", 2014 [SB14]; Svendsen, Birkedal, and Parkinson, "Modular reasoning about separation of concurrent data structures", 2013 [SBP13]; Appel, "Verified Software Toolchain", 2012 [App12].

Nevertheless, in this part of the dissertation, we show that it *is* in fact possible to equip step-indexed separation logics with support for liveness reasoning. Specifically, we will show how to transform Iris into a new logic **Transfinite Iris** that (unlike Iris) supports the verification of two essential liveness properties—*termination* and *termination-preserving refinement*—in the presence of higher-order state. But to do so, we first need to revisit the most basic foundations of step-indexed separation logics, because it turns out that the root of the problem concerns the very notion of what a "step-index" is. We will switch from finite step-indexing (with natural numbers as step-indices; see §4.3) to transfinite step-indexing (with ordinals as step-indices; see §9.2). But before we get there, let us begin with a concrete example to illustrate the kind of properties that we are interested in proving.

A motivating example. Consider the following example, a combinator for *recursive memoization* written in OCaml[2]:

[2] For this example, we use OCaml syntax. In principle, any higher-order stateful language would suffice, including Java, Python, and JavaScript.

```
let memo_rec t =
  let m = Hashtbl.create 0 in
  let rec g x =
    match Hashtbl.find_opt m x with
    | Some y -> y
    | None   -> let y = t g x in
                Hashtbl.add m x y; y
  in g
```

Recursive memoization is an optimization technique for recursive functions: results of recursive calls are cached and then retrieved whenever those calls are executed again. To memoize a recursive function `f`: σ `->` τ, the combinator

`memo_rec: ((`σ `->` τ`) -> (`σ `->` τ`)) -> (`σ `->` τ`)`

is applied to a *template* `t: (`σ `->` τ`) -> (`σ `->` τ`)` of the function `f`. In addition to the argument of type σ, the template takes a function of type σ `->` τ as an argument, which it can use for making recursive calls. For example, the recursive function `let rec f x = e` (where `f` and `x` can occur free in `e`) is memoized by:

```
let f_memo = let t = (fun f x -> e) in memo_rec t
```

The implementation of `memo_rec` uses a map `m` (here a hash map of initial size 0) to store results. The resulting function `f_memo` behaves like `f`, except for one key difference: it retrieves entries from the map `m` if they have been computed previously and stores results in `m` after it has computed them.

Now, consider what is required to *verify* `memo_rec`. As a combinator, `memo_rec` is written in a generic fashion (*i.e.,* parametric over the template `t`) and does not impose many restrictions on the template `t`, and hence the original function `f`.[3] For example, if the original function `f` diverges on argument `x`, so will the memoized version `f_memo`. Moreover, the original function `f` does not need to be verified itself, or even have a known specification for `memo_rec` to be of use. In short, the correct behavior of `f_memo` is *relative* to the possible behavior of `f`. One way to establish this formally is by showing the memoized function `f_memo` to be a *refinement* of the original function `f`, meaning that the behaviors exhibited by `f_memo` are *contained within* those exhibited by `f`.

[3] The main requirement, as we will see in §8.1, is that executions of `f` are deterministic (or "repeatable") in the sense that if `f` were to be executed again with the same argument, then it would return the same result.

On the one hand, due to the presence of higher-order state (the type τ of values stored in the hash table is arbitrary), step-indexed separation logics are one of the only tools available for proving this type of refinement. On the other hand, there is an important caveat: the refinement these logics support merely establishes that *if* `f_memo v` *terminates* with a result `r`, then `f v` terminates with a related result. We call this a *result refinement.* This result refinement says nothing, however, about what happens if `f_memo v` *diverges* (*i.e.,* does not terminate). For example, if we were to replace `t g x` with `g x` in Line 6 of the definition of `memo_rec`, then the resulting function returned by `memo_rec` would still be a refinement of `f` according to the theorem provable in prior step-indexed logics such as Iris—yet it would diverge on every input!

What we would really like to prove is a stronger theorem, stating additionally that if `f_memo v` has a non-terminating execution, then so does `f v`—or equivalently, if `f v` always terminates, then so does `f_memo v`. In this case, we say that `f_memo` is a *termination-preserving refinement* of `f`. Proving this refinement, however, requires examining infinite traces of `f_memo`'s behavior—in other words, *liveness* reasoning.

There has been some prior work on *approximating* liveness reasoning within step-indexed logics. In particular, Dockins and Hobor[4] and Mével, Jourdan, and Pottier[5] have shown how to prove termination if the user gives explicit time complexity bounds. Tassarotti, Jung, and Harper[6] have shown how to prove termination-preserving refinement under some restrictions: the original (source) program must only exhibit bounded nondeterminism, and internally, the refinement proof can only rely on bounded stuttering (with the bound for stuttering chosen up front).

[4] Dockins and Hobor, "A theory of termination via indirection", 2010 [DH10]; Dockins and Hobor, "Time bounds for general function pointers", 2012 [DH12].

[5] Mével, Jourdan, and Pottier, "Time credits and time receipts in Iris", 2019 [MJP19].

[6] Tassarotti, Jung, and Harper, "A higher-order logic for concurrent termination-preserving refinement", 2017 [TJH17].

In all of the above, however, the restrictions effectively serve to turn the property being proven from a liveness property into a safety property. For example, although "e terminates" is a liveness property, "e terminates in n steps" (where n is an explicit user-specified bound) is a safety property, since its validity can be determined after examining only the first n steps of e's execution. Moreover, the restrictions of Tassarotti, Jung, and Harper's approach render it insufficient to prove termination-preserving refinement for an example like `memo_rec`, where the number of steps required for the hash table lookup in `memo_rec` is unbounded (since the size of the hash table is not statically bounded).

Transfinite Iris and the existential property. In the following, we show how step-indexed separation logics—in particular Iris—can be transformed to support *true* liveness reasoning, albeit with a fundamental change to how they are modeled. Our first step is to identify the key property that prior step-indexed separation logics fail to satisfy, but that would enable liveness reasoning if it held. Concretely, we observe that what these logics are missing is the *existential property*:

$$\text{if } \vdash \exists x : X.\, \Phi\, x, \text{ then } \vdash \Phi\, x \text{ for some } x : X.$$

where $\vdash P$ means P is true in the step-indexed logic (*i.e.,* for Iris $\vdash P \triangleq \mathsf{True} \vdash P$ for the entailment $P \vdash Q$ from §4.3). The existential property ensures that existential quantification *inside the step-indexed logic* corresponds to existential quantification *outside the step-indexed logic* (*i.e.,* at the meta level).

Intuitively, the existential property is useful for liveness reasoning because, when we do liveness reasoning inside a step-indexed logic, we often need to prove propositions that are existentially quantified. For example, when we prove a termination-preserving refinement, we will end up needing to show that for all steps of execution in the target program (*e.g.,* `memo_rec t`), there *exist* some corresponding steps in the source program (*e.g.,* the original recursive function `f`). The existential choice of source steps is made *inside* the proof in the step-indexed logic, but ultimately in order to establish the termination-preserving refinement, we need to be able to hoist that existential choice out to the meta level. The existential property enables us to do just that.

Unfortunately, the existential property is fundamentally incompatible with how *step-indexed* logics have thus far been modeled. Step-indexed logics traditionally model propositions as predicates over a *natural number* n as the step-index similar to the model of Iris that we have discussed in §4.3. In this standard model, the existential property is demonstrably false (§6.6).

To validate the existential property and thereby enable liveness reasoning, we thus propose to change the underlying notion of the step-index from a finite one to a *transfinite* one: from natural numbers to *ordinals*. We will do so below for Iris, resulting in a new logic that we call **Transfinite Iris**. We use Transfinite Iris to establish two key liveness properties—*termination* and *termination-preserving refinement*—and apply it to a range of interesting examples, including the `memo_rec` example from above.

In order to get to the bottom of our core "existential dilemma" (*i.e.,* how the existential property can enable liveness reasoning for step-indexed logics, and how to provide a semantic foundation for it), we focus our attention on one of the primary *raisons d'être* of step-indexed separation logics: reasoning about *sequential, higher-order stateful programs.* Of course, one of the original motivations for Iris was *safety reasoning about concurrent programs*, and Transfinite Iris inherits that support (see §7). But we leave step-indexed *liveness reasoning about concurrent programs* as an important direction for future work.

Contributions. Our main contribution in this part of the dissertation is Transfinite Iris, a transfinitely step-indexed version of Iris. While the idea of transfinite step-indexing is not new (see §10 for a discussion of related work), Transfinite Iris is to our knowledge the first step-indexed program logic that satisfies the existential property, thereby truly supporting proving liveness properties directly inside the program logic.

As part of developing Transfinite Iris, we make the following contributions:

- We identify the *existential property* as the key property missing from prior step-indexed separation logics to support liveness reasoning, and we show that it is fundamentally inconsistent with standard step-indexed models (§6).
- As the foundation of Transfinite Iris, we develop a transfinitely step-indexed model of Iris (§9) that enjoys the existential property. As part of this model, we solve a new and challenging recursive domain equation for modeling Transfinite Iris's propositions. We describe the construction at a high level in §9.3. Further details can be found in the appendix of Transfinite Iris.[7]
- On top of the transfinitely step-indexed model, we develop two program logics for two liveness properties: one for termination-preserving refinement (§7.1) and one for termination (§7.2).
- We demonstrate the power of Transfinite Iris on a range of interesting examples, including the `memo_rec` example presented above (§8).

Transfinite Iris and all examples in §8 are fully mechanized in Rocq using the Iris Proof Mode.[8] See the Transfinite Iris Rocq development[9] for the Rocq proofs.

[7] Spies et al., *Transfinite Iris appendix and Rocq development*, 2021 [Spi+21a].

[8] Krebbers, Timany, and Birkedal, "Interactive proofs in higher-order concurrent separation logic", 2017 [KTB17]; Krebbers et al., "MoSeL: A general, extensible modal framework for interactive proofs in separation logic", 2018 [Kre+18].

[9] Spies et al., *Transfinite Iris appendix and Rocq development*, 2021 [Spi+21a].

Chapter 6

The Existential Property

Let us start by explaining how the *existential property* is key to enabling step-indexed logics to prove termination and termination-preserving refinement. We set the resources of separation logic aside in this chapter and focus only on step-indexing. We first define refinements (§6.1) and explain how they are proven using simulations (§6.2). We then show why step-indexing is useful for defining such simulations but falls short for termination-preserving refinements (§6.3). Next, we discuss how a step-indexed logic with the existential property enables proofs of termination-preserving refinements (§6.4) and termination (§6.5) and, finally, we describe how the existential property is justified by transfinite step-indexing (§6.6).

6.1 Refinements

Intuitively, a refinement between a target program t and a source program s expresses that all observable behaviors of the target t are also valid behaviors of the source s. To make this notion precise (and to distill the difference from prior work), we fix an abstract and simplified setting. We assume a source language S and a target language T. We assume programs in both languages are expressions, equipped with a small-step operational semantics. We write $s \rightsquigarrow_{\text{src}} s'$ for a step of the source language and $t \rightsquigarrow_{\text{tgt}} t'$ for a step of the target language. We assume (for simplicity) that the only values in both languages are Booleans, denoted by b.

To clarify what a refinement is in this abstract setting, we have to specify what the "observable behaviors" of a program should be. In the simplest case, the only observable behavior is the result of a program. In this case, the corresponding refinement between target t and source s is given by:

for all b, if t evaluates to b, then s evaluates to b.

where"evaluates to b" means there exists an execution ending in the Boolean b.[1] We dub this refinement a *result refinement* and write $t \sqsubseteq_{\text{res}} s$.

[1] We do not require reduction to be deterministic, so for one starting state t there may be multiple possible executions (and correspondingly also multiple return values).

For a termination-preserving refinement, we additionally consider divergence (*i.e.,* non-termination) as an observable behavior. Formally, a *termination-preserving refinement* between t and s, written $t \sqsubseteq_{\text{term}} s$, is given by:

(1) for all b, if t evaluates to b, then s evaluates to b, and

(2) if t diverges, then s diverges.

Here, "diverges" means there exists a divergent execution. This refinement, as the name suggests, preserves termination[2] from the source s to the target t.

[2] That is, "if s terminates on all execution paths, then t terminates on all execution paths" is (classically) equivalent to "if t diverges, then s diverges".

6.2 Proving Refinements using Simulations

A well-known technique to prove refinements is to (1) give a small-step simulation ($\preceq$) between target and source expressions, and (2) show that the simulation is *adequate*, *i.e.,* for every target and source expressions t and s, the simulation $t \preceq s$ implies the desired refinement between t and s.

For example, we can prove a termination-preserving refinement $t \sqsubseteq_{\text{term}} s$ by establishing that s simulates t in lock-step, as captured by the following coinductively-defined relation:

$$t \preceq s \triangleq_{\text{coind}} (\exists (b : \mathbb{B}).\, t = s = b) \lor \begin{pmatrix} (\exists t'.\, t \rightsquigarrow_{\text{tgt}} t') \land \\ \forall t'.\, t \rightsquigarrow_{\text{tgt}} t' \Rightarrow \exists s'.\, s \rightsquigarrow_{\text{src}} s' \land t' \preceq s' \end{pmatrix}$$

In the definition of $t \preceq s$, either both sides have reached the same result b, or the target t can take some step (to avoid cases where the target is a Boolean different from the source), and every step of the target ($t \rightsquigarrow_{\text{tgt}} t'$) can be replayed in the source ($s \rightsquigarrow_{\text{src}} s'$) such that the resulting expressions t' and s' are again in the simulation.

6.3 Step-Indexed Simulations

While this coinductively-defined simulation relation suffices to prove refinements of first-order programs, it often falls short when considering programming languages with “cyclic” features like recursive types and higher-order state.[3] For such languages, step-indexing has proved to be a very fruitful technique, as shown by the abundance of work on step-indexed techniques for proving *result refinements.*[4]

We can define a step-indexed simulation in Iris—and more generally in a step-indexed logic with the *later modality* $\triangleright P$ (see §3.2)—as follows:

$$t \preceq_{\triangleright} s \triangleq (\exists (b : \mathbb{B}).\, t = s = b) \lor \begin{pmatrix} (\exists t'.\, t \rightsquigarrow_{\text{tgt}} t') \land \\ \forall t'.\, t \rightsquigarrow_{\text{tgt}} t' \Rightarrow \exists s'.\, s \rightsquigarrow_{\text{src}} s' \land \triangleright (t' \preceq_{\triangleright} s') \end{pmatrix}$$

Superficially, this simulation relation is very close to the simulation relation $t \preceq s$. But there are two key differences: First, this simulation relation is defined *inside* the step-indexed logic (*e.g.,* as an *iProp*, similar to the weakest precondition in §4.1). Second, the recursive occurrence is guarded by a later modality $\triangleright P$, making the definition step-indexed, and allowing us to use a guarded fixpoint (see $\mu\, f x.\, P\, f\, x$ in §4.3) to define it.

The problem with step-indexed simulations. As it turns out, these two differences have a profound impact. While the normal simulation relation $t \preceq s$ is adequate for *termination preserving refinements* $t \sqsubseteq_{\text{term}} s$, the step-indexed one is not—it is only adequate for *result refinements* $t \sqsubseteq_{\text{res}} s$. To see why, we should take a look at what happens inside the step-indexed model. In the model, the simulation is stratified into a family of approximations ($\preceq_i$), indexed by a

[3] In principle, one can also use standard coinductively-defined simulation relations for such languages as, *e.g.,* Gäher et al. [Gäh+22] demonstrate. However, step-indexed simulations are the de-facto standard for these languages. The reason is that via, *e.g.,* impredicative invariants, they allow one to handle challenging recursive reasoning that is often necessary for more advanced examples with shared, mutable, higher-order state.

[4] Ahmed, Dreyer, and Rossberg, “State-dependent representation independence”, 2009 [ADR09]; Dreyer, Ahmed, and Birkedal, “Logical step-indexed logical relations”, 2011 [DAB11]; Dreyer et al., “A relational modal logic for higher-order stateful ADTs”, 2010 [Dre+10]; Turon et al., “Logical relations for fine-grained concurrency”, 2013 [Tur+13]; Turon, Dreyer, and Birkedal, “Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency”, 2013 [TDB13]; Krebbers, Timany, and Birkedal, “Interactive proofs in higher-order concurrent separation logic”, 2017 [KTB17]; Krogh-Jespersen, Svendsen, and Birkedal, “A relational model of types-and-effects in higher-order concurrent separation logic”, 2017 [KSB17]; Timany et al., “A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of runST”, 2018 [Tim+18]; Frumin, Krebbers, and Birkedal, “ReLoC: A mechanised relational logic for fine-grained concurrency”, 2018 [FKB18].

step-index i:[5]

$$\begin{aligned} t \preceq_0 s &\triangleq \mathsf{True} \\ t \preceq_{i+1} s &\triangleq (\exists (b : \mathbb{B}).\, t = s = b) \lor \\ &\quad \begin{pmatrix} (\exists t'.\, t \rightsquigarrow_{\text{tgt}} t') \land \\ \forall t'.\, t \rightsquigarrow_{\text{tgt}} t' \Rightarrow \exists s'.\, s \rightsquigarrow_{\text{src}} s' \land t' \preceq_i s' \end{pmatrix} \end{aligned}$$

[5] Technically, this relation is "off by one". The base case of $(\preceq_{\triangleright})$ in the Iris model is not True but what is presented here as $t \preceq_1 s$. If we consider the limit $\forall i.\, t \preceq_i s$, then the difference vanishes, *i.e.*, $(\vdash t \preceq_{\triangleright} s) \Leftrightarrow \forall i.\, t \preceq_i s$.

The above definition is structurally recursive on the natural number i. The simulation relation $t \preceq_{\triangleright} s$ is equivalent to the limit of the approximations, *i.e.*, $(\vdash t \preceq_{\triangleright} s) \Leftrightarrow \forall i.\, t \preceq_i s$.

For this step-indexed simulation, we can prove a result refinement, which is what is usually done for step-indexed simulation relations:

Lemma 39. *If* $\vdash t \preceq_{\triangleright} s$, *then* $t \sqsubseteq_{res} s$.

Proof Sketch. Let $t = t_0 \rightsquigarrow_{\text{tgt}} \cdots \rightsquigarrow_{\text{tgt}} t_n = b$ be the execution of t to b. Recall that $\vdash t \preceq_{\triangleright} s$ means $\forall i.\, t \preceq_i s$. We pick $i \triangleq n + 1$ and extract an execution $s = s_0 \rightsquigarrow_{\text{src}} \cdots \rightsquigarrow_{\text{src}} s_n$ such that $t_n \preceq_1 s_n$ from $t \preceq_{n+1} s$ by unrolling the definition of $(\preceq_i)$ n times. Since $t_n = b$, the expression t_n can no longer take steps. Consequently, in the definition of $t_n \preceq_1 s_n$, only the first clause can be true. We obtain $s_n = b$. ◻

Unfortunately, unlike the coinductively-defined simulation relation $t \preceq s$, the step-indexed simulation relation $t \preceq_{\triangleright} s$ is (in general)[6] *not* adequate for termination-preserving refinements. Let us try to prove that it implies a termination-preserving refinement:

[6] If one restricts the source language enough (*e.g.*, only bounded non-determinism and only stuttering up to a fixed bound; see §7.1.3), then one can still use such a relation for termination-preserving refinements. This is what Tassarotti, Jung, and Harper [TJH17] do. See §10 for a comparison.

$$\text{if} \vdash t \preceq_{\triangleright} s \text{ and } t \text{ diverges, then } s \text{ diverges.}$$

and see where the argument goes wrong. If we attempt a proof similar to the proof of Lemma 39, we are stuck when we try to determine a sufficient value of the step-index i. We have seen that for any natural number i, we can extract a finite trace of the source (of length i) from $t \preceq_i s$. However, which execution we obtain this way can depend on the step-index i, meaning for each step-index i there could be a different finite execution of the source.

For example, consider the case where the target t_∞ is an infinite loop, and the source $s_{<\infty}$ non-deterministically picks a natural number n, and then executes n steps before terminating. For every i, we can find a trace of i steps where $s_{<\infty}$ simulates t_∞, but there is no divergent execution of $s_{<\infty}$. Thus, we cannot extract one coherent, infinite execution of $s_{<\infty}$ from $t_\infty \preceq_{\triangleright} s_{<\infty}$.

6.4 The Existential Property

As a consequence, one cannot prove that the simulation relation $\preceq_{\triangleright}$ implies a termination-preserving refinement in step-indexed logics like Iris. If, however, instead of thinking of step-indexed propositions as predicates over natural numbers, we adopt a higher-level perspective (*i.e.*, we forget the model from §4), then we may rightly wonder: what property are step-indexed logics missing that prohibits us from proving termination-preserving refinements? The answer to this question is the *existential property*:

$$\text{if } \vdash \exists x : X.\, \Phi\, x, \text{ then } \vdash \Phi\, x \text{ for some } x : X.$$

If we imagine we were working in a step-indexed logic that enjoyed the existential property, then we could, in fact, prove:

Lemma 40. *If* $\vdash t \preceq_{\triangleright} s$ *and* t *diverges, then* s *diverges.*

Proof Sketch. Let $t = t_0 \rightsquigarrow_{\text{tgt}} t_1 \rightsquigarrow_{\text{tgt}} \cdots$ be an infinite target execution. By coinduction, we will construct an infinite source execution $s = s_0 \rightsquigarrow_{\text{src}} s_1 \rightsquigarrow_{\text{src}} \cdots$. Initially, we know $\vdash t \preceq_{\triangleright} s$. With $t = t_0 \rightsquigarrow_{\text{tgt}} t_1$, we can use $\vdash t \preceq_{\triangleright} s$ to obtain $\vdash \exists s'.\, s \rightsquigarrow_{\text{src}} s' \wedge \triangleright(t_1 \preceq_{\triangleright} s')$. With the existential property, we obtain an s_1 such that $s \rightsquigarrow_{\text{src}} s_1$ and $\vdash \triangleright(t_1 \preceq_{\triangleright} s_1)$. Step-indexed logics like Iris enjoy the rule $\vdash \triangleright P$ implies $\vdash P$, allowing us to strip off the later modality. Thus, we obtain $\vdash t_1 \preceq_{\triangleright} s_1$. We can then proceed in a similar manner to obtain $s_2, s_3, \ldots$ by coinduction for $\vdash t_1 \preceq_{\triangleright} s_1$. □

6.5 Termination

Once we have a version of the simulation ($\preceq_{\triangleright}$) that is adequate for proving termination-preserving refinements, we can repurpose it to prove another liveness property: *termination.* That is, observe that the source language in our simulation does not have to be a programming language—it merely has to be a transition system. If we instantiate it with a relation that always terminates (*i.e.,* the inverse of a well-founded relation), then termination-preserving refinement ensures termination of the target:

Lemma 41. *If* $\vdash t \preceq_{\triangleright} s$ *for some* s*, and the source relation* ($\rightsquigarrow_{src}$) *is the inverse of a well-founded relation, then* t *terminates along all execution paths.*

Proof Sketch. By way of contradiction, assume there is an infinite execution $t = t_0 \rightsquigarrow_{\text{tgt}} t_1 \rightsquigarrow_{\text{tgt}} \cdots$. We obtain an infinite execution $s = s_0 \rightsquigarrow_{\text{src}} s_1 \rightsquigarrow_{\text{src}} \cdots$ analogous to Lemma 40. This infinitely descending chain is a contradiction to the assumption that ($\rightsquigarrow_{\text{src}}$) is the inverse of a well-founded relation. □

In particular, we can obtain a proof technique for termination by choosing ordinals as the source language with $\alpha \rightsquigarrow_{\text{src}} \beta \triangleq \beta < \alpha$, since they cannot be decreased infinitely often. (We could also choose natural numbers as the source language, but then we would be back to proving bounded termination.)[7] As we will see in §7.2, ordinals will allow us to abstract over dynamic information revealed during the execution of the program, which makes termination proofs more compositional.

[7] Recall from §5 that bounded termination is a safety property. Its main downside is that it requires one to determine explicit finite bounds.

6.6 Justifying the Existential Property

Sadly, the existential property does not hold in step-indexed logics like Iris—for the same reason that, in Iris, $\vdash t \preceq_{\triangleright} s$ does not imply $t \sqsubseteq_{\text{term}} s$: the witnesses of existential quantification *could depend on the step-index* i. For example, consider the proposition $\vdash \exists n : \mathbb{N}.\, \triangleright^n \mathsf{False}$. Intuitively, it means that eventually the step-index runs out. It is *provable* in Iris, because if we drop down to the step-indexed model (see §4.3), then we see that it means

for every step-index i*, there is a natural number* n *such that* $n > i$

This is trivially true by picking $n \triangleq i + 1$.

So how can we transform Iris so that it will enjoy the existential property? The fundamental modification that we make is to move from *finite* step-indexing with natural numbers to *transfinite* step-indexing with ordinals.

To explain how transfinite step-indexing validates the existential property, we consider what went wrong in the case of step-indexing with natural numbers. Both in the above example $\vdash \exists n : \mathbb{N}.\ \triangleright^n \mathsf{False}$ and the simulation $\vdash t_\infty \preceq_\triangleright s_{<\infty}$ from §6.3, the problem was that there are witnesses that simply "outlast" the current step-index i. For $\vdash \exists n : \mathbb{N}.\ \triangleright^n \mathsf{False}$, given any step-index i, we could always pick a witness n greater than i; for $\vdash t_\infty \preceq_\triangleright s_{<\infty}$, given any step-index i, we could always pick an execution of $s_{<\infty}$ that takes longer than i steps to terminate. However, if we use ordinals as step-indices, then in both examples this is no longer possible. For example, there is no $n : \mathbb{N}$ that is larger than ω, which is (by definition) the first ordinal larger than all natural numbers. As a result, $\vdash \exists n : \mathbb{N}.\ \triangleright^n \mathsf{False}$ is no longer provable in a model with ω as a step-index, a *transfinitely* step-indexed model.

We will formally show in §9 that a model with sufficiently large ordinals *does* validate the existential property—thus enabling liveness reasoning. More specifically, we will not completely eliminate the dependency of the existential witness on the current step-index (see the definition of $\exists x : X.\, P\, x$ in §9.1). Instead, we will develop a cardinality argument between the type of step-indices and the witness type X for when the witness $x : X$ does, in fact, *not* depend on the step-index. Concretely, the existential property

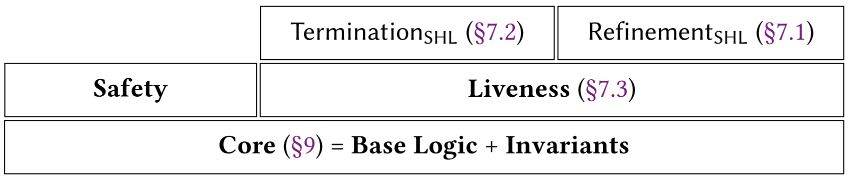
$$\text{if } \vdash \exists x : X.\, \Phi\, x, \text{ then } \vdash \Phi\, x \text{ for some } x : X.$$

holds, when the quantified type X is "substantially smaller"[8] than the type of the underlying step-indices. For example, if one restricts X to only *finite* types, then the existential property already holds for step-indexing with natural numbers (so in regular Iris). If the type X is countable, then the existential property holds for uncountable ordinals as step-indices. In general, for every type X, we will construct ordinals in §9 such that Transfinite Iris step-indexed with these ordinals enjoys the existential property for X. At the same time, if one fixes a particular choice of ordinals for the model, then there will also be choices for the type of witnesses X that do not enjoy the existential property (*e.g.,* the type of step-indicies or the type of Iris propositions *iProp*).

[8] The formal requirement, as we will see in §9.1, is that the step-index type contains suprema of families of step-indices indexed by the type X. In practice, this means we need to be able to place the type X in Rocq in a universe below the universe of the type of step-indices.

Chapter 7

The Program Logics of Transfinite Iris



Termination$_{\mathsf{SHL}}$ (§7.2)
Refinement$_{\mathsf{SHL}}$ (§7.1)
Safety
Liveness (§7.3)
Core (§9) = Base Logic + Invariants

Figure 7.1: Roadmap of Transfinite Iris

We now put the key ideas from §6 into action by introducing **Transfinite Iris**—a step-indexed separation logic framework capable of proving safety, termination, and termination-preserving refinements of higher-order stateful programs. We start with a tour of the components of the framework (depicted in Fig. 7.1). Along the way, we highlight which aspects of Transfinite Iris are original and which are inherited from Iris.

We start with the **core logic** of Transfinite Iris. As in Iris (see §4), the core of Transfinite Iris is a step-indexed logic of bunched implications[1] with resources and invariants. It has all the connectives of Iris's base logic (see §4.3) and, additionally, Iris's impredicative invariants[2] (see §3.4.1). It includes the typical connectives of step-indexed logic (*e.g.,* the later modality) and separation logic (*e.g.,* separating conjunction). It does not, however, have exactly the same rules and properties as Iris's base logic: the core logic of Transfinite Iris enjoys the existential property, which Iris does not. In exchange, it loses two of Iris's commuting rules which are in conflict with the existential property (see §9.2).

[1] O'Hearn and Pym, "The logic of bunched implications", 1999 [OP99].

[2] Jung et al., "Higher-order ghost state", 2016 [Jun+16]; Svendsen and Birkedal, "Impredicative concurrent abstract predicates", 2014 [SB14].

While, on the surface, the core logic differs only marginally from Iris, it differs substantially on the inside! In Transfinite Iris, to validate the existential property, we must change the step-indexed model used to define the base logic. This change constitutes the titular difference from Iris: in the model, we use *transfinite* step-indexing (*i.e.,* with ordinals), whereas Iris's model is based on finite step-indexing (*i.e.,* with natural numbers; see §4.3). We postpone further discussion of the details of this change until §9.

To enable program verification, we extend the core logic with two program logics: one for proving *safety properties* and one for proving *liveness properties*. The **safety logic** is inherited from Iris with small modifications to the program logic introduced in §3. (As discussed in §9.2, we need to account for the loss of the two commuting rules in the core logic.) Since it is derived from the regular Iris safety logic (discussed in §3), we focus on the liveness logic in the following.

The **liveness logic** (§7.3) is a new contribution of Transfinite Iris. It is a generic program logic with constructs for proving termination preservation from source to target. Below, we consider two instantiations with "Sequential

HeapLang" (SHL), the sequential fragment of HeapLang (*i.e.,* the fragment of HeapLang that we have discussed prior to §3.5; see §2.1 and §2.2). We develop Refinement$_{\mathsf{SHL}}$ (in §7.1), a program logic for proving termination-preserving refinements between programs in SHL, and we develop Termination$_{\mathsf{SHL}}$ (in §7.2), a program logic for proving termination of programs in SHL.

7.1 Termination-Preserving Refinement

To introduce Refinement$_{\mathsf{SHL}}$, we proceed in three steps: First, we review the canonical approach for internalizing result refinements in separation logic pioneered by Turon, Dreyer, and Birkedal[3] (§7.1.1). Then, we explain how Refinement$_{\mathsf{SHL}}$ goes beyond this approach to handle termination-preserving refinements. We start with a version which does not yet support stuttering (§7.1.2) and then add rules for stuttering (§7.1.3).

[3] Turon, Dreyer, and Birkedal, "Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency", 2013 [TDB13].

7.1.1 Result Refinements in Iris

Following the "Iris approach" of modularly building up complex reasoning principles from simple abstractions, we do not bake in a simulation relation such as ($\preceq_{\triangleright}$) from §6.3 as a primitive. Instead, we define it in terms of simpler connectives—Hoare triples and separation-logic resources—following the approach of Turon, Dreyer, and Birkedal.[4] Below, we recall their approach for proving result refinements in the context of vanilla Iris and then explain how we adapt the approach to prove termination-preserving refinements in §7.1.2.

[4] Turon, Dreyer, and Birkedal, "Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency", 2013 [TDB13].

The central connectives of the approach of Turon, Dreyer, and Birkedal are:

$$P, Q ::= \cdots \mid \ell \mapsto v \mid \{P\}\ e\ \{v.\ Q(v)\}_{\mathsf{rr}} \mid \ell \mapsto_{\mathsf{src}} v \mid \mathsf{src}(e)$$

The Hoare triple $\{P\}\ e\ \{v.\ Q(v)\}_{\mathsf{rr}}$ is used to reason about the target, and the resource assertion $\mathsf{src}(e)$ is used to reason about the source.[5] (We mark the Hoare triple with "$_{\mathsf{rr}}$" for "result refinement", because we will introduce different Hoare triples for termination-preserving refinement in §7.1.2.) The points-to assertion $\ell \mapsto v$ is used to reason about the target memory, and the points-to assertion $\ell \mapsto_{\mathsf{src}} v$ is used to reason about the source memory. The expressions e in the source and the target are drawn from SHL in the following.

[5] Since they work in a concurrent setting, Turon, Dreyer, and Birkedal consider multiple threads in the source program. Here, we consider only a single one.

The analog of ($\preceq_{\triangleright}$) from §6, generalized to arbitrary ground types G—*e.g.,* unit ($\mathbb{1}$), Booleans ($\mathbb{B}$), natural numbers ($\mathbb{N}$)—is defined as:

$$e_t \preceq^{\mathsf{rr}}_G e_s \triangleq \forall K.\ \{\mathsf{src}(K[e_s])\}\ e_t\ \{v.\ \mathsf{src}(K[v]) * v \in G\}_{\mathsf{rr}}$$

Recall that when proving a result refinement between e_t and e_s, we need to show that for every execution of e_t there is a corresponding execution of e_s. In the approach of Turon, Dreyer, and Birkedal, the Hoare triple $\{P\}\ e_t\ \{v.\ Q\}_{\mathsf{rr}}$ is used to reason about *every* execution of e_t, and the separation logic resource $\mathsf{src}(e_s)$ to reason about the existence of *some* execution of e_s. Intuitively, $\mathsf{src}(e_s)$ says that the program on the right of the refinement is currently e_s. The proof rules allow one to transform $\mathsf{src}(e_s)$ into $\mathsf{src}(e_s')$ if and only if e_s reduces to e_s'.

Hence, $\{\mathsf{src}(e_s)\}\ e_t\ \{v.\ \mathsf{src}(v) * v \in G\}_{\mathsf{rr}}$ says that if the right-hand side initially is e_s, then for every execution of e_t that results in v, the right-hand side reduces to the same value v. Additionally, the definition of the refinement $e_t \preceq^{\mathsf{rr}}_G e_s$ quantifies over all evaluation contexts K to make the judgment compositional. (At the top level, one takes K to be the empty context.)

value-tgt-rr
$$\{P(v)\}\ v\ \{w.\ P(w)\}_{\mathrm{rr}}$$

bind-tgt-rr
$$\frac{\{P\}\ e_t\ \{v.\ Q(v)\}_{\mathrm{rr}} \qquad \forall v.\ \{Q(v)\}\ K[v]\ \{w.\ R(w)\}_{\mathrm{rr}}}{\{P\}\ K[e_t]\ \{w.\ R(w)\}_{\mathrm{rr}}}$$

pure-tgt-rr
$$\frac{\{P\}\ e_t'\ \{v.\ Q(v)\}_{\mathrm{rr}} \qquad e_t \rightarrow_{\mathrm{pure}} e_t'}{\{\triangleright P\}\ e_t\ \{v.\ Q(v)\}_{\mathrm{rr}}}$$

pure-src-rr
$$\frac{\left\{\mathsf{src}(K[e_s']) * P\right\}\ e_t\ \{v.\ Q(v)\}_{\mathrm{rr}} \qquad e_s \rightarrow_{\mathrm{pure}} e_s'}{\{\mathsf{src}(K[e_s]) * P\}\ e_t\ \{v.\ Q(v)\}_{\mathrm{rr}}}$$

store-tgt-rr
$$\{\ell \mapsto v * \triangleright P\}\ \ell := w\ \{\_.\ \ell \mapsto w * P\}_{\mathrm{rr}}$$

store-src-rr
$$\frac{\{\ell \mapsto_{\mathrm{src}} w * \mathsf{src}(K[()])\}\ e_t\ \{u.\ Q(u)\}_{\mathrm{rr}}}{\{\ell \mapsto_{\mathrm{src}} v * \mathsf{src}(K[\ell := w])\}\ e_t\ \{u.\ Q(u)\}_{\mathrm{rr}}}$$

load-tgt-rr
$$\{\ell \mapsto v * \triangleright P\}\ !\ell\ \{w.\ w = v * \ell \mapsto v * P\}_{\mathrm{rr}}$$

load-src-rr
$$\frac{\{\ell \mapsto_{\mathrm{src}} v * \mathsf{src}(K[v])\}\ e_t\ \{w.\ Q(w)\}_{\mathrm{rr}}}{\{\ell \mapsto_{\mathrm{src}} v * \mathsf{src}(K[!\ell])\}\ e_t\ \{w.\ Q(w)\}_{\mathrm{rr}}}$$

ref-tgt-rr
$$\{\triangleright P\}\ \mathsf{ref}(v)\ \{w.\ \exists \ell.\ w = \ell * \ell \mapsto v * P\}_{\mathrm{rr}}$$

ref-src-rr
$$\frac{\{\exists \ell.\ \ell \mapsto_{\mathrm{src}} v * \mathsf{src}(K[\ell])\}\ e_t\ \{w.\ Q(w)\}_{\mathrm{rr}}}{\{\mathsf{src}(K[\mathsf{ref}(v)])\}\ e_t\ \{w.\ Q(w)\}_{\mathrm{rr}}}$$

Figure 7.2: A selection of proof rules for result refinements.

Iris. The approach of Turon, Dreyer, and Birkedal was first brought to Iris by Krebbers, Timany, and Birkedal.[6] They use Iris's regular Hoare triples $\{P\}\ e_t\ \{v.\ Q(v)\}$ (from §3.3) to reason about the target, and they use the regular points-to $\ell \mapsto v$ to reason about the memory of the target. The source assertion $\mathsf{src}(e_s)$ and the source memory points-to $\ell \mapsto_{\mathrm{src}} v$ are encoded using resources (§4.2). The details of this construction (*e.g.,* how exactly these assertions are modeled) are not important in the following, since we will define different Hoare triples for termination-preserving refinements in §7.3.[7] We focus on the resulting program logic for proving refinements.

[6] Krebbers, Timany, and Birkedal, "Interactive proofs in higher-order concurrent separation logic", 2017 [KTB17].

[7] Krebbers, Timany, and Birkedal allocate a global invariant that internally maintains the state of the current source execution (*i.e.,* the current source program and heap). This invariant incrementally builds up the source execution. It reflects the current state of the source into the logic via resources, which underpin $\mathsf{src}(e_s)$ and $\ell \mapsto_{\mathrm{src}} v$. We will see more of this variant—albeit not how it is modeled—in Part III for defining a binary logical relation.

Proof rules. There are two kinds of rules: (1) standard Hoare triple rules for reasoning about the target e_t and (2) rules for interacting with the resource $\mathsf{src}(e_s)$ for simulating the source expression. A selection of rules is given in Fig. 7.2. (Of course, standard structural rules which are agnostic about source and target such as framing and the rule of consequence still apply.) The *target rules* are the standard rules from §2 for proving Hoare triples in SHL (see Fig. 2.1 and Fig. 2.3).[8] For example, we can use the rule pure-tgt-rr for pure steps, bind-tgt-rr for binding on a subexpression in an evaluation context K, and store-tgt-rr for updating a reference. As additional *source rules*, we have pure-src-rr for pure steps, store-src-rr for updating a location in the source, load-src-rr for reading a location in the source, and ref-src-rr for allocating a new reference in the source. Notably, there are no rules for binding and values on the source side. Instead, all other rules (*e.g.,* pure-src-rr) are phrased in a style where they apply in any evaluation context K.

[8] Here, we have strengthened pure-tgt-rr, store-tgt-rr, load-tgt-rr, and ref-tgt-rr compared to hoare-pure-step, hoare-store, hoare-load, and hoare-ref Fig. 2.1 and Fig. 2.3 by allowing them to strip the later modality associated with their step from the precondition, similar to the strengthened weakest precondition rules in §3.2.

Löb induction. None of the above rules mention anything about recursion or loops. The reason is that—as we have seen in §3.2—we can use Löb induction to reason about recursive programs in Iris. In particular, specialized to Hoare triples, one can derive the following variant:[9]

[9] To avoid confusing with function application, we have indicated the potential dependency on x with a superscript in this rule.

HOARE-LÖB
$$\frac{\forall x.\ \left\{P^x * \triangleright \left(\forall y.\ \{P^y\}\ e^y\ \{v.\ Q^y(v)\}_{\mathsf{rr}}\right)\right\}\ e^x\ \{v.\ Q^x(v)\}_{\mathsf{rr}}}{\forall x.\ \{P^x\}\ e^x\ \{v.\ Q^x(v)\}_{\mathsf{rr}}}$$

To prove a (universally quantified) Hoare triple, we can assume in the precondition that the Hoare triple already holds later.

Let us consider a simple example simulation using HOARE-LÖB (returning to the function first from Example 13):

Lemma 42. *If* $\forall n : \mathbb{N}.\ f\ n \preceq^{\mathsf{rr}}_{\mathbb{B}} g\ n$*, then* $\forall n : \mathbb{N}.\ \mathtt{first}\ f\ n \preceq^{\mathsf{rr}}_{\mathbb{N}} \mathtt{first}\ g\ n$*, where* $\mathtt{first} \triangleq \mathsf{fix}\,\mathtt{first}\ p\,x.\ \mathsf{if}\ p\,x\ \mathsf{then}\ x\ \mathsf{else}\ \mathtt{first}\ p\ (x+1)$.

Proof Sketch. We show $\Phi \triangleq \forall n : \mathbb{N}.\ \mathtt{first}\ f\ n \preceq^{\mathsf{rr}}_{\mathbb{N}} \mathtt{first}\ g\ n$. Since $(\preceq^{\mathsf{rr}}_{G})$ is defined in terms of Hoare triples, by HOARE-LÖB, we have to show

$$\forall n : \mathbb{N}.\ \{\mathsf{src}(K[\mathtt{first}\ g\ n]) * \triangleright\Phi\}\ \mathtt{first}\ f\ n\ \{v.\ \mathsf{src}(K[v]) * v \in \mathbb{N}\}_{\mathsf{rr}}.$$

By executing a pure step of the recursive function first in the target (the recursive unfolding) using PURE-TGT-RR, we have to, in turn, show

$$\begin{array}{l}\{\mathsf{src}(K[\mathtt{first}\ g\ n]) * \Phi\}\\ \quad (\lambda x.\ \mathsf{if}\ f\ x\ \mathsf{then}\ x\ \mathsf{else}\ \mathtt{first}\ f\,(x+1))\ n\\ \{v.\ \mathsf{src}(K[v]) * v \in \mathbb{N}\}_{\mathsf{rr}}\end{array}$$

Note that the later modality ($\triangleright$) has been stripped from Φ in the precondition. Similarly, we can execute the function first in the source for one step to "$(\lambda x.\ \mathsf{if}\ g\ x\ \mathsf{then}\ x\ \mathsf{else}\ \mathtt{first}\ g(x+1))\ n$" by rule PURE-SRC-RR:

$$\begin{array}{l}\{\mathsf{src}(K[(\lambda x.\ \mathsf{if}\ g\ x\ \mathsf{then}\ x\ \mathsf{else}\ \mathtt{first}\ g(x+1))\ n]) * \Phi\}\\ \quad (\lambda x.\ \mathsf{if}\ f\ x\ \mathsf{then}\ x\ \mathsf{else}\ \mathtt{first}\ f\,(x+1))\ n\\ \{v.\ \mathsf{src}(K[v]) * v \in \mathbb{N}\}_{\mathsf{rr}}\end{array}$$

The rest then follows by executing more pure steps and using BIND-TGT-RR to execute $f\ n$ and $g\ n$ in the source and the target using the assumption $f\ n \preceq^{\mathsf{rr}}_{\mathbb{B}} g\ n$ (for the evaluation context $K' \triangleq K[\mathsf{if} \bullet \mathsf{then}\ n\ \mathsf{else}\ \mathtt{first}\ g\ (n+1)]$ in the source). Depending on the outcome, we either (1) end the execution in both the source and the target, or (2) execute the respective recursive occurrence of $\mathtt{first}\ f\ (n+1)$ (resp. $\mathtt{first}\ g\ (n+1)$) using the assumption $\Phi = \forall m.\ \mathtt{first}\ f\ m \preceq^{\mathsf{rr}}_{\mathbb{N}} \mathtt{first}\ g\ m$. ☐

Problem. The sketched approach to refinements in Iris works well for proving result refinements, but it is not adequate for terminating-preserving refinements. For example, defining $e_{\mathsf{loop}} \triangleq \mathtt{first}\ (\lambda\_.\ \mathsf{false})\ 0$, we can prove $e_{\mathsf{loop}} \preceq^{\mathsf{rr}}_{\mathbb{N}} 0$ by LÖB induction, analogously to the proof of Lemma 42:

Lemma 43. $e_{\mathsf{loop}} \preceq^{\mathsf{rr}}_{\mathbb{N}} 0$

Proof Sketch. We show $\Phi \triangleq \forall n : \mathbb{N}.\ \mathtt{first}\ (\lambda\_.\ \mathsf{false})\ n \preceq^{\mathsf{rr}}_{\mathbb{N}} 0$. By HOARE-LÖB, we have to show

$$\forall n : \mathbb{N}.\ \{\mathsf{src}(K[0]) * \triangleright\Phi\}\ \mathtt{first}\ (\lambda\_.\ \mathsf{false})\ n\ \{v.\ \mathsf{src}(K[v]) * v \in \mathbb{N}\}_{\mathsf{rr}}.$$

By executing a pure step of the recursive function `first` in the target (the recursive unfolding) using pure-tgt-rr, it remains to show

$$\begin{array}{l}\{\mathsf{src}(K[0]) * \Phi\} \\ \quad (\lambda x.\, \mathsf{if}\ (\lambda\_.\, \mathsf{false})\ x\ \mathsf{then}\ x\ \mathsf{else}\ \mathtt{first}\ (\lambda\_.\, \mathsf{false})(x+1))\ n \\ \{v.\, \mathsf{src}(K[v]) * v \in \mathbb{N}\}_{\mathsf{rr}}\end{array}$$

Executing more pure steps, we reach

$$\{\mathsf{src}(K[0]) * \Phi\}\ \mathsf{if\ false\ then}\ n\ \mathsf{else}\ \mathtt{first}\ (\lambda\_.\, \mathsf{false})(n+1)\ \{v.\, \mathsf{src}(K[v]) * v \in \mathbb{N}\}_{\mathsf{rr}}$$

and then $\{\mathsf{src}(K[0]) * \Phi\}\ \mathtt{first}\ (\lambda\_.\, \mathsf{false})\ (n+1)\ \{v.\, \mathsf{src}(K[v]) * v \in \mathbb{N}\}_{\mathsf{rr}}$. The claim follows from $\Phi = \forall m : \mathbb{N}.\, \mathtt{first}\ (\lambda\_.\, \mathsf{false})\ m \preceq_{\mathbb{N}}^{\mathsf{rr}} 0$ (for $m \triangleq n+1$). □

Note that $e_{\mathrm{loop}} \preceq_{\mathbb{N}}^{\mathsf{rr}} 0$ (*i.e.,* Lemma 43) is clearly not a termination-preserving refinement, since the target always diverges while the source has terminated.

7.1.2 Termination-Preserving Refinements

Let us now turn to Refinement$_{\mathsf{SHL}}$ and discuss how it addresses this problem. In the following, we present the logical connectives of Refinement$_{\mathsf{SHL}}$, their intuitive semantics, and their proof rules. We postpone proving that it is adequate (*i.e.,* that it ensures termination preserving refinements) to §7.3.

Later stripping and source steps. Let us reconsider the argument for why $e_{\mathrm{loop}} \preceq_{\mathbb{N}}^{\mathsf{rr}} 0$ is provable with the rules discussed in §7.1.1. There, we could prove a refinement of a diverging target in Iris, without constructing a diverging source execution. To avoid the same happening in Refinement$_{\mathsf{SHL}}$, we identify the problem that allowed the target to diverge (in terms of the rules of the logic): the interplay of Löb induction and the target stepping rules. Specifically, as the proof of Lemma 43 shows, using Löb induction, we can assume the goal under a later modality ($\triangleright$). Once we perform a target step (*e.g.,* using pure-tgt-rr), we can strip off the later, regardless of whether we have already performed a source step (*e.g.,* using pure-src-rr) or not.

To avoid this issue, we ensure that stripping-off a later requires both a target *and* a source step. To do so, Refinement$_{\mathsf{SHL}}$ uses two different Hoare triples:

$$P, Q ::= \cdots \mid \{P\}\ e\ \{v.\, Q(v)\}_{\mathsf{tpr}} \mid \langle\!\{P\}\!\rangle\ e\ \langle\!\{v.\, Q(v)\}\!\rangle_{\mathsf{tpr}}$$

The *source-stepping* triple $\{P\}\ e\ \{v.\, Q(v)\}_{\mathsf{tpr}}$ allows us to perform a step in the source, strip off a later, and continue with the target. (Here, the subscript "$_{\mathsf{tpr}}$" indicates that this Hoare triple is used for proving termination-preserving refinements.) The *target-stepping* triple $\langle\!\{P\}\!\rangle\ e\ \langle\!\{v.\, Q(v)\}\!\rangle_{\mathsf{tpr}}$ allows us to perform a step in the target and continue with the source. To strip off a later, we always need a roundtrip between both triples, thereby necessitating a step in both the target and source.

It is important to note that in the above description of the roles of the triples $\{P\}\ e\ \{v.\, Q(v)\}_{\mathsf{tpr}}$ and $\langle\!\{P\}\!\rangle\ e\ \langle\!\{v.\, Q(v)\}\!\rangle_{\mathsf{tpr}}$, we have *shifted* which side gets to eliminate a later modality: in Refinement$_{\mathsf{SHL}}$, the *source steps* are the ones that allow us to eliminate a later modality (whereas in §7.1.1 it was the target steps). This shift will allow us to give a very general rule for stuttering the source

value-tgt-tpr
$$\{P(v)\}\; v\; \{w.\, P(w)\}_{\mathsf{tpr}}$$

bind-tgt-tpr
$$\frac{\{P\}\; e_t\; \{v.\, Q(v)\}_{\mathsf{tpr}} \qquad \forall v.\, \{Q(v)\}\; K[v]\; \{w.\, R(w)\}_{\mathsf{tpr}}}{\{P\}\; K[e_t]\; \{w.\, R(w)\}_{\mathsf{tpr}}}$$

pure-tgt-tpr
$$\frac{\{P\}\; e_t'\; \{v.\, Q(v)\}_{\mathsf{tpr}} \qquad e_t \to_{\mathsf{pure}} e_t'}{\langle\!\langle P\rangle\!\rangle\; e_t\; \langle\!\langle v.\, Q(v)\rangle\!\rangle_{\mathsf{tpr}}}$$

pure-src-tpr
$$\frac{\langle\!\langle \mathsf{src}(K[e_s']) * P\rangle\!\rangle\; e_t\; \langle\!\langle v.\, Q(v)\rangle\!\rangle_{\mathsf{tpr}} \qquad e_s \to_{\mathsf{pure}} e_s' \qquad e_t \notin \mathit{Val}}{\{\mathsf{src}(K[e_s]) * \triangleright P\}\; e_t\; \{v.\, Q(v)\}_{\mathsf{tpr}}}$$

store-tgt-tpr
$$\langle\!\langle \ell \mapsto v\rangle\!\rangle\; \ell := w\; \langle\!\langle \_.\, \ell \mapsto w\rangle\!\rangle_{\mathsf{tpr}}$$

store-src-tpr
$$\frac{\langle\!\langle \ell \mapsto_{\mathsf{src}} w * \mathsf{src}(K[()]) * P\rangle\!\rangle\; e_t\; \langle\!\langle u.\, Q(u)\rangle\!\rangle_{\mathsf{tpr}} \qquad e_t \notin \mathit{Val}}{\{\ell \mapsto_{\mathsf{src}} v * \mathsf{src}(K[\ell := w]) * \triangleright P\}\; e_t\; \{u.\, Q(u)\}_{\mathsf{tpr}}}$$

load-tgt-tpr
$$\langle\!\langle \ell \mapsto v\rangle\!\rangle\; !\ell\; \langle\!\langle w.\, w = v * \ell \mapsto v\rangle\!\rangle_{\mathsf{tpr}}$$

load-src-tpr
$$\frac{\langle\!\langle \ell \mapsto_{\mathsf{src}} v * \mathsf{src}(K[v]) * P\rangle\!\rangle\; e_t\; \langle\!\langle w.\, Q(w)\rangle\!\rangle_{\mathsf{tpr}} \qquad e_t \notin \mathit{Val}}{\{\ell \mapsto_{\mathsf{src}} v * \mathsf{src}(K[!\ell]) * \triangleright P\}\; e_t\; \{w.\, Q(w)\}_{\mathsf{tpr}}}$$

ref-tgt-tpr
$$\langle\!\langle \mathsf{True}\rangle\!\rangle\; \mathsf{ref}(v)\; \langle\!\langle w.\, \exists \ell.\, w = \ell * \ell \mapsto v\rangle\!\rangle_{\mathsf{tpr}}$$

ref-src-tpr
$$\frac{\langle\!\langle \exists \ell.\, \ell \mapsto_{\mathsf{src}} v * \mathsf{src}(K[\ell]) * P\rangle\!\rangle\; e_t\; \langle\!\langle w.\, Q(w)\rangle\!\rangle_{\mathsf{tpr}} \qquad e_t \notin \mathit{Val}}{\{\mathsf{src}(K[\mathsf{ref}(v)]) * \triangleright P\}\; e_t\; \{w.\, Q(w)\}_{\mathsf{tpr}}}$$

Figure 7.3: A selection of proof rules for proving termination-preserving refinements in Refinement$_{\mathsf{SHL}}$.

in §7.1.3 (see stutter-src-tpr in Fig. 7.4). For now, all that matters is that to strip off a later, we need to take both a source and a target step.

For the simulation, we define (using the new Hoare triples):

$$e_t \preceq_G^{\mathsf{tpr}} e_s \triangleq \forall K.\, \{\mathsf{src}(K[e_s])\}\; e_t\; \{v.\, \mathsf{src}(K[v]) * v \in G\}_{\mathsf{tpr}}$$

Proof rules. The proof rules of Refinement$_{\mathsf{SHL}}$ are depicted in Fig. 7.3. They mirror the rules from Fig. 7.2 except that now (1) the target-stepping triples $\langle\!\langle P\rangle\!\rangle\; e\; \langle\!\langle v.\, Q(v)\rangle\!\rangle_{\mathsf{tpr}}$ are used for the target steps and (2) the source steps eliminate laters. For example, we can use pure-src-tpr to execute a pure step in the source, which requires us to prove a target-stepping triple next. We can then use, for example, pure-tgt-tpr to get back to a source-stepping triple.

Let us see these Hoare triples in action by reproving Lemma 42 from §7.1.1 in Refinement$_{\mathsf{SHL}}$. Note that since the definition of the simulation changed, this now gives us a termination-preserving refinement.

Lemma 44. *If* $\forall n : \mathbb{N}.\, f\, n \preceq_{\mathbb{B}}^{\mathsf{tpr}} g\, n$, *then* $\forall n : \mathbb{N}.\, \mathtt{first}\, f\, n \preceq_{\mathbb{N}}^{\mathsf{tpr}} \mathtt{first}\, g\, n$.

Proof Sketch. We show $\Phi \triangleq \forall n : \mathbb{N}.\, \mathtt{first}\, f\, n \preceq_{\mathbb{N}}^{\mathsf{tpr}} \mathtt{first}\, g\, n$. By hoare-löb, it remains to show that

$$\forall n : \mathbb{N}.\, \{\mathsf{src}(K[\mathtt{first}\, g\, n]) * \triangleright \Phi\}\; \mathtt{first}\, f\, n\; \{v.\, \mathsf{src}(K[v]) * v \in \mathbb{N}\}_{\mathsf{tpr}}.$$

By taking a source step using pure-src-tpr, we have to show:

$$\begin{array}{l} \langle\!\langle \mathsf{src}(K[(\lambda x.\, \mathsf{if}\, g\, x\, \mathsf{then}\, x\, \mathsf{else}\, \mathtt{first}\, g(x+1))\, n]) * \Phi\rangle\!\rangle \\ \quad \mathtt{first}\, f\, n \\ \langle\!\langle v.\, \mathsf{src}(K[v]) * v \in \mathbb{N}\rangle\!\rangle_{\mathsf{tpr}} \end{array}$$

Note that (1) we have switched to the target-stepping Hoare triple and that (2) the later modality ($\triangleright$) has been stripped from the precondition (from a source step). Similarly, we can execute the function in the target for one step with PURE-TGT-TPR, and thereby switch back to the source-stepping Hoare triple:

$$\begin{aligned}
&\{\mathsf{src}(K[(\lambda x.\ \mathsf{if}\ g\,x\ \mathsf{then}\ x\ \mathsf{else}\ \mathtt{first}\ g(x+1))\ n]) * \Phi\}\\
&\quad(\lambda x.\ \mathsf{if}\ f\,x\ \mathsf{then}\ x\ \mathsf{else}\ \mathtt{first}\ f(x+1))\ n\\
&\{v.\ \mathsf{src}(K[v]) * v\in\mathbb{N}\}_{\mathsf{tpr}}
\end{aligned}$$

The rest of the proof is analogous to Lemma 42. □

In contrast, we cannot prove $e_{\mathsf{loop}} \preceq_{\mathbb{N}}^{\mathsf{tpr}} 0$. There are no corresponding source steps for the target steps of e_{loop}. Thus, we cannot use LÖB induction (as in Lemma 43), since source steps are necessary to remove the later that arises.

7.1.3 Stuttering

Before we can verify more interesting examples with Refinement$_{\mathsf{SHL}}$ (in §8), we first need to add *stuttering*. That is, the back-and-forth between target stepping and source stepping discussed in §7.1.2 leads to a *lock-step simulation*, a simulation where there is a one-to-one correspondence between target and source steps. In general, for programs like `memo_rec` from §5, such a simulation is too restrictive. For them, we need rules that can be used to advance the source (without stepping the target), and we need rules that can be used to advance the target (without stepping the source)—otherwise known as *stuttering*.[10]

Refinement$_{\mathsf{SHL}}$ offers two different forms of stuttering: (1) *bounded source stuttering* and (2) *no-later stuttering*. The first form allows one to stutter the source (with the ability to remove laters) via a form of "*stutter credits*". The second form allows one to stutter the source or the target for an arbitrary number of steps—but without the ability to remove laters after a step. The proof rules for both (extending those of Fig. 7.3) are depicted in Fig. 7.4.

[10] Stuttering is needed for `memo_rec`, because the memoized version and the original behave very differently at times: if we have a cache hit (*i.e.,* an argument was previously memoized), then the original version will recompute the result and the memoized version will look up the value in the table. The number of steps each version takes is completely independent: it can change from argument to argument, and it depends on how many arguments have already been cached in the hash table.

Bounded source stuttering. To enable bounded stuttering of the source, Refinement$_{\mathsf{SHL}}$ generalizes the source stepping rules (from Fig. 7.3): the idea is that, whenever we take a source step, we can pick a natural number n and then stutter the source for n target steps before we have to take another source step.

The rules of the bounded source stuttering mechanism are depicted in the upper half of Fig. 7.4. The key piece of the mechanism is an additional resource, the *stutter credits* $\$_{\mathsf{st}}\ n$. A stutter credit $\$_{\mathsf{st}}\ 1$ means we get to stutter the source for one step by flipping to the target-stepping triple with SRC-STUTTER-CRED-TPR (and removing a later modality from the precondition in the process).[11] We can split and combine stutter credits as we see fit with STUTTER-CRED-SPLIT. And we get to allocate an arbitrary number of additional stutter credits with every source step. More specifically, we can use the rules PURE-SRC-STUTTER-TPR, STORE-SRC-STUTTER-TPR, LOAD-SRC-STUTTER-TPR, and REF-SRC-STUTTER-TPR to step the source and obtain new credits. They generalize the rules from Fig. 7.3 by allowing us to allocate $\$_{\mathsf{st}}\ n$ additional stutter credits.

[11] In Part III, we will see another form of credits that allow one to remove later modalities, *later credits.* Crucially, unlike the stutter credits presented here, they are not tied to (source or target) programs and instead allow removing laters as part of purely logical reasoning. They also allow one to eliminate *more than one later* in between actual program steps, which these rules do not.

With these stutter credits, we can then prove, for example,

Lemma 45. *Let* $f \triangleq \lambda x.\ x-1 \geq 41$ *and* $g \triangleq \lambda x.\ x \geq 42$. *Then* $f\ n \preceq_{\mathbb{B}}^{\mathsf{tpr}} g\ n$.

Bounded Source Stuttering

src-stutter-cred-tpr

$$\frac{\langle\!\langle P \rangle\!\rangle\ e_t\ \langle\!\langle v.\, Q(v) \rangle\!\rangle_{\mathrm{tpr}} \qquad e_t \notin \mathit{Val}}{\{\$_{\mathrm{st}}\ 1 * \triangleright P\}\ e_t\ \{v.\, Q(v)\}_{\mathrm{tpr}}}$$

stutter-cred-split

$$\$_{\mathrm{st}}\ (n+m) \dashv\vdash \$_{\mathrm{st}}\ n * \$_{\mathrm{st}}\ m$$

pure-src-stutter-tpr

$$\frac{\langle\!\langle \mathsf{src}(K[e_s']) * \$_{\mathrm{st}}\ n * P \rangle\!\rangle\ e_t\ \langle\!\langle v.\, Q(v) \rangle\!\rangle_{\mathrm{tpr}} \qquad e_s \rightarrow_{\mathrm{pure}} e_s' \qquad e_t \notin \mathit{Val}}{\{\mathsf{src}(K[e_s]) * \triangleright P\}\ e_t\ \{v.\, Q(v)\}_{\mathrm{tpr}}}$$

store-src-stutter-tpr

$$\frac{\langle\!\langle \ell \mapsto_{\mathrm{src}} w * \$_{\mathrm{st}}\ n * \mathsf{src}(K[()]) * P \rangle\!\rangle\ e_t\ \langle\!\langle u.\, Q(u) \rangle\!\rangle_{\mathrm{tpr}} \qquad e_t \notin \mathit{Val}}{\{\ell \mapsto_{\mathrm{src}} v * \mathsf{src}(K[\ell := w]) * \triangleright P\}\ e_t\ \{u.\, Q(u)\}_{\mathrm{tpr}}}$$

load-src-stutter-tpr

$$\frac{\langle\!\langle \ell \mapsto_{\mathrm{src}} v * \$_{\mathrm{st}}\ n * \mathsf{src}(K[v]) * P \rangle\!\rangle\ e_t\ \langle\!\langle w.\, Q(w) \rangle\!\rangle_{\mathrm{tpr}} \qquad e_t \notin \mathit{Val}}{\{\ell \mapsto_{\mathrm{src}} v * \mathsf{src}(K[!\ell]) * \triangleright P\}\ e_t\ \{w.\, Q(w)\}_{\mathrm{tpr}}}$$

ref-src-stutter-tpr

$$\frac{\langle\!\langle \exists \ell.\, \ell \mapsto_{\mathrm{src}} v * \$_{\mathrm{st}}\ n * \mathsf{src}(K[\ell]) * P \rangle\!\rangle\ e_t\ \langle\!\langle w.\, Q(w) \rangle\!\rangle_{\mathrm{tpr}} \qquad e_t \notin \mathit{Val}}{\{\mathsf{src}(K[\mathsf{ref}(v)]) * \triangleright P\}\ e_t\ \{w.\, Q(w)\}_{\mathrm{tpr}}}$$

No-Later Stuttering

stutter-src-tpr

$$\frac{\langle\!\langle P \rangle\!\rangle\ e_t\ \langle\!\langle v.\, Q(v) \rangle\!\rangle_{\mathrm{tpr}} \qquad e_t \notin \mathit{Val}}{\{P\}\ e_t\ \{v.\, Q(v)\}_{\mathrm{tpr}}}$$

stutter-tgt-tpr

$$\frac{\{P\}\ e_t\ \{v.\, Q(v)\}_{\mathrm{tpr}} \qquad e_t \notin \mathit{Val}}{\{\Rrightarrow_{\mathrm{src}} P\}\ e_t\ \{v.\, Q(v)\}_{\mathrm{tpr}}}$$

src-upd-return

$$P \vdash \Rrightarrow_{\mathrm{src}} P$$

src-upd-bind

$$(\Rrightarrow_{\mathrm{src}} P) * (P \mathrel{-\!\!*} \Rrightarrow_{\mathrm{src}} Q) \vdash \Rrightarrow_{\mathrm{src}} Q$$

src-upd-frame

$$(\Rrightarrow_{\mathrm{src}} P) * Q \vdash \Rrightarrow_{\mathrm{src}} P * Q$$

src-upd-pure

$$\mathsf{src}(K[e_s]) * e_s \rightarrow_{\mathrm{pure}} e_s' \vdash \Rrightarrow_{\mathrm{src}} \mathsf{src}(K[e_s'])$$

src-upd-store

$$\ell \mapsto_{\mathrm{src}} v * \mathsf{src}(K[\ell := w]) \vdash \Rrightarrow_{\mathrm{src}} \ell \mapsto_{\mathrm{src}} w * \mathsf{src}(K[()])$$

src-upd-load

$$\ell \mapsto_{\mathrm{src}} v * \mathsf{src}(K[!\ell]) \vdash \Rrightarrow_{\mathrm{src}} \ell \mapsto_{\mathrm{src}} v * \mathsf{src}(K[v])$$

src-upd-ref

$$\mathsf{src}(K[\mathsf{ref}(v)]) \vdash \Rrightarrow_{\mathrm{src}} \exists \ell.\, \ell \mapsto_{\mathrm{src}} v * \mathsf{src}(K[\ell])$$

Figure 7.4: Stuttering rules in Refinement$_{\mathsf{SHL}}$.

Here, the target requires more steps than the source, since it must evaluate $x - 1$. However, the step in the source (*i.e.,* the β-reduction of $g\,n$) can be used to allocate 1 stutter credit to evaluate $x - 1$ for $x \triangleq n$ in the target.[12]

As an aside, note that a simple trick to enable this kind of stuttering—as we will see in §7.3.2—is to pick as the source language the lexicographic product of actual source programs (*e.g.,* here SHL-expressions and heaps) and a stutter budget. Each actual source step resets the stutter budget, and—to "stutter the source"—one can simply decrease the stutter budget. In fact, this is how Tassarotti, Jung, and Harper[13] and Timany et al.[14] encode stuttering for their step-indexed termination-preserving refinements. However, since they use *finite step-indexing*, they must impose restrictions on how the stutter budget is reset (see also §10): they only allow resetting the budget to a globally fixed bound D. The reason is that without restrictions on the stutter budget, the same issue arises as in §6.3 (*i.e.,* the chosen stutter budget could depend on the current step-index). In contrast, since we use *transfinite step-indexing*

[12] For this example, we do not strictly need stuttering credits. We could also use no-later stuttering. However, in general, the bounded source stuttering is useful if we want to remove later modalities while simultaneously stuttering the source. Since examples that use this ability tend to be quite involved (*e.g.,* `memo_rec` in §8.1), we use a simple example here.

[13] Tassarotti, Jung, and Harper, "A higher-order logic for concurrent termination-preserving refinement", 2017 [TJH17].

[14] Timany et al., "Trillium: Higher-order concurrent and distributed separation logic for intensional refinement", 2024 [Tim+24a].

in Transfinite Iris (see §9), we do not need to impose any restrictions on the stutter budget: every source step can pick an arbitrary, unconstrained number of new stutter credits to allocate (see Fig. 7.4).

No-later stuttering. Besides the bounded source stuttering above, there is a second form of stuttering that $\mathrm{Refinement}_{\mathrm{SHL}}$ supports: *no-later stuttering*. That is, it allows arbitrary steps in source and target *if they do not strip off laters*. This stuttering mechanism makes the target and source steps largely independent. We can use it, for example, to then prove refinements such as

Example 46 (Fibonacci Refinement)**.**

$$\forall n : \mathbb{N}.\, \mathtt{fib}_{\mathrm{exp}}\, n \preceq^{\mathrm{tpr}}_{\mathbb{N}} \mathtt{fib}_{\mathrm{lin}}\, n \quad \textit{and} \quad \forall n : \mathbb{N}.\, \mathtt{fib}_{\mathrm{lin}}\, n \preceq^{\mathrm{tpr}}_{\mathbb{N}} \mathtt{fib}_{\mathrm{exp}}\, n$$

where $\mathtt{fib}_{\mathrm{exp}}$ takes an exponential number of steps to compute the result, and $\mathtt{fib}_{\mathrm{lin}}$ computes it linearly:

$$\begin{aligned}
\mathtt{fib}_{\mathrm{exp}} &\triangleq \mathsf{fix}\, \mathtt{fib}\, n.\, \mathsf{if}\ n \leq 1\ \mathsf{then}\ n\ \mathsf{else}\ \mathtt{fib}(n-1) + \mathtt{fib}(n-2) \\
\mathtt{fib}_{\mathrm{lin}} &\triangleq \lambda x.\, \pi_1\, (\mathtt{fibl}(x)) \\
\mathtt{fibl} &\triangleq \mathsf{fix}\, \mathsf{f}\, n.\, \mathsf{if}\ n == 0\ \mathsf{then}\ (0,1)\ \mathsf{else}\ \mathsf{let}\ (x,y) = \mathsf{f}\ (n-1)\ \mathsf{in}\ (y, x+y)
\end{aligned}$$

●

The rules of this mechanism are depicted in the lower half of Fig. 7.4. Let us start by focusing on *stuttering the source* (*i.e.*, executing target steps without source steps). The key rule for this kind of stuttering is STUTTER-SRC-TPR. It allows us to switch from the source-stepping Hoare triple $\{P\}\ e\ \{v.\, Q(v)\}_{\mathrm{tpr}}$ to the target-stepping Hoare triple $\langle\!\{P\}\!\rangle\ e\ \langle\!\{v.\, Q(v)\}\!\rangle_{\mathrm{tpr}}$ *without performing a source step*. With this rule, we can now derive, for example, the following rule for executing pure target steps unconditionally:

$$\frac{\{P\}\ e_t'\ \{v.\, Q(v)\}_{\mathrm{tpr}} \qquad e_t \to_{\mathrm{pure}} e_t'}{\{P\}\ e_t\ \{v.\, Q(v)\}_{\mathrm{tpr}}}\ \textsc{stutter-src-tpr-pure-no-later}$$

One can use this rule to stutter the source without the bookkeeping of the "stutter budget" needed for the bounded stuttering approach above. In particular, for the Fibonacci implementations in Example 46, we can prove the following result by simply stuttering (and in fact completely ignoring) the source:

Lemma 47.

$$\{\mathsf{True}\}\ \mathtt{fib}_{\mathrm{exp}}(n)\ \{v.\, v = F_n\}_{\mathrm{tpr}} \qquad \{\mathsf{True}\}\ \mathtt{fib}_{\mathrm{lin}}(n)\ \{v.\, v = F_n\}_{\mathrm{tpr}}$$

for all natural numbers n, where $F_0 = 0$, $F_1 = 1$, and $F_{n+2} = F_{n+1} + F_n$.

However, the no-later stuttering *does not* subsume the bounded source stuttering. The reason is that STUTTER-SRC-TPR—unlike the source stuttering of SRC-STUTTER-CRED-TPR—does not allow one to remove a later modality from the precondition. This is for a good reason. It would be *unsound* (for termination-preserving refinements) to also allow removing a later modality. More specifically, if STUTTER-SRC-TPR allowed us to remove a later,[15] then it would allow a round-trip analogous to STUTTER-SRC-TPR-PURE-NO-LATER, but *with the ability to eliminate a later modality.* Thus, we would suddenly be able to replay the proof of $e_{\mathrm{loop}} \preceq^{\mathrm{rr}}_{\mathbb{N}} 0$ from Lemma 43 using Löb induction.

[15] The **unsound rule** would be

$$\frac{\langle\!\{P\}\!\rangle\ e_t\ \langle\!\{v.\, Q(v)\}\!\rangle_{\mathrm{tpr}} \qquad e_t \notin \mathit{Val}}{\{\triangleright P\}\ e_t\ \{v.\, Q(v)\}_{\mathrm{tpr}}}$$

which does not hold in $\mathrm{Refinement}_{\mathrm{SHL}}$.

The rule STUTTER-SRC-TPR also demonstrates why it is important that we have shifted the later stripping from the target rules (*e.g.*, see PURE-TGT-RR in Fig. 7.2) to the source rules (*e.g.*, see PURE-SRC-TPR in Fig. 7.3). If target steps still allowed the elimination of later modalities,[16] then STUTTER-SRC-TPR would analogously allow replaying the proof of $e_{\text{loop}} \preceq^{\text{rr}}_{\mathbb{N}} 0$ from Lemma 43 using LÖB induction.

[16] The **unsound rule** would be

$$\frac{\{P\}\ e'_t\ \{v.\,Q(v)\}_{\text{tpr}} \qquad e_t \rightarrow_{\text{pure}} e'_t}{\langle\{\triangleright P\}\rangle\ e_t\ \langle\{v.\,Q(v)\}\rangle_{\text{tpr}}}$$

which does not hold in Refinement$_{\text{SHL}}$.

Besides stuttering the source, Refinement$_{\text{SHL}}$ also provides rules for *stuttering the target* without a later. Concretely, it provides an additional update modality $\Rrightarrow_{\text{src}} P$ for updating the source expression. It behaves much like the regular ghost state update $\Rrightarrow P$: we can introduce it (SRC-UPD-RETURN), we can compose it with (SRC-UPD-BIND), and we can frame assertions R into it (SRC-UPD-FRAME). Moreover, we can execute it in the precondition of a Hoare triple (STUTTER-TGT-TPR).[17] To execute a source step, we can use the rules SRC-UPD-PURE, SRC-UPD-STORE, SRC-UPD-LOAD, and SRC-UPD-REF. For example, we can use SRC-UPD-PURE to execute a pure step $e_s \rightarrow_{\text{pure}} e'_s$.

[17] The rule STUTTER-TGT-TPR is the analogue of the rule HOARE-UPD in §3.6.

Note that, in contrast to source stuttering (STUTTER-SRC-TPR), target stuttering (STUTTER-TGT-TPR) does not switch Hoare triples. Instead, it simply allows one to execute many source steps where one could previously execute only one.

For the Fibonacci implementations from Example 46, we can use this form of stuttering to prove the following source updates:

Lemma 48.

$$\mathsf{src}(K[\mathtt{fib}_{\text{exp}}\ n]) \vdash \Rrightarrow_{\text{src}} \mathsf{src}(K[F_n]) \qquad \mathsf{src}(K[\mathtt{fib}_{\text{lin}}\ n]) \vdash \Rrightarrow_{\text{src}} \mathsf{src}(K[F_n])$$

for all natural numbers n, where $F_0 = 0$, $F_1 = 1$, and $F_{n+2} = F_{n+1} + F_n$.

We can then combine both forms of stuttering to derive:

Corollary 49.

$$\forall n : \mathbb{N}.\, \mathtt{fib}_{\text{exp}}\ n \preceq^{\text{tpr}}_{\mathbb{N}} \mathtt{fib}_{\text{lin}}\ n \qquad \forall n : \mathbb{N}.\, \mathtt{fib}_{\text{lin}}\ n \preceq^{\text{tpr}}_{\mathbb{N}} \mathtt{fib}_{\text{exp}}\ n$$

Proof. By combining Lemma 47 for executing the resp. target and Lemma 48 for executing the resp. source to the result F_n, using STUTTER-TGT-TPR to execute the source updates. □

7.2 Termination

Having discussed Refinement$_{\text{SHL}}$ for proving termination-preserving refinements, let us now turn to how we can use Transfinite Iris to prove termination. Recall from §6.5 that termination and termination-preserving refinement are closely related. In the latter, if the source always terminates, then the target always terminates. Thus, we obtain a proof technique for termination if the source reduction relation is the inverse of a well-founded relation.

Below, we instantiate Transfinite Iris's liveness logic with ordinals as the source and SHL as the target. We obtain Termination$_{\text{SHL}}$—a program logic for proving termination in the sequential fragment of HeapLang. Termination$_{\text{SHL}}$ generalizes what is known as time credits[18] to *transfinite* time credits. As we will see below, they enable termination arguments based on *dynamic* information learned during program execution. Transfinite time credits are themselves not new. They were already used in an earlier logic by Rocha Pinto et al.[19] However, the logic of Rocha Pinto et al. is *not step-indexed* and thus it does

[18] Atkey, "Amortised resource analysis with separation logic", 2011 [Atk11]; Pilkiewicz and Pottier, "The essence of monotonic state", 2011 [PP11]; Mével, Jourdan, and Pottier, "Time credits and time receipts in Iris", 2019 [MJP19].

[19] Rocha Pinto et al., "Modular termination verification for non-blocking concurrency", 2016 [Roc+16].

value-term
$$\{P(v)\}\ v\ \{w.\ P(w)\}_{\mathsf{term}}$$

bind-term
$$\frac{\{P\}\ e\ \{v.\ Q(v)\}_{\mathsf{term}} \qquad \forall v.\ \{Q(v)\}\ K[v]\ \{w.\ R(w)\}_{\mathsf{term}}}{\{P\}\ K[e]\ \{w.\ R(w)\}_{\mathsf{term}}}$$

split-cred-term
$$\$(\alpha \oplus \beta) \Leftrightarrow \$\alpha * \$\beta$$

store-term
$$\langle\!\langle \ell \mapsto v \rangle\!\rangle\ \ell := w\ \langle\!\langle \_.\ \ell \mapsto w \rangle\!\rangle_{\mathsf{term}}$$

load-term
$$\langle\!\langle \ell \mapsto v \rangle\!\rangle\ !\ell\ \langle\!\langle w.\ w = v * \ell \mapsto v \rangle\!\rangle_{\mathsf{term}}$$

ref-term
$$\langle\!\langle \mathsf{True} \rangle\!\rangle\ \mathsf{ref}(v)\ \langle\!\langle w.\ \exists \ell.\ w = \ell * \ell \mapsto v \rangle\!\rangle_{\mathsf{term}}$$

pure-term
$$\frac{\{P\}\ e'\ \{v.\ Q(v)\}_{\mathsf{term}} \qquad e \rightarrow_{\mathsf{pure}} e'}{\langle\!\langle P \rangle\!\rangle\ e\ \langle\!\langle v.\ Q(v) \rangle\!\rangle_{\mathsf{term}}}$$

flip-term
$$\frac{\langle\!\langle P \rangle\!\rangle\ e\ \langle\!\langle v.\ Q(v) \rangle\!\rangle_{\mathsf{term}} \qquad e \notin Val}{\{P\}\ e\ \{v.\ Q(v)\}_{\mathsf{term}}}$$

spend-cred-term
$$\frac{\langle\!\langle \$\beta * P \rangle\!\rangle\ e\ \langle\!\langle v.\ Q(v) \rangle\!\rangle_{\mathsf{term}} \qquad \beta < \alpha \qquad e \notin Val}{\{\$\alpha * \triangleright P\}\ e\ \{v.\ Q(v)\}_{\mathsf{term}}}$$

Figure 7.5: A selection of proof rules for proving termination in $\mathsf{Termination}_{\mathsf{SHL}}$.

not support impredicative invariants. In contrast, in $\mathsf{Termination}_{\mathsf{SHL}}$, step-indexing equips us with impredicative invariants, which we use in our examples to handle shared, mutable, higher-order state. Concretely, we demonstrate the power of our technique by mechanizing two examples (§8.2 and §8.3), including (in just 850 lines of Rocq) a stronger version of the main theorem of Spies, Krishnaswami, and Dreyer:[20] termination of a linear language with asynchronous channels.

[20] Spies, Krishnaswami, and Dreyer, "Transfinite step-indexing for termination", 2021 [SKD21].

The program logic. Compared to $\mathsf{Refinement}_{\mathsf{SHL}}$, we obtain slightly different logical connectives and proof rules in $\mathsf{Termination}_{\mathsf{SHL}}$ by picking ordinals as the source language. Instead of the resources $\ell \mapsto_{\mathsf{src}} v$, $\mathsf{src}(e)$, and $\$_{\mathsf{st}}\ n$, we have a connective $\$\alpha$ referring to the ordinal source (see §7.3.2 for the definition). We once again obtain two Hoare triples. In $\mathsf{Termination}_{\mathsf{SHL}}$, they are denoted $\{P\}\ e\ \{v.\ Q(v)\}_{\mathsf{term}}$ and $\langle\!\langle P \rangle\!\rangle\ e\ \langle\!\langle v.\ Q(v) \rangle\!\rangle_{\mathsf{term}}$ to indicate that we are proving termination.

The rules of $\mathsf{Termination}_{\mathsf{SHL}}$ are depicted in Fig. 7.5. They mirror the target rules of $\mathsf{Refinement}_{\mathsf{SHL}}$ in Fig. 7.3. The only rules of note are spend-cred-term, flip-term, and split-cred-term. The rule flip-term is the analog of stutter-src-tpr: it allows us to flip from $\{P\}\ e\ \{v.\ Q(v)\}_{\mathsf{term}}$ to $\langle\!\langle P \rangle\!\rangle\ e\ \langle\!\langle v.\ Q(v) \rangle\!\rangle_{\mathsf{term}}$ (without a later). The rule spend-cred-term replaces the source stepping rules: if we can decrease the ordinal source $\$\alpha$, then we get to eliminate a later. We will discuss the rule split-cred-term below.

With $\mathsf{Termination}_{\mathsf{SHL}}$, we can prove that an expression e terminates safely in a value (see Theorem 51 in §7.3.2) by proving

$$\mathsf{terminates}(e) \triangleq \exists \alpha.\ \{\$\alpha\}\ e\ \{\_.\mathsf{True}\}_{\mathsf{term}}.$$

As mentioned above, the ordinal source $\$\alpha$ generalizes *time credits*,[21] which are traditionally used for proving complexity results with separation logic.[22] Traditional time credits enable one to prove the safety property of *bounded termination* (*i.e.,* they enable one to prove that a program "terminates in n steps of computation", where the bound n has to be fixed up-front). What we obtain by using ordinals in Transfinite Iris are *transfinite* time credits. They go beyond bounded termination: they allow us to prove the liveness property of *termination*[23] for examples where it is non-trivial (if not impossible) to determine finite bounds.

[21] Atkey, "Amortised resource analysis with separation logic", 2011 [Atk11]; Pilkiewicz and Pottier, "The essence of monotonic state", 2011 [PP11]; Mével, Jourdan, and Pottier, "Time credits and time receipts in Iris", 2019 [MJP19].

[22] The rule flip-term makes $\mathsf{Termination}_{\mathsf{SHL}}$ quite flexible, because we only need to spend a time credit when we want to remove a later from the precondition. However, it also means $\mathsf{Termination}_{\mathsf{SHL}}$ is not suitable for proving complexity results anymore, since not every step requires spending a credit.

[23] It is well-known that bounded termination is a safety property while termination (without a bound) is a liveness property [SKD21]. Bounded termination can be falsified by exhibiting an execution which does not terminate within the given bound, a finite prefix. For termination, this is not the case: termination can only be falsified by an *infinite* execution.

Time credits. To illustrate why this is useful, let us consider an example. Suppose we have a function f that returns a natural number, and we want to prove that $e_{\text{two}} \triangleq f() + f()$ terminates. If n_f is the maximum number of steps it takes to compute $f()$, then we know that it takes $2 \cdot n_f + 1$ steps for e_{two} to terminate. With both traditional and transfinite time credits, this amounts to proving the Hoare triple $\left\{\$(2 \cdot n_f + 1)\right\}\ e_{\text{two}}\ \{\_. \mathsf{True}\}$, where (ignoring stuttering) one time credit has to be spent for every step of e_{two}.

To prove this triple modularly, we make use of the distinguishing feature of time credits that makes them an ideal fit for separation logic: time credits can be split and combined (similar to the stuttering credits in §7.1.3). That is, we have (as an instance of split-cred-term for the finite case) $\$(n + m) \Leftrightarrow \$n * \$m$. We use this rule to factorize the termination proof of e_{two}: we first prove termination of f as $\left\{\$n_f\right\}\ f()\ \{m.\, m \in \mathbb{N}\}$, and then we use this triple twice to prove the termination of e_{two}.

Transfinite credits. Now, consider a small generalization of the example, proving termination of:

$$\begin{aligned}&\mathsf{let}\ k = u\ ()\ \mathsf{in}\ \mathsf{let}\ a = \mathsf{ref}(0)\ \mathsf{in}\\ &\mathsf{for}\ i\ \mathsf{in}\ 0, \ldots, k-1\ \mathsf{do}\ a := {!a} + f\ ()\end{aligned}$$

Here, u is a function returning a natural number k. We compute the sum of k-times executing f, and store that in a. To verify this program compositionally, we only assume that the function u, when given enough time credits n_u, returns a natural number, *i.e.,* $\{\$n_u\}\ u()\ \{m.\, m \in \mathbb{N}\}$.

In this setting, finite time credits are no longer sufficient. The number of steps it takes to execute the whole program depends on the output of $u\ ()$. The problem is that the number of credits required here depends on dynamic information—it depends on the execution of the program.

With *transfinite* time credits (*i.e.,* ordinals), we can statically abstract over this dynamic information. That is, we can show that $\$(\omega \oplus n_u)$ credits are enough to prove termination of the whole program. Given $\$(\omega \oplus n_u)$ time credits, we can spend $\$n_u$ on the execution of $u\ ()$ with split-cred-term. After obtaining the result k, we can use spend-cred-term to decrease the ω credits to (roughly) $k \cdot n_f$, which are sufficient for the remainder of the execution.

The addition operation $\alpha \oplus \beta$ in split-cred-term is Hessenberg addition[24]—a well-behaved, commutative notion of addition on ordinals. Commutativity is essential for us to be able to use ordinals as separation logic resources, since the latter are required to form a partial commutative monoid (see §4.2).

[24] Hessenberg, *Grundbegriffe der Mengenlehre*, 1906 [Hes06].

While this example is contrived, it highlights the core problem: in compositional termination proofs, there may not be enough information available to bound the length of the execution statically. With transfinite termination bounds, we can pick the termination bound *dynamically* based on information that is only learned *during* the execution (*e.g.,* the value of k above).

7.3 The Liveness Logic

Let us now turn to the underlying **liveness logic** of Transfinite Iris of which $\mathsf{Refinement}_{\mathsf{SHL}}$ (§7.1) and $\mathsf{Termination}_{\mathsf{SHL}}$ (§7.2) are instantiations. We first discuss—at the same level of abstraction[25] as in §4.1—the weakest precondition

[25] Similar to the discussion in §4.1, we make some simplifications here to ease the presentation. In particular, we again omit fancy updates and their masks, and we omit how support for "non-atomic invariants" (*i.e.,* impredicative invariants for sequential languages that can be opened for multiple steps) is integrated into this definition. Both clutter the presentation and are completely standard for Iris.

The Weakest Precondition

$$\begin{aligned}
&\mathbf{wp}\ v\ \{w.\, Q(w)\} \triangleq_{\mathit{lfp}} \forall h, a.\ \mathrm{SI}(h) * I(a) \mathrel{-\!\!*} \Rrightarrow \mathrm{SI}(h) * I(a) * Q(v) \\
&\mathbf{wp}\ e\ \{w.\, Q(w)\} \triangleq_{\mathit{lfp}} \forall h, a.\ \mathrm{SI}(h) * I(a) \mathrel{-\!\!*} \Rrightarrow \qquad\qquad \text{if } e \notin \mathit{Val} \\
&\quad\left(\begin{array}{l}
\triangleright \Rrightarrow (\mathrm{progress}(e,h) * (\forall e', h'.\, (e,h) \rightsquigarrow (e',h') \mathrel{-\!\!*} \Rrightarrow (\exists a'.\, a \hookrightarrow^{+} a' * I(a') * \mathrm{SI}(h') * \mathbf{wp}\ e'\ \{w.\, Q(w)\}))) \\
\vee \quad \Rrightarrow (\mathrm{progress}(e,h) * (\forall e', h'.\, (e,h) \rightsquigarrow (e',h') \mathrel{-\!\!*} \Rrightarrow (I(a) * \mathrm{SI}(h') * \mathbf{wp}\ e'\ \{w.\, Q(w)\})))
\end{array}\right)
\end{aligned}$$

The Target-Stepping Weakest Precondition

$$\begin{aligned}
\mathbf{wp}\ e\ \langle\!\{w.\, Q(w)\}\!\rangle \triangleq{}& \forall h, a.\ \mathrm{SI}(h) * I(a) \mathrel{-\!\!*} \\
& \Rrightarrow \mathrm{progress}(e,h) * (\forall e', h'.\, (e,h) \rightsquigarrow (e',h') \mathrel{-\!\!*} \Rrightarrow (I(a) * \mathrm{SI}(h') * \mathbf{wp}\ e'\ \{w.\, Q(w)\}))
\end{aligned}$$

Figure 7.6: The liveness weakest preconditions of Transfinite Iris.

of the liveness logic (§7.3.1) and then the instantiations for Refinement$_{\mathsf{SHL}}$ and Termination$_{\mathsf{SHL}}$ (§7.3.2).

7.3.1 The Weakest Precondition

Let us start with the two kinds of Hoare triples that we have encountered in the previous sections: the triples $\langle\!\{P\}\!\rangle\ e\ \langle\!\{v.\, Q(v)\}\!\rangle$ for stepping the target, and the triples $\{P\}\ e\ \{v.\, Q(v)\}$ for stepping the source. (We have encountered them for two different instantiations of the source language, which we will discuss in §7.3.2.) Similarly to how we defined regular safety Hoare triples in Iris in terms of a weakest precondition (in §3.3), we will now define these in terms of two weakest preconditions $\mathbf{wp}\ e\ \{v.\, Q(v)\}$ and $\mathbf{wp}\ e\ \langle\!\{v.\, Q(v)\}\!\rangle$:

$$\begin{aligned}
\{P\}\ e\ \{v.\, Q(v)\} &\triangleq \Box(P \mathrel{-\!\!*} \mathbf{wp}\ e\ \{v.\, Q(v)\}) \\
\langle\!\{P\}\!\rangle\ e\ \langle\!\{v.\, Q(v)\}\!\rangle &\triangleq \Box(P \mathrel{-\!\!*} \mathbf{wp}\ e\ \langle\!\{v.\, Q(v)\}\!\rangle)
\end{aligned}$$

The weakest preconditions are defined in Fig. 7.6. Their definition is quite a mouthful, so let us unpack it step by step.

The weakest precondition $\mathbf{wp}\ e\ \{v.\, Q(v)\}$. We start with the weakest precondition $\mathbf{wp}\ e\ \{v.\, Q(v)\}$. To integrate the notion of a "source program" into the definition, we assume a *source language* $\Lambda = (A, \rightsquigarrow_{\mathrm{src}}, I)$, consisting of a type of source states A, a *source step relation* $a \rightsquigarrow_{\mathrm{src}} a'$, and a *source state interpretation* $I(a)$. The source state interpretation I, analogous to the regular state interpretation SI (see §4.1), associates source states a with resources (*e.g.,* to give meaning to the points-to assertion $\ell \mapsto_{\mathrm{src}} v$ or the time credits $\$\alpha$). All additions concerning the source language in the definition of the weakest precondition (in Fig. 7.6) are highlighted in red.

Let us now unpack the two cases of the weakest precondition $\mathbf{wp}\ e\ \{v.\, Q(v)\}$. As for the standard weakest precondition, we distinguish whether the expression is a value or a proper expression. In the value case, one must prove the postcondition $Q(v)$ underneath an update modality "$\Rrightarrow$". However, unlike for the standard weakest precondition (see §4.1), this weakest precondition makes the state interpretation $\mathrm{SI}(h)$ and the source state interpretation $I(a)$ available in the value case (and demands they are returned after the update). In doing so, it makes the state interpretation available in *both* cases, which can occasionally be useful (*e.g.,* for more complex state interpretations than the one for HeapLang in §4.1), but does not make a big difference compared to the value case of the safety weakest precondition.

The more interesting case is when e is a proper expression. In this case, we assume the state interpretation $\mathrm{SI}(h)$ and, *additionally*, the source state interpretation $I(a)$ for the current source state a. We then have *two* options:

1. *Option 1.* We get a later modality (highlighted in blue) to remove guarding laters from our assumptions (*e.g.*, as part of a Löb induction), and then we must execute at least one step of the source a (*i.e.*, we prove $a \rightsquigarrow_{\mathrm{src}}{}^{+} a'$ and establish the source state interpretation for a'), or

2. *Option 2.* There is no later modality, and we keep the source state a unchanged (*i.e.*, we re-establish $I(a)$).

We will elaborate on the difference between the two options below. The remainder of the definition (*e.g.*, that we prove progress of e, that we consider every possible successor expression e', and that we re-establish the state interpretation for the new heap h') is basically the same as in the original weakest precondition (see §4.1).

Laters and source steps. The interplay of the later modality and the source steps in this definition of $\mathbf{wp}\ e\ \{v.\ Q(v)\}$ is what enables removing laters in, *e.g.*, the source stepping rules in Fig. 7.3. More specifically, unlike for the regular weakest precondition, here the later modality in the definition of $\mathbf{wp}\ e\ \{v.\ Q(v)\}$ is associated with *source steps*: we get to remove a guarding later (*e.g.*, in PURE-SRC-TPR in Fig. 7.3 or SPEND-CRED-TERM in Fig. 7.5) if we are willing to step the source for at least one step.

If we choose Option 1 and we execute (at least) one source step (*e.g.*, in PURE-SRC-TPR in Fig. 7.3), we must still execute at least one step in the target. This is where the weakest precondition $\mathbf{wp}\ e\ \langle\!\{v.\ Q(v)\}\!\rangle$ comes in. It forces us to reason about the next step of the target before we return to $\mathbf{wp}\ e\ \{v.\ Q(v)\}$.

The target stepping weakest precondition $\mathbf{wp}\ e\ \langle\!\{v.\ Q(v)\}\!\rangle$. More specifically, the target stepping weakest precondition asks us to execute one target step *without a later*: we must show progress of e and then we must establish the source state interpretation $I(a)$ again for the current source state a and a weakest precondition for the successor expression e'. Notably, after the step (*i.e.*, for the successor expression e'), the definition *switches back* to the other weakest precondition $\mathbf{wp}\ e\ \{v.\ Q(v)\}$ (note the absence of pointy brackets).

Source stuttering. Recall (from §7.1.3) that we do not have to prove a lock-step simulation and, instead, we can stutter the source and the target. Let us first focus on source stuttering: steps in the target without corresponding source steps (*i.e.*, the rules STUTTER-SRC-TPR and FLIP-TERM). What enables this kind of stuttering is Option 2, which allows one to execute a target step without stepping the source. This option basically coincides with $\mathbf{wp}\ e\ \langle\!\{v.\ Q(v)\}\!\rangle$, justifying the switch from $\{P\}\ e\ \{v.\ Q(v)\}$ to $\langle\!\{P\}\!\rangle\ e\ \langle\!\{v.\ Q(v)\}\!\rangle$ in these rules.

However, we have to be careful. For a termination-preserving refinement, it would be unsound if we allow the source to be stuttered an *unbounded number of steps* (*i.e.*, if we allow the target to diverge). We have to make sure that another source step is executed eventually (or the target terminates). To ensure that this is the case, the definition of the weakest precondition $\mathbf{wp}\ e\ \{v.\ Q(v)\}$ uses *a least fixpoint* (as opposed to the guarded fixpoint in the weakest precondition in §4.1). The least fixpoint ensures that Option 2 is not used indefinitely and eventually Option 1 has to be chosen again (which will execute a source step).

Target stuttering. To stutter the target, we have introduced the source update modality $\Rrightarrow_{\mathrm{src}} P$ (in §7.1.3). We can now define it as follows:

$$\Rrightarrow_{\mathrm{src}} P \triangleq \forall a.\ I(a) \mathrel{-\!\!*} {\Rrightarrow} \exists a'.\ a \hookrightarrow^{*} a' * I(a') * P$$

We assume the source state interpretation $I(a)$ and execute the source for zero or more steps before we re-establish I for the new state a'. We can justify this stuttering using the fact that Option 1 allows multiple source steps to be executed and not just a single one.

The existential property and adequacy. Before we move on to the concrete instantiations of the liveness logic (§7.3.2), let us briefly discuss why we need the *existential property* for this weakest precondition $\mathbf{wp}\ e\ \{v.\ Q(v)\}$ to be adequate for termination-preserving refinements and termination. Recall (from §6.4) that the crux to get a termination-preserving refinement out of a step-indexed simulation is to lift the existentially quantified source execution outside the logic. Specifically, the simple, step-indexed simulation $t \preceq_{\triangleright} s$ in §6.3 contains an existentially quantified step of the source expression "$\exists s'.\ s \rightsquigarrow_{\mathrm{src}} s' \land \cdots$" that we need to get out for a termination-preserving refinement. The same holds true for this more complex separation logic simulation: we need to get out the existential quantification "$\exists a'.\ a \hookrightarrow^{+} a' * \cdots$" in Option 1.

The existential property allows us to extract "$\exists a'.\ a \hookrightarrow^{+} a' * \cdots$" as part of the adequacy proof. Unfortunately, the adequacy proof itself is somewhat gnarly due to the increased complexity of $\mathbf{wp}\ e\ \{v.\ Q(v)\}$ compared to $t \preceq_{\triangleright} s$ (*e.g.*, it involves an induction on the least-fixpoint in the definition of $\mathbf{wp}\ e\ \{w.\ Q(w)\}$). It can be found in the Transfinite Iris Rocq development.[26] Below, we focus on the instantiations of the adequacy proof that we obtain for termination and termination-preserving refinements. In §9, we then shine a light on how we obtain the existential property by using ordinals as step-indices.

[26] Spies et al., *Transfinite Iris appendix and Rocq development*, 2021 [Spi+21a].

7.3.2 Instantiations of the Liveness Logic

Let us now discuss the two concrete instantiations that we have seen in the previous sections: $\mathsf{Refinement}_{\mathsf{SHL}}$ (§7.1) and $\mathsf{Termination}_{\mathsf{SHL}}$ (§7.2).

Termination-preserving refinements. For $\mathsf{Refinement}_{\mathsf{SHL}}$, we choose the source language where the states are of type $A \triangleq (\mathit{Expr} \times \mathit{Heap}) \times \mathbb{N}$, meaning lexicographically ordered pairs of the current source program configuration (*i.e.*, the current source expression and heap) and the current stuttering limit (for the stuttering credits $\$_{\mathrm{st}}\ n$). We allow the following source transitions:

$$\begin{aligned} ((e,h),n) &\rightsquigarrow_{\mathrm{src}} ((e',h'),m) \qquad && \textit{if}\ \ (e,h) \rightsquigarrow (e',h') \\ ((e,h),n) &\rightsquigarrow_{\mathrm{src}} ((e,h),m) && \textit{if}\ \ m < n \end{aligned}$$

That is, we allow actual source steps (which can pick an arbitrary new stutter budget m) and stuttering steps where the actual source is unchanged and the stuttering budget decreases.

We then define the source resource theory using different instances of the authoritative resource algebra (§4.2.1):

$$I((e,h),n) \triangleq \mathsf{heap}^{\gamma_{\text{heap-src}}}(h) * \boxed{\bullet\mathsf{ex}(e)}^{\gamma_{\text{expr}}} * \boxed{\bullet n}^{\gamma_{\text{stutter}}}$$

$$\mathsf{src}(e) \triangleq \boxed{\circ\mathsf{ex}(e)}^{\gamma_{\text{expr}}} \qquad \ell \mapsto_{\text{src}} v \triangleq \ell \mapsto^{\gamma_{\text{heap-src}}} v \qquad \$_{\text{st}}\, n \triangleq \boxed{\circ n}^{\gamma_{\text{stutter}}}$$

That is, we use three different resources: $\gamma_{\text{heap-src}}$ to track the current source heap, γ_{expr} to track the current source expression, and γ_{stutter} to track the amount of stuttering credits. The resource $\gamma_{\text{heap-src}}$ is from the heap resource algebra (§4.2.2), *i.e., $Auth(Loc \xrightarrow{\text{fin}} Ex(Val))$*. The resource γ_{expr} is from the resource algebra $Auth(Ex(Expr))$, which has two elements $\circ\mathsf{ex}(e)$ and $\bullet\mathsf{ex}(e)$ that are always in sync (*i.e.,* one cannot update one without the other). The resource γ_{stutter} is from the credit resource algebra $Auth(\mathbb{N},+)$ (see §4.2.2), which contains the total amount $\bullet m$ and the fragments $\circ n$ that together add up to m.

For this instantiation, we obtain the following adequacy result for our simulation $e_t \preceq^{\text{tpr}}_{G} e_s$ (from §7.1.2):

Theorem 50. *If $\vdash e_t \preceq^{\text{tpr}}_{G} e_s$, then e_t is a termination-preserving refinement of e_s.*

Proof Sketch. This proof is similar to that of Lemma 40. To prove termination preservation, we construct an infinite execution using Transfinite Iris's existential property, but for the more elaborate weakest precondition $\textbf{wp}\ e\ \{v.\, Q(v)\}$ from Fig. 7.6. The key, as mentioned above, is extracting the source steps "$\exists a'.\, a \hookrightarrow^{+} a' * \cdots$" in the definition of $\textbf{wp}\ e\ \{v.\, Q(v)\}$. □

Termination. For Termination$_{\mathsf{SHL}}$, we choose the source language with states from $A \triangleq \mathbb{O}$, meaning *ordinals.* To be precise, we do not use the same ordinals that we will use for transfinite step-indexing. Instead, we use another copy of ordinals that we place, in Rocq, in the universe below the step-index type (see also §9). This ensures that $\mathbb{O}$ is "substantially smaller", which means we can use the existential property on it. We allow the following source transitions:

$$\alpha \rightsquigarrow_{\text{src}} \beta \qquad\qquad \textit{if}\ \ \beta < \alpha$$

That is, the only allowed transition is decreasing the ordinal.

We define the source resource theory as follows:

$$I(\alpha) \triangleq \boxed{\bullet\alpha}^{\gamma_{\text{ord}}} \qquad\qquad \$\alpha \triangleq \boxed{\circ\alpha}^{\gamma_{\text{ord}}}$$

where the resource γ_{ord} is drawn from the resource algebra $Auth(\mathbb{O},\oplus)$, which—analogous to $Auth(\mathbb{N},+)$—contains the total amount $\bullet\alpha$ and the fragments $\circ\beta$ that together add up to α. In the case of ordinals, we use Hessenberg addition[27] $\alpha \oplus \beta$ (in place of regular, commutative addition for natural numbers).

[27] Hessenberg, *Grundbegriffe der Mengenlehre*, 1906 [Hes06].

For this instantiation, we obtain the following adequacy result:

Theorem 51. *If $\vdash \mathsf{terminates}(e)$, then e terminates safely.*

Proof Sketch. Recall that $\mathsf{terminates}(e) \triangleq \exists\alpha.\, \{\$\alpha\}\ e\ \{\_.\mathsf{True}\}$. We use the existential property for the quantification over α in the definition of terminates, and then we use similar reasoning as in the adequacy sketch of the termination-preserving refinement (see Theorem 50) to show that e cannot diverge. □

Chapter 8

Case Studies

Equipped with Termination$_{\mathsf{SHL}}$ and Refinement$_{\mathsf{SHL}}$, let us now consider several case studies that we can verify with them. We consider a termination-preserving refinement for the memo_rec combinator from §5 (§8.1), termination for a reentrant event loop (§8.2), and the logical relation for termination of asynchronous channels of Spies, Krishnaswami, and Dreyer[1] (§8.3).

[1] Spies, Krishnaswami, and Dreyer, "Transfinite step-indexing for termination", 2021 [SKD21].

8.1 Recursive Memoization

Recall that when memo_rec is used to memoize a recursive function, the results of recursive calls are cached in a lookup table and reused (instead of recomputed). In SHL, we define memo_rec as:

```
memo_rec ≜ λt. let tbl = map() in
          fix g x. match get tbl x with
               | None ⇒  let y = t g x in set tbl x y; y
               | Some y ⇒  y
               end
```

Here, t is the *template* of the function that we want to memoize. (Recall from §5 that the template describes the function body e in $\mathsf{fix}\ f\ x.\ e$ as a function $t : (\tau \to \sigma) \to (\tau \to \sigma)$.) The function map creates a mutable lookup table, get retrieves an element from the table, and set stores an element in a table.[2]

[2] To support element lookup, the map operations need an equality comparison operation on the keys. In the Rocq implementation, this is passed around as an additional function eq, which becomes a parameter of the memo_rec function. For simplicity, we elide it here. Moreover, we use an association list as a lookup table instead of a hash table (used in §5), because whether the keys are hashed or not is orthogonal to the example.

We have proved a generic and modular specifications for memo_rec, which we will discuss at the end of §8.1.2. Since it is somewhat involved, we first consider the memoization of pure functions as a stepping stone in §8.1.1 and, to keep matters concrete, we consider two examples: memoizing the Fibonacci function and the Levenshtein distance[3] (templates depicted in Fig. 8.1).

[3] Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals", 1965 [Lev65].

8.1.1 Pure Templates

We start by considering the memoization of *pure* templates for functions of type $\mathbb{N} \to \mathbb{N}$, such as the Fib template for Fibonacci, shown in Fig. 8.1. Given a template t, we define the standard recursive version of the function as $r_t \triangleq \mathsf{fix}\ g\ n.\ t\ g\ n$ and the memoized version as $m_t \triangleq \mathsf{memo\_rec}\ t$.

We wish to show that m_t refines r_t, in the sense that for all natural numbers n, we have $m_t\ n \preceq_{\mathbb{N}} r_t\ n$. Thus, for memoizing pure functions, we prove:[4]

[4] We can get a simpler specification here than the one we provide for stateful templates (memo-rec) by leveraging the assumption that r_t is pure. We have also proven this specification in the Rocq development of Transfinite Iris [Spi+21a].

pure-memo-rec

$$\frac{\forall g.\ \triangleright(\forall n.\ g\ n \preceq_{\mathbb{N}} r_t\ n) \Rightarrow \forall n.\ t\ g\ n \preceq_{\mathbb{N}} r_t\ n \qquad r_t \text{ is pure}}{\forall n.\ m_t\ n \preceq_{\mathbb{N}} r_t\ n}$$

```
$\texttt{Fib}\ \mathit{fib}\ n \triangleq \text{if } n < 2 \text{ then } n \text{ else } \mathit{fib}(n-1) + \mathit{fib}(n-2)$
$\texttt{Slen}\ \mathit{slen}\ s \triangleq \text{if } !s == 0 \text{ then } 0 \text{ else } \mathit{slen}(s+1) + 1$
$\texttt{Lev}\ \mathit{slen}\ \mathit{lev}\ (s, t) \triangleq$
  $\text{if } !s == 0 \text{ then } \mathit{slen}\ t \text{ else}$
  $\text{if } !t == 0 \text{ then } \mathit{slen}\ s \text{ else}$
  $\text{if } !s == !t \text{ then } \mathit{lev}(s+1, t+1) \text{ else}$
  $1 + \texttt{min}(\mathit{lev}(s, t+1), \mathit{lev}(s+1, s), \mathit{lev}(s+1, t+1))$
```

Figure 8.1: Recursive templates of memoized functions.

The premise constrains the template t: for any function g that implements r_t, the template applied to g should also implement r_t. The assumption about g is guarded by a later modality ($\triangleright$) such that a proof of the premise must execute a step of the source $r_t\ n$ before this assumption can be used. Typically, this is trivial since $r_t\ n \rightarrow_{\text{pure}} t\ r_t\ n$, and for safe, pure templates like `Fib`, the residual goal ($t\ g\ n \preceq_{\mathbb{N}} t\ r_t\ n$) is provable by a simple lock-step simulation.

Proof Sketch for pure-memo-rec. We first derive standard Hoare triples for the operations of the lookup table. Then, we proceed with the verification of `memo_rec`. After allocating a lookup table *tbl*, the expression m_t reduces to:

```
$h \triangleq \text{fix } g\ x.\ \text{match } \texttt{get}\ \mathit{tbl}\ x \text{ with}$
              $\mid \text{None} \Rightarrow\ \text{let } y = t\ g\ x \text{ in } \texttt{set}\ \mathit{tbl}\ x\ y;\ y$
              $\mid \text{Some } y \Rightarrow\ y$
              $\text{end}$
```

We must now prove $\forall n.\ h\ n \preceq_{\mathbb{N}} r_t\ n$. We proceed by using Löb induction in a way similar to Lemma 44. The induction hypothesis from Löb is

$$\triangleright(\forall n.\ h\ n \preceq_{\mathbb{N}} r_t\ n), \quad \textit{and we show} \quad \forall n.\ h\ n \preceq_{\mathbb{N}} r_t\ n.$$

Given an argument n, we do a case analysis on whether the result has already been stored in the lookup table or not:

1. In case it has not, `memo_rec` calls $t\ h\ n$. Applying the premise of pure-memo-rec, $\forall g.\ \triangleright(\forall n.\ g\ n \preceq_{\mathbb{N}} r_t\ n) \Rightarrow \forall n.\ t\ g\ n \preceq_{\mathbb{N}} r_t\ n$, to the induction hypothesis, we know $t\ h\ n \preceq_{\mathbb{N}} r_t\ n$. The resulting value of $t\ h\ n$ and $r_t\ n$, say m, is then stored in the lookup table. We use an invariant assertion[5] (§3.4) to ensure that all values in the lookup table are the result of running r_t on the associated keys. Concretely, since r_t is pure, we would remember $r_t\ n \rightarrow^{+}_{\text{pure}} m$ for key n in the invariant.

2. In case n is found in the lookup table, we must argue that the stored value m is equal to the result of $r_t\ n$. However, since we remember $r_t\ n \rightarrow^{+}_{\text{pure}} m$ when the result is computed for the first time, we can justify a source execution of $r_t\ n$ to m using the stuttering rule pure-src-stutter-tpr (repeatedly). □

[5] As mentioned in §7.3.1, we use *non-atomic* invariants in Transfinite Iris (which can be opened for multiple steps) for reasoning about sequential code. We gloss over the distinction between non-atomic and regular invariants here. For the full details, see the Transfinite Iris Rocq development [Spi+21a].

We can then instantiate pure-memo-rec to obtain the following corollary:

Corollary 52. $\forall n.\ m_{\texttt{Fib}}\ n \preceq_{\mathbb{N}} r_{\texttt{Fib}}\ n$

8.1.2 Stateful Templates

Memoization can also be applied to templates that use state, so long as they execute in a *repeatable* fashion. That is, when the function is run multiple times, it must return the same value. This ensures that it is correct to reuse the stored values in the lookup table during memoization. Our specification for memoization of stateful templates replaces the purity side condition in PURE-MEMO-REC with an encoding of this repeatability condition.

To see an example of where our specification is useful, consider the Levenshtein function template in Fig. 8.1, which computes the edit distance between two strings. We parameterize the template by a function *slen* used for computing the length of strings. The input strings are stored on the heap as null-terminated arrays, as in languages like C.[6] Because the length of the same substring is computed multiple times, the Levenshtein function can be optimized further by also memoizing the string length function, using the template `Slen` from Fig. 8.1. Thus, to memoize the Levenshtein function, we define:

$$\texttt{mlev} \triangleq \mathsf{let}\ \mathit{mslen} = \texttt{memo\_rec}\ \texttt{Slen}\ \mathsf{in}\ \texttt{memo\_rec}\ (\texttt{Lev}\ \mathit{mslen})$$

[6] To represent the strings on the heap, we use a feature of HeapLang that we have not discussed yet: arrays and address arithmetic. That is, instead of a single reference cell, one can also allocate an array of values on the heap in (sequential) HeapLang. The different fields in the array are then accessed via address arithmetic (*e.g.*, for the location ℓ, the location of the next entry is $\ell + 1$).

The resulting function performs nested memoization: it memoizes the length function `slen` $\triangleq \mathsf{fix}\ \mathsf{s}\ x.\ \texttt{Slen}\ \mathsf{s}\ x$ inside of the template `Lev`.

This use of the `Lev` template is not pure: it not only reads from heap allocated state (the string), but also accesses and modifies the internally allocated memo table used in `mslen`. Nevertheless, the template is repeatable, so long as the input strings are not modified after memoizing.

The `memo_rec` specification. Let us now turn to the specification. To define it, we introduce a generalization of the simulation $e_t \preceq^{\mathsf{tpr}}_{G} e_s$ to two functions g and f, where the executions of f *must be repeatable*. Concretely, we will now define a relation $g \preceq^{\mathsf{repeat}}_{P,Q} f$, where source executions of f will be repeatable, P is a persistent (see §3.3) precondition (*i.e.,* $\forall v, v'.\ P\, v\, v' \vdash \square\, P\, v\, v'$) for the arguments of g and f, and Q is a persistent postcondition (*i.e.,* $\forall w, w'.\ Q\, w\, w' \vdash \square\, Q\, w\, w'$) for the resulting values of g and f. (The postcondition Q replaces the role of the set G in the simulation $e_t \preceq^{\mathsf{tpr}}_{G} e_s$.) We define the simulation as:

$$\begin{aligned}
&g \preceq^{\mathsf{repeat}}_{P,Q} f \triangleq \forall v, v', K, n.\\
&\quad \{P(v, v') * \mathsf{src}(K[f\, v'])\}\\
&\qquad g\, v\\
&\quad \{w.\ \exists w'.\ Q(w, w') * \$_{\mathsf{st}}\, n * \mathsf{src}(K[w'] * \mathsf{repeat}(f, v, w))\}_{\mathsf{tpr}}
\end{aligned}$$

We start out with two values v and v' related by the precondition P, and the current source expression is $f\, v'$ (in some evaluation context K). If we now execute $g\, v$ in the target, then the result is some value w such that the source executes to some value w' and the two are related by the postcondition Q'. Moreover, the postcondition provides an arbitrary amount of stutter credits $\$_{\mathsf{st}}\, n$ to whoever executes this simulation. (Recall the stutter credits can be obtained by executing source steps; see Fig. 7.4.) In addition to the postcondition Q, we obtain the knowledge that we can *repeat this execution in the source* $\mathsf{repeat}_{P,Q}(f, v, w)$ (defined below). This fact—in place of $r_t\, n \rightarrow^{+}_{\mathsf{pure}} m$ in PURE-MEMO-REC—is what allows us to stutter the target when a lookup succeeds, and we execute f again in the source.

We define the notion of a repeatable source execution as follows:

$$\mathsf{repeat}_{P,Q}(f, v, w) \triangleq \Box(\forall v''.\, P(v, v'') \twoheadrightarrow \exists w''.\ \Box\,\mathsf{exec}(f\,v'', w'') * Q(w, w''))$$
$$\mathsf{exec}(e, u) \triangleq \forall K.\, \mathsf{src}(K[e]) \twoheadrightarrow \Rrightarrow_{\mathsf{src}} \mathsf{src}(K[u])$$

The predicate $\mathsf{exec}(e, u)$ effectively says that e can be executed to u on the source side (in an arbitrary evaluation context K). The predicate $\mathsf{repeat}_{P,Q}(f, v, w)$ says that for an argument-result pair v and w *of the target* (*i.e.,* g in the definition of $g \preceq^{\mathsf{repeat}}_{P,Q} f$), we can repeatably execute the function f in the source.[7] More specifically, if we start with an argument v'' that is related to v by the precondition P (*e.g.,* for `mslen`, an immutable string the with same contents as v), then we can execute f in the source (*e.g.,* the recursive definition of the string length $\mathsf{fix\,slen}\,x.\,\mathtt{Slen}\,\mathsf{slen}\,x$) and we obtain a value w'' that is related to the target result w via the postcondition Q (*e.g.,* the resulting integers are the same). The persistency modalities "$\Box$" in this definition ensure that we can execute f multiple times, and the source update "$\Rrightarrow_{\mathsf{src}}$" ensures that we can stutter the target while we execute f in the source.

[7] We will see a similar construction in Part III (see §13.1.2). There it is not used to express repeatability but out-of-order execution of the source program. In particular, the condition on the source program there is not persistent, so it cannot be repeated arbitrarily often.

Given the notion of a repeatable source execution, we can now state the generic specification for `memo_rec`:

memo-rec
$$\frac{\forall g.\ \{\triangleright(g \preceq^{\mathsf{repeat}}_{P,Q} f)\}\ t\ g\ \{h.\ h \preceq^{\mathsf{repeat}}_{P,Q} f\}_{\mathsf{tpr}}}{\{\$_{\mathsf{st}}\ 1\}\ \mathtt{memo\_rec}\ t\ \{h.\ h \preceq^{\mathsf{repeat}}_{P,Q} f\}_{\mathsf{tpr}}}$$

It assumes that t is a template for f in the sense that for any g that implements f, it returns a function h that implements f (analogous to the assumption $\triangleright(\forall n.\, g\ n \preceq_{\mathbb{N}} r_t\ n) \Rightarrow \forall n.\, t\ g\ n \preceq_{\mathbb{N}} r_t\ n$ in pure-memo-rec). It then ensures that `memo_rec` returns a function h that itself implements f.

We can use this specification to derive a result about memoizing the Levenshtein function. Concretely, let us define $\mathtt{lev} \triangleq \mathsf{fix}\,\mathtt{l}\,x.\,\mathtt{Lev}\ \mathtt{slen}\ \mathtt{l}\ x$. Then, we can derive the following corollary:

Corollary 53. *Let* $p \triangleq \lambda x.\, \mathtt{mlev}\,x$ *and* $q \triangleq \lambda x.\, \mathtt{lev}\,x$*. Then* $p \preceq^{\mathsf{repeat}}_{(\mathbb{S}\times\mathbb{S})_{\mathsf{eq}},\mathbb{N}_{\mathsf{eq}}} q$*, where* $(\mathbb{S}\times\mathbb{S})_{\mathsf{eq}}$ *expresses equality over pairs of immutable heap-allocated strings and* $\mathbb{N}_{\mathsf{eq}}$ *equality over natural numbers.*

A note on stuttering. Note that besides assuming the condition on the template t, the specification memo-rec also requires a stuttering credit $\$_{\mathsf{st}}\ 1$. It—together with the credits returned in the postcondition of $(\preceq^{\mathsf{repeat}}_{P,Q})$—are used to interact with an impredicative invariant that we set up for `memo_rec` to store the repeatable source executions (*i.e.,* to store $\mathsf{repeat}_{P,Q}(f, v, w)$ for key-value pairs v and w). The impredicative invariant induces laters (see §3.4.1), and the credits are used to stutter the source and eliminate the laters. (In the case of the pure specification, the credits are not necessary, because the invariant is timeless.) In Part III of this dissertation, we will discuss invariants and step-indexing at length, so we do not go into further details here.

The stutter credits, however, are not hard to come by, since all that is needed is a step in the source (see the rules in Fig. 7.4). For example, in Corollary 53, we wrap the original implementation of the Levenshtein function `lev` and the memoized version `mlev` in a λ-abstraction. This wrapping suffices to execute one step in the target and in the source, which allows one to allocate enough stutter credits for the memoization.

8.2 A Reentrant Event Loop

Let us now turn to our first application of $\text{Termination}_{\text{SHL}}$. We illustrate how transfinite time credits interact with other features of step-indexing using a reentrant event loop as an example. In this example, an event loop consists of a stack of functions q, which can be extended with `addtask` $q\ f$, and can be executed through `run` q:

```
       mkloop() ≜ stack()
addtask q f ≜ push q f
          run q ≜ match pop q with
                  | None ⇒ ()
                  | Some f ⇒ f (); run q
                  end
```

Importantly, the event loop is *reentrant*: a function f that is added can extend the event loop with new functions when executed. As such, proving termination of `run` is subtle: there is no intrinsic termination measure, since the size of the stack q can increase before it eventually decreases. In fact, if one is not careful, it could even never terminate (*e.g.,* if each task adds another task to the queue).

We can prove that the event loop eventually terminates *if* it is used according to the following specifications. This is because we require spending credits for enqueueing tasks, and there are only a transfinite number of credits available.

$$\{\mathsf{True}\}\ \texttt{mkloop}()\ \{q.\ \mathsf{eloop}(q)\}_{\mathsf{term}} \qquad \mathsf{persistent}(\mathsf{eloop}(q))$$

$$\{\$2 * \mathsf{eloop}(q) * \{P\}\ f\ ()\ \{\_.\ \mathsf{True}\}_{\mathsf{term}} * P\}\ \texttt{addtask}\ q\ f\ \{\_.\ \mathsf{True}\}_{\mathsf{term}}$$

$$\{\$1 * \mathsf{eloop}(q)\}\ \texttt{run}\ q\ \{\_.\ \mathsf{True}\}_{\mathsf{term}}$$

We can create a new event loop with `mkloop`; the representation predicate that we obtain $\mathsf{eloop}(q)$ is persistent, such that it can be shared freely afterwards. We can run the event loop with `run`, and we can add a task f with `addtask`. When we want to add a task f to the event loop, we must prove a Hoare triple $\{P\}\ f\ ()\ \{\_.\ \mathsf{True}\}_{\mathsf{term}}$ for it and ensure the precondition P (*e.g.,* containing additional time credits) currently holds. The specification for `addtask` can still be used *after* the event loop has been started with `run` (*e.g.,* from one of the functions enqueued in the event loop). To ensure termination, the precondition for `addtask` consumes a constant $c = 2$ credits,[8] which are logically transferred to the event loop stack and, additionally, requires the cost of the task to be paid upfront via the precondition P.

[8] There are two credits, because one is used to open an invariant, and the other one is used to justify the recursion inside the `run` function.

To verify the `run` specification (and hence prove termination of `run`), we exploit the step-indexing underlying Transfinite Iris. Concretely, we use Löb induction, since it does not require an intrinsic termination measure. By using Löb induction, we obtain an assumption justifying the termination of a recursive execution of `run` guarded by a later modality ($\triangleright$). This later is then removed when using spend-cred-term (in Fig. 7.5), which requires spending a time credit. Here, we spend the time credits that are deposited by the calls to `addtask`. The intuition is that even though extra jobs may be added while `run` executes, only a finite number can be added in each run, because the total number of credits available is an ordinal.

As an aside, note that even though each operation only uses a finite amount of credits, transfinite credits are still useful here if, *e.g.,* the amount of scheduled tasks depends on dynamic information (as in the example in §7.2).

8.3 A Logical Relation for Asynchronous Channels

As a larger application of $\text{Termination}_{\text{SHL}}$, transfinite time credits allow us to obtain and mechanize the main result of Spies, Krishnaswami, and Dreyer:[9] termination of a linear language with asynchronous channels. For their termination proof, they introduce a transfinitely step-indexed logical relation which, internally, uses a bespoke form of transfinite time credits and transfinite step-indexing. In $\text{Termination}_{\text{SHL}}$ we can, using our general form of transfinite time credits, simplify their logical relation (which they define and use in a 40-page appendix) and mechanize their result in 500 lines of Rocq. With an additional 350 lines of Rocq, we generalize their result with impredicative polymorphism.

[9] Spies, Krishnaswami, and Dreyer, "Transfinite step-indexing for termination", 2021 [SKD21].

Impredicative invariants for logical relations. Our generalization to impredicative polymorphism relies crucially on *impredicative invariants* (§3.4.1). To explain in a bit more detail how we leverage impredicative invariants and, moreover, why it would be non-trivial to generalize the logical relation of Spies, Krishnaswami, and Dreyer without using Transfinite Iris, let us briefly review how logical relations are typically formalized in Iris.[10] In an Iris logical relation, a type τ is interpreted as an Iris predicate (*i.e.,* an element of type $Val \to iProp$). For example, the ML-style reference type $\mathsf{ref}(\tau)$ can be interpreted as a predicate on memory locations ℓ that asserts that there is an *invariant* on ℓ, which in turn asserts that the value v stored at ℓ must satisfy the predicate associated with τ. The key point then is that, because of polymorphic types, τ could for instance be a type variable, modeled as an *arbitrary* unknown Iris predicate, and thus it is crucial that the invariant in the model of $\mathsf{ref}(\tau)$ is impredicative.

[10] We will see another example of a logical relation formalized in Iris, a binary logical relation, in Part III.

To the best of our knowledge, there is no way to avoid impredicative invariants here. This is related to the well-known "type-world circularity" that arises in logical relations for languages with polymorphism and higher-order state (see *e.g.,* Ahmed's earlier work on concrete step-indexed models of such type systems[11]). Hence, if one tries to generalize the logical relation of Spies, Krishnaswami, and Dreyer without using Transfinite Iris, one has to solve the type-world circularity "by hand," meaning one essentially has to solve the kind of recursive domain equation underlying Transfinite Iris (see §4.5 for the Iris version and §9.3 for the Transfinite Iris version). Here, we instead leverage that Transfinite Iris has impredicative invariants to obtain a simple generalization of the logical relation with polymorphism.

[11] Ahmed, "Semantics of types for mutable state", 2004 [Ahm04].

Below, we explain how to encode the language λ_{CHAN} of Spies, Krishnaswami, and Dreyer in SHL (§8.3.1), we simplify their logical relation (§8.3.2), and then we extend it to handle impredicative polymorphism (§8.3.3).

8.3.1 Language

The language λ_{CHAN} is a linear λ-calculus with Booleans, natural numbers, and pairs extended with asynchronous channels. Besides constructs which are already included in SHL (*e.g.,* Booleans, pairs, and natural numbers), λ_{CHAN} features (1) a primitive operation iter for iteration on natural numbers and (2) primitives on asynchronous channels. For iteration, we define $\mathtt{iter}(e, e_0, x.\, e_S) \triangleq \mathtt{it}\, e_0\, e\, (\lambda x.\, e_S)$ where the body of the iteration is implemented as the recursive function $\mathtt{it} \triangleq \mathsf{fix}\, \mathtt{it}\, s.\ \lambda n, f.\, \mathsf{if}\ n == 0\ \mathsf{then}\ s\ \mathsf{else}\ \mathtt{it}\, (f\ s)\, (n-1)\, f$. Below we explain how to encode the asynchronous channels.

Channels. The asynchronous channels offer three operations: chan to create a new channel, which results in pair of a "send-handle" and a "receive-handle"; $\mathtt{put}$ to send a value over a channel; and $\mathtt{get}$ to receive a value over a channel. Channels can be in one of three states: empty, containing a value (waiting for a $\mathtt{get}$), or containing a continuation (waiting for a $\mathtt{put}$). We encode these states with three constructors E, $\mathsf{V}(v)$, and $\mathsf{C}(f)$, and write

$$\mathsf{case}\ e\ \mathsf{of}\ \mathsf{E} \Rightarrow e_1 \mid \mathsf{V}(v) \Rightarrow e_2 \mid \mathsf{C}(f) \Rightarrow e_3$$

for case analysis on them. (We can derive the constructors in SHL with nested sums and the case analysis construct using match expressions.) Channels are then implemented as heap locations that store one of these constructors.

We encode the three essential channel operations as follows:

$$\begin{aligned}
\mathsf{chan} &\triangleq \mathsf{let}\ c := \mathsf{ref}(\mathsf{E})\ \mathsf{in}\ (c, c)\\
\mathtt{get} &\triangleq \lambda(c, f).\ \mathsf{case}\ !c\ \mathsf{of}\ \mathsf{E} \Rightarrow c := \mathsf{C}(f) \mid \mathsf{V}(v) \Rightarrow c := \mathsf{E}; f(v) \mid \mathsf{C}(f) \Rightarrow \mathtt{div}()\\
\mathtt{put} &\triangleq \lambda(c, v).\ \mathsf{case}\ !c\ \mathsf{of}\ \mathsf{E} \Rightarrow c := \mathsf{V}(v) \mid \mathsf{V}(v) \Rightarrow \mathtt{div}() \mid \mathsf{C}(f) \Rightarrow c := \mathsf{E}; f(v)\\
\mathtt{div} &\triangleq \mathsf{fix}\ f\ x.\ f(x)
\end{aligned}$$

The operation chan creates an empty channel. The operation $\mathtt{get}$ either stores the continuation (if the channel is still empty), or it empties the channel and executes the continuation with the current value. The operation $\mathtt{put}$ either stores the value (if the channel is still empty), or it empties the channel and executes the continuation with the current value.

Since the channels are used linearly, we can never be in the situation that a $\mathtt{get}$ encounters a continuation in the channel (or dually a $\mathtt{put}$ encounters a value). Thus, in λ_{CHAN}, there is no case for C in $\mathtt{get}$ and V in $\mathtt{put}$. Here, being interested in safe termination, we have replaced these cases with divergence, ensuring that they are never executed if we can prove termination.

Type system. In its original (monomorphic) formulation, λ_{CHAN} is equipped with the linear type system depicted in Fig. 8.2. We write Γ_1, Γ_2 for the disjoint union of the linear typing contexts Γ_1 and Γ_2.

8.3.2 Simplified Logical Relation

Let us now turn to how we define a logical relation for λ_{CHAN} in Transfinite Iris. Instead of first defining the type formers (*e.g.*, $\mathbb{1}$, $\mathbb{N}$, $T \otimes U$, *etc.*) syntactically and then interpreting them semantically, we directly define their semantic interpretations in the form of semantic types and semantic type formers. In our case, the semantic types T, U : SemType are predicates from values to propositions, meaning SemType $\triangleq$ *Val* $\longrightarrow$ *iProp*.[12] We define

[12] Unlike in most logical relations in Iris, the semantic interpretation of types is linear (or rather affine) here. However, for the channels, we still need sharing of the underlying memory, which is why we use an impredicative invariant here.

$$\begin{aligned}
\mathbb{1}_{\mathrm{sem}} &\triangleq \{v \mid v = ()\}\\
\mathbb{B}_{\mathrm{sem}} &\triangleq \{v \mid \exists b : \mathbb{B}.\ v = b\}\\
\mathbb{N}_{\mathrm{sem}} &\triangleq \{v \mid \exists n : \mathbb{N}.\ v = n\}\\
T \multimap U &\triangleq \left\{f \mid \forall v.\ v \in T \twoheadrightarrow \mathbf{wp}\ f\ v\ \{w.\ w \in U\}\right\}\\
T \otimes U &\triangleq \{v \mid \exists v_1, v_2.\ v = (v_1, v_2) * v_1 \in T * v_2 \in U\}\\
\mathrm{Get}(T) &\triangleq \left\{v \mid \exists \ell, \gamma_{\mathrm{get}}, \gamma_{\mathrm{put}}.\ \$1 * v = \ell * \boxed{\mathsf{I}(\gamma_{\mathrm{get}}, \gamma_{\mathrm{put}}, \ell, T)}^{\mathcal{N}.\ell} * \boxed{\mathsf{ex}()}^{\gamma_{\mathrm{get}}}\right\}\\
\mathrm{Put}(T) &\triangleq \left\{v \mid \exists \ell, \gamma_{\mathrm{get}}, \gamma_{\mathrm{put}}.\ \$1 * v = \ell * \boxed{\mathsf{I}(\gamma_{\mathrm{get}}, \gamma_{\mathrm{put}}, \ell, T)}^{\mathcal{N}.\ell} * \boxed{\mathsf{ex}()}^{\gamma_{\mathrm{put}}}\right\}
\end{aligned}$$

$$x : A \vdash x : A \qquad \frac{\Gamma \vdash e : B}{\Gamma, x : A \vdash e : B} \qquad \emptyset \vdash () : \mathbb{1} \qquad \frac{\Gamma_1 \vdash e_1 : \mathbb{1} \qquad \Gamma_2 \vdash e_2 : A}{\Gamma_1, \Gamma_2 \vdash e_1; e_2 : A} \qquad \Gamma \vdash b : \mathbb{B}$$

$$\frac{\Gamma_1 \vdash e : \mathbb{B} \qquad \Gamma_2 \vdash e_1 : A \qquad \Gamma_2 \vdash e_2 : A}{\Gamma_1, \Gamma_2 \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : A} \qquad \emptyset \vdash n : \mathbb{N} \qquad \frac{\Gamma_1 \vdash e_1 : \mathbb{N} \qquad \Gamma_2 \vdash e_2 : \mathbb{N}}{\Gamma_1, \Gamma_2 \vdash e_1 + e_2 : \mathbb{N}}$$

$$\frac{\Gamma_1 \vdash e : \mathbb{N} \qquad \Gamma_2 \vdash e_0 : A \qquad x : A \vdash e_S : A}{\Gamma_1, \Gamma_2 \vdash \mathtt{iter}(e_0, e, x.\, e_S) \vdash A} \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.\, e : A \multimap B} \qquad \frac{\Gamma_1 \vdash e_1 : A \multimap B \qquad \Gamma_2 \vdash e_2 : A}{\Gamma_1, \Gamma_2 \vdash e_1\, e_2 : B}$$

$$\frac{\Gamma_1 \vdash e_1 : A \qquad \Gamma_2 \vdash e_2 : B}{\Gamma_1, \Gamma_2 \vdash (e_1, e_2) : A \otimes B} \qquad \frac{\Gamma_1 \vdash e_1 : A_1 \otimes A_2 \qquad \Gamma_1, x : A_1, y : A_2 \vdash e_2 : B}{\Gamma_1, \Gamma_2 \vdash \mathsf{let}\ (x, y) = e_1\ \mathsf{in}\ e_2 : B} \qquad \emptyset \vdash \mathsf{chan}() : \mathsf{Get}(A) \otimes \mathsf{Put}(A)$$

$$\frac{\Gamma_1 \vdash e_1 : \mathsf{Get}(A) \qquad \Gamma_2 \vdash e_2 : A \multimap \mathbb{1}}{\Gamma_1, \Gamma_2 \vdash \mathtt{get}(e_1, e_2) : \mathbb{1}} \qquad \frac{\Gamma_1 \vdash e_1 : \mathsf{Put}(A) \qquad \Gamma_2 \vdash e_2 : A}{\Gamma_1, \Gamma_2 \vdash \mathtt{put}(e_1, e_2) : \mathbb{1}}$$

Figure 8.2: The (monomorphic) type system of λ_{CHAN}

The channel interpretation. For the interpretation of the $\mathsf{Get}(\cdot)$ and $\mathsf{Put}(\cdot)$ types, we maintain an impredicative invariant saying that the channel location points to a valid channel state.[13] We associate every channel with two ghost names γ_{get} and γ_{put}. Their ghost state—of the resource algebra $Ex(\mathsf{unit})$ from §4.2.1—$\boxed{\mathsf{ex}()}^{\gamma_{\mathsf{get}}}$ and $\boxed{\mathsf{ex}()}^{\gamma_{\mathsf{put}}}$ encodes the permission to use the get and put operation, respectively. The exclusive resource algebra and the definition of the invariant ensure that, for instance, when we own $\boxed{\mathsf{ex}()}^{\gamma_{\mathsf{get}}}$, the invariant cannot be in the $\mathsf{C}(f)$ case.

[13] As mentioned in §7.3.1, we use *non-atomic* invariants in Transfinite Iris for reasoning about sequential code. We gloss over the distinction between non-atomic and regular invariants here. For the full details, see the Transfinite Iris Rocq development [Spi+21a].

$$\begin{aligned}\mathsf{I}(\gamma_{\mathsf{get}}, \gamma_{\mathsf{put}}, \ell, T) \triangleq\ & \ell \mapsto \mathsf{E} \\ & \vee (\exists v.\, \ell \mapsto \mathsf{V}(v) * v \in T * \boxed{\mathsf{ex}()}^{\gamma_{\mathsf{put}}}) \\ & \vee (\exists f.\, \ell \mapsto \mathsf{C}(f) * f \in (T \multimap \mathbb{1}_{\mathsf{sem}}) * \boxed{\mathsf{ex}()}^{\gamma_{\mathsf{get}}})\end{aligned}$$

Moreover, the interpretations of $\mathsf{Get}(\cdot)$ and $\mathsf{Put}(\cdot)$ include one time credit which is used (with the rule SPEND-CRED-TERM) to remove the later that arises when opening the invariant.

Semantic typing. We can now define the semantic typing relation. For the semantic typing relation, we use as contexts Γ finite maps from variable names to semantic types SemType:[14]

[14] Here, $\gamma(e)$ denotes the result of substituting the free variables in e according to the substitution γ.

$$\begin{aligned}\mathcal{G}[\![\Gamma]\!] &\triangleq \{\gamma \mid \ast_{x:T \in \Gamma}\, \gamma(x) \text{ is defined} * \gamma(x) \in T\} \\ \Gamma \vDash e : T &\triangleq\ \vdash \exists \alpha.\, \$\alpha \twoheadrightarrow \forall \gamma.\, \gamma \in \mathcal{G}[\![\Gamma]\!] \twoheadrightarrow \mathbf{wp}\ \gamma(e)\ \{v.\, v \in T\}\end{aligned}$$

The only non-standard aspect about our semantic typing is that we existentially quantify over the number of initial time credits. This works, because for every program a sufficient ordinal number of time credits can be (inductively) computed. In short, channel creation requires two time credits—one for the send handle and one for the receive handle—and iteration can be overapproximated by ω times the credits required for the body of the iteration.

Soundness. We use the traditional approach to proving soundness of a logical relation. We show that in each of the rules, we can replace the syntactic notion ($\vdash$) with the semantic notion ($\vDash$). Concretely, we prove the rules in Fig. 8.3.

In addition, we use the logical relation and Theorem 51 to derive:

Lemma 54. *If $\emptyset \vDash e : T$, then e terminates safely.*

$$x : T \vDash x : T \qquad \frac{\Gamma \vDash e : U}{\Gamma, x : T \vDash e : U} \qquad \emptyset \vDash () : \mathbb{1}_{\text{sem}} \qquad \frac{\Gamma_1 \vDash e_1 : \mathbb{1}_{\text{sem}} \qquad \Gamma_2 \vDash e_2 : T}{\Gamma_1, \Gamma_2 \vDash e_1; e_2 : T} \qquad \Gamma \vDash b : \mathbb{B}_{\text{sem}}$$

$$\frac{\Gamma_1 \vDash e : \mathbb{B}_{\text{sem}} \qquad \Gamma_2 \vDash e_1 : T \qquad \Gamma_2 \vDash e_2 : T}{\Gamma_1, \Gamma_2 \vDash \text{if } e \text{ then } e_1 \text{ else } e_2 : T} \qquad \emptyset \vDash n : \mathbb{N}_{\text{sem}} \qquad \frac{\Gamma_1 \vDash e_1 : \mathbb{N}_{\text{sem}} \qquad \Gamma_2 \vDash e_2 : \mathbb{N}_{\text{sem}}}{\Gamma_1, \Gamma_2 \vDash e_1 + e_2 : \mathbb{N}_{\text{sem}}}$$

$$\frac{\Gamma_1 \vDash e : \mathbb{N}_{\text{sem}} \qquad \Gamma_2 \vDash e_0 : T \qquad x : T \vDash e_S : T}{\Gamma_1, \Gamma_2 \vDash \mathtt{iter}(e_0, e, x.\, e_S) : T} \qquad \frac{\Gamma, x : T \vDash e : U}{\Gamma \vDash \lambda x.\, e : T \multimap U} \qquad \frac{\Gamma_1 \vDash e_1 : T \multimap U \qquad \Gamma_2 \vDash e_2 : T}{\Gamma_1, \Gamma_2 \vDash e_1\, e_2 : U}$$

$$\frac{\Gamma_1 \vDash e_1 : T \qquad \Gamma_2 \vDash e_2 : U}{\Gamma_1, \Gamma_2 \vDash (e_1, e_2) : T \otimes U} \qquad \frac{\Gamma_1 \vDash e_1 : T_1 \otimes T_2 \qquad \Gamma_1, x : T_1, y : T_2 \vDash e_2 : U}{\Gamma_1, \Gamma_2 \vDash \text{let } (x, y) = e_1 \text{ in } e_2 : U} \qquad \emptyset \vDash \mathsf{chan}() : \mathsf{Get}(T) \otimes \mathsf{Put}(T)$$

$$\frac{\Gamma_1 \vDash e_1 : \mathsf{Get}(T) \qquad \Gamma_2 \vDash e_2 : T \multimap \mathbb{1}_{\text{sem}}}{\Gamma_1, \Gamma_2 \vDash \mathtt{get}(e_1, e_2) : \mathbb{1}_{\text{sem}}} \qquad \frac{\Gamma_1 \vDash e_1 : \mathsf{Put}(T) \qquad \Gamma_2 \vDash e_2 : T}{\Gamma_1, \Gamma_2 \vDash \mathtt{put}(e_1, e_2) : \mathbb{1}_{\text{sem}}}$$

Figure 8.3: The compatibility lemmas for the monomorphic logical relation.

8.3.3 Adding Impredicative Polymorphism

To add polymorphism to the language, we embed the usual introduction and elimination primitives as derived forms of the term language:

$$\Lambda.e \triangleq \lambda().\, e \qquad e\langle\rangle \triangleq e() \qquad \text{unpack } e \text{ as } x \text{ in } e' \triangleq (\lambda x.\, e')e \qquad \text{pack } e \triangleq e$$

This brings us to our extension of the linear type system of λ_{CHAN} with polymorphism. The type system is given in Fig. 8.4. We use types A (possibly) containing type variables X, a linear typing context Γ over these types, and now additionally a type variable context Δ. We define the well-formedness judgments:

$$\frac{A \text{ is closed under } \Delta}{\Delta \vdash A} \qquad \frac{\forall x : A \in \Gamma. \Delta \vdash A}{\Delta \vdash \Gamma}$$

Semantic Interpetation. With impredicative polymorphism, our types can now depend on type variables. We reflect this in the semantic interpretation by additionally parameterizing the semantic type formers with (maps of) semantic types for the type variables. That is, we consider polymorphic semantic types $\mathbf{T}, \mathbf{U} : \mathsf{PolySemType}$ where $\mathsf{PolySemType} \triangleq (\mathsf{id} \xrightarrow{\text{fin}} \mathsf{SemType}) \to \mathsf{SemType}$ (*i.e.,* predicates from finite maps of semantic types to semantic types). We lift

$$\Delta; x : A \vdash x : A \qquad \frac{\Delta; \Gamma \vdash e : B}{\Delta; \Gamma, x : A \vdash e : B} \qquad \Delta; \emptyset \vdash () : \mathbb{1} \qquad \frac{\Delta; \Gamma_1 \vdash e_1 : \mathbb{1} \qquad \Delta; \Gamma_2 \vdash e_2 : A}{\Delta; \Gamma_1, \Gamma_2 \vdash e_1; e_2 : A} \qquad \Delta; \Gamma \vdash b : \mathbb{B}$$

$$\frac{\Delta; \Gamma_1 \vdash e : \mathbb{B} \qquad \Delta; \Gamma_2 \vdash e_1 : A \qquad \Delta; \Gamma_2 \vdash e_2 : A}{\Delta; \Gamma_1, \Gamma_2 \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : A} \qquad \Delta; \emptyset \vdash n : \mathbb{N} \qquad \frac{\Delta; \Gamma_1 \vdash e_1 : \mathbb{N} \qquad \Delta; \Gamma_2 \vdash e_2 : \mathbb{N}}{\Delta; \Gamma_1, \Gamma_2 \vdash e_1 + e_2 : \mathbb{N}}$$

$$\frac{\Delta; \Gamma_1 \vdash e : \mathbb{N} \qquad \Delta; \Gamma_2 \vdash e_0 : A \qquad \Delta; x : A \vdash e_S : A}{\Delta; \Gamma_1, \Gamma_2 \vdash \mathtt{iter}(e_0, e, x.\, e_S) \vdash A} \qquad \frac{\Delta; \Gamma, x : A \vdash e : B}{\Delta; \Gamma \vdash \lambda x.\, e : A \multimap B} \qquad \frac{\Delta; \Gamma_1 \vdash e_1 : A \multimap B \qquad \Delta; \Gamma_2 \vdash e_2 : A}{\Delta; \Gamma_1, \Gamma_2 \vdash e_1\, e_2 : B}$$

$$\frac{\Delta; \Gamma_1 \vdash e_1 : A \qquad \Delta; \Gamma_2 \vdash e_2 : B}{\Delta; \Gamma_1, \Gamma_2 \vdash (e_1, e_2) : A \otimes B} \qquad \frac{\Delta; \Gamma_1 \vdash e_1 : A_1 \otimes A_2 \qquad \Delta; \Gamma_1, x : A_1, y : A_2 \vdash e_2 : B}{\Delta; \Gamma_1, \Gamma_2 \vdash \mathsf{let}\ (x, y) = e_1\ \mathsf{in}\ e_2 : B}$$

$$\Delta; \emptyset \vdash \mathsf{chan}() : \mathsf{Get}(A) \otimes \mathsf{Put}(A) \qquad \frac{\Delta; \Gamma_1 \vdash e_1 : \mathsf{Get}(A) \qquad \Delta; \Gamma_2 \vdash e_2 : A \multimap \mathbb{1}}{\Delta; \Gamma_1, \Gamma_2 \vdash \mathtt{get}(e_1, e_2) : \mathbb{1}}$$

$$\frac{\Delta; \Gamma_1 \vdash e_1 : \mathsf{Put}(A) \qquad \Delta; \Gamma_2 \vdash e_2 : A}{\Delta; \Gamma_1, \Gamma_2 \vdash \mathtt{put}(e_1, e_2) : \mathbb{1}} \qquad \frac{\Delta, X; \Gamma \vdash e : A \qquad \Delta \vdash \Gamma \qquad X \notin \Delta}{\Delta; \Gamma \vdash \Lambda.e : \forall X.\, A} \qquad \frac{\Delta; \Gamma \vdash e : \forall X.\, A \qquad \Delta \vdash B}{\Delta; \Gamma \vdash e\langle\rangle : A[B/X]}$$

$$\frac{\Delta; \Gamma \vdash e : A[B/X]}{\Delta; \Gamma \vdash \mathsf{pack}\ e : \exists X.\, A} \qquad \frac{\Delta; \Gamma_1 \vdash e : \exists X.\, A \qquad \Delta, X; \Gamma_2, x : A \vdash e_2 : B \qquad \Delta \vdash B \qquad \Delta \vdash \Gamma_2 \qquad X \notin \Delta}{\Delta; \Gamma_1, \Gamma_2 \vdash \mathsf{unpack}\ e\ \mathsf{as}\ x\ \mathsf{in}\ e_2 : B}$$

Figure 8.4: Polymorphic linear type-system for λ_{CHAN}.

the type formers of the monomorphic case to the polymorphic case:

$$\mathbb{1}_{\text{sem, poly}}(\_) \triangleq \mathbb{1}_{\text{sem}} \qquad \mathbb{B}_{\text{sem, poly}}(\_) \triangleq \mathbb{B}_{\text{sem}} \qquad \mathbb{N}_{\text{sem, poly}}(\_) \triangleq \mathbb{N}_{\text{sem}}$$

$$(\mathbf{T} \multimap \mathbf{U})(\delta) \triangleq \mathbf{T}(\delta) \multimap \mathbf{U}(\delta) \qquad (\mathbf{T} \otimes \mathbf{U})(\delta) \triangleq \mathbf{T}(\delta) \otimes \mathbf{U}(\delta)$$

$$\mathsf{Get}(\mathbf{T})(\delta) \triangleq \mathsf{Get}(\mathbf{T}(\delta)) \qquad \mathsf{Put}(\mathbf{T})(\delta) \triangleq \mathsf{Put}(\mathbf{T}(\delta))$$

$$(\forall X.\, \mathbf{T})(\delta) \triangleq \{f \mid \forall \mathbf{U}.\, \mathbf{wp}\ f()\ \{w.\, w \in (\mathbf{T}[\mathbf{U}/X])(\delta)\}\}$$

$$(\exists X.\, \mathbf{T})(\delta) \triangleq \{v \mid \exists \mathbf{U}.\, v \in (\mathbf{T}[\mathbf{U}/X])(\delta)\}$$

$$(X)(\delta) \triangleq \{v \mid X \in \mathsf{dom}(\delta) * v \in \delta(X)\}$$

where δ is a finite map from type variables to semantic types and where we define $(\mathbf{T}[\mathbf{U}/X])(\delta) \triangleq \mathbf{T}(\delta[X := \mathbf{U}(\delta)])$.

Semantic typing. For the semantic typing, we now use contexts of the form $\mathbf{\Gamma} : \mathsf{id} \xrightarrow{\text{fin}} \mathsf{PolySemType}$, and we define the semantic typing relation as:

$$\begin{aligned}
\mathcal{D}[\![\Delta]\!] &\triangleq \{\delta : \mathsf{id} \xrightarrow{\text{fin}} \mathsf{SemType} \mid \mathop{\ast}_{X \in \Delta} \delta(X) \text{ is defined}\} \\
\mathcal{G}[\![\mathbf{\Gamma}]\!](\delta) &\triangleq \{\gamma \mid \mathop{\ast}_{x : \mathbf{T} \in \Gamma} \gamma(x) \text{ is defined} * \gamma(x) \in \mathbf{T}(\delta)\} \\
\delta =_\Delta \delta' &\triangleq \forall X \in \Delta.\, (\forall v.\, v \in \delta(X) \dashv\vdash v \in \delta'(X)) \\
\Delta \vDash \mathbf{T} &\triangleq \forall \delta, \delta'.\, \delta =_\Delta \delta' \Rightarrow (\forall v.\, v \in \mathbf{T}(\delta) \dashv\vdash v \in \mathbf{T}(\delta')) \\
\Delta \vDash \mathbf{\Gamma} &\triangleq \forall (x : \mathbf{T}) \in \mathbf{\Gamma}.\, \Delta \vDash \mathbf{T}
\end{aligned}$$

$$\begin{aligned}
&\Delta; \mathbf{\Gamma} \vDash e : \mathbf{T} \triangleq \\
&\quad \vdash \exists \alpha.\, \$\alpha \twoheadrightarrow \forall \delta, \gamma.\, \delta \in \mathcal{D}[\![\Delta]\!] \twoheadrightarrow \gamma \in \mathcal{G}[\![\mathbf{\Gamma}]\!](\delta) \twoheadrightarrow \mathbf{wp}\ \gamma(e)\ \{v.\, v \in \mathbf{T}(\delta)\}
\end{aligned}$$

Soundness. We once again show that in each of the rules, we can replace the syntactic notion ($\vdash$) with the semantic notion ($\vDash$). Specifically, we prove the rules in Fig. 8.5. In addition, we use the logical relation and Theorem 51 to derive:

Lemma 55. *If $\emptyset; \emptyset \vDash e : \mathbf{T}$, then e terminates safely.*

$$\Delta; x : \mathbf{T} \vDash x : \mathbf{T} \qquad \frac{\Delta; \mathbf{\Gamma} \vDash e : \mathbf{U}}{\Delta; \mathbf{\Gamma}, x : \mathbf{T} \vDash e : \mathbf{U}} \qquad \Delta; \emptyset \vDash () : \mathbb{1}_{\text{sem, poly}} \qquad \frac{\Delta; \mathbf{\Gamma}_1 \vDash e_1 : \mathbb{1}_{\text{sem, poly}} \qquad \Delta; \mathbf{\Gamma}_2 \vDash e_2 : \mathbf{T}}{\Delta; \mathbf{\Gamma}_1, \mathbf{\Gamma}_2 \vDash e_1; e_2 : \mathbf{T}}$$

$$\Delta; \mathbf{\Gamma} \vDash b : \mathbb{B}_{\text{sem, poly}} \qquad \frac{\Delta; \mathbf{\Gamma}_1 \vDash e : \mathbb{B}_{\text{sem, poly}} \qquad \Delta; \mathbf{\Gamma}_2 \vDash e_1 : \mathbf{T} \qquad \Delta; \mathbf{\Gamma}_2 \vDash e_2 : \mathbf{T}}{\Delta; \mathbf{\Gamma}_1, \mathbf{\Gamma}_2 \vDash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \mathbf{T}} \qquad \Delta; \emptyset \vDash n : \mathbb{N}_{\text{sem, poly}}$$

$$\frac{\Delta; \mathbf{\Gamma}_1 \vDash e_1 : \mathbb{N}_{\text{sem, poly}} \qquad \Delta; \mathbf{\Gamma}_2 \vDash e_2 : \mathbb{N}_{\text{sem, poly}}}{\Delta; \mathbf{\Gamma}_1, \mathbf{\Gamma}_2 \vDash e_1 + e_2 : \mathbb{N}_{\text{sem, poly}}} \qquad \frac{\Delta; \mathbf{\Gamma}_1 \vDash e : \mathbb{N}_{\text{sem, poly}} \qquad \Delta; \mathbf{\Gamma}_2 \vDash e_0 : \mathbf{T} \qquad \Delta; x : \mathbf{T} \vDash e_S : \mathbf{T}}{\Delta; \mathbf{\Gamma}_1, \mathbf{\Gamma}_2 \vDash \mathtt{iter}(e_0, e, x.\, e_S) \vDash \mathbf{T}}$$

$$\frac{\Delta; \mathbf{\Gamma}, x : \mathbf{T} \vDash e : \mathbf{U}}{\Delta; \mathbf{\Gamma} \vDash \lambda x.\, e : \mathbf{T} \multimap \mathbf{U}} \qquad \frac{\Delta; \mathbf{\Gamma}_1 \vDash e_1 : \mathbf{T} \multimap \mathbf{U} \qquad \Delta; \mathbf{\Gamma}_2 \vDash e_2 : \mathbf{T}}{\Delta; \mathbf{\Gamma}_1, \mathbf{\Gamma}_2 \vDash e_1\, e_2 : \mathbf{U}} \qquad \frac{\Delta; \mathbf{\Gamma}_1 \vDash e_1 : \mathbf{T} \qquad \Delta; \mathbf{\Gamma}_2 \vDash e_2 : \mathbf{U}}{\Delta; \mathbf{\Gamma}_1, \mathbf{\Gamma}_2 \vDash (e_1, e_2) : \mathbf{T} \otimes \mathbf{U}}$$

$$\frac{\Delta; \mathbf{\Gamma}_1 \vDash e_1 : \mathbf{T}_1 \otimes \mathbf{T}_2 \qquad \Delta; \mathbf{\Gamma}_1, x : \mathbf{T}_1, y : \mathbf{T}_2 \vDash e_2 : \mathbf{U}}{\Delta; \mathbf{\Gamma}_1, \mathbf{\Gamma}_2 \vDash \mathsf{let}\ (x, y) = e_1\ \mathsf{in}\ e_2 : \mathbf{U}} \qquad \Delta; \emptyset \vDash \mathsf{chan}() : \mathsf{Get}(\mathbf{T}) \otimes \mathsf{Put}(\mathbf{T})$$

$$\frac{\Delta; \mathbf{\Gamma}_1 \vDash e_1 : \mathsf{Get}(\mathbf{T}) \qquad \Delta; \mathbf{\Gamma}_2 \vDash e_2 : \mathbf{T} \multimap \mathbb{1}_{\text{sem, poly}}}{\Delta; \mathbf{\Gamma}_1, \mathbf{\Gamma}_2 \vDash \mathtt{get}(e_1, e_2) : \mathbb{1}_{\text{sem, poly}}} \qquad \frac{\Delta; \mathbf{\Gamma}_1 \vDash e_1 : \mathsf{Put}(\mathbf{T}) \qquad \Delta; \mathbf{\Gamma}_2 \vDash e_2 : \mathbf{T}}{\Delta; \mathbf{\Gamma}_1, \mathbf{\Gamma}_2 \vDash \mathtt{put}(e_1, e_2) : \mathbb{1}_{\text{sem, poly}}}$$

$$\frac{\Delta, X; \mathbf{\Gamma} \vDash e : \mathbf{T} \qquad \Delta \vDash \mathbf{\Gamma} \qquad X \notin \Delta}{\Delta; \mathbf{\Gamma} \vDash \Lambda.e : \forall X.\, \mathbf{T}} \qquad \frac{\Delta; \mathbf{\Gamma} \vDash e : \forall X.\, \mathbf{T}}{\Delta; \mathbf{\Gamma} \vDash e\langle\rangle : \mathbf{T}[\mathbf{U}/X]} \qquad \frac{\Delta; \mathbf{\Gamma} \vDash e : \mathbf{T}[\mathbf{U}/X]}{\Delta; \mathbf{\Gamma} \vDash \mathsf{pack}\ e : \exists X.\, \mathbf{T}}$$

$$\frac{\Delta; \mathbf{\Gamma}_1 \vDash e : \exists X.\, \mathbf{T} \qquad \Delta, X; \mathbf{\Gamma}_2, x : \mathbf{T} \vDash e_2 : \mathbf{U} \qquad \Delta \vDash \mathbf{U} \qquad \Delta \vDash \mathbf{\Gamma}_2 \qquad X \notin \Delta}{\Delta; \mathbf{\Gamma}_1, \mathbf{\Gamma}_2 \vDash \mathsf{unpack}\ e\ \mathsf{as}\ x\ \mathsf{in}\ e_2 : \mathbf{U}}$$

Figure 8.5: The compatibility lemmas for the polymorphic logical relation.

Chapter 9

Foundations of Transfinite Iris

Thus far, we have focused on applying Transfinite Iris by discussing its program logics (§7) and using them to verify several case studies (§8). Let us now turn to the foundations of Transfinite Iris to discuss how it is modeled (*i.e.,* to the *core logic* in Fig. 7.1, consisting of the base logic and impredicative invariants).

As with Iris (see §4), the model of the base logic of Transfinite Iris factors into two largely orthogonal parts: the part dealing with step-indexing, and the part dealing with ownership and resources. We first set aside resources to focus on the existential property (§9.1). Then, we include resources to discuss the revised base logic and the effect of transfinite step-indexing on the rules of Iris (§9.2). Finally, we discuss the recursive domain equation that we have to solve in order to obtain impredicative invariants in Transfinite Iris (§9.3).

9.1 The Existential Property via Transfinite Step-Indexing

As we have discussed in §6.3 (and in §4.3), in a traditional step-indexed logic propositions are modeled as predicates over natural numbers. To validate the existential property, in Transfinite Iris, we model our propositions as predicates on *ordinals.* Below, we explain how ordinals as step-indices give rise to the existential property. We focus on step-indexed propositions that are *only* parametric in the step-index (and no resources yet; see §9.2).

Ordinals. We write *Ord* for the type of ordinals. As with natural numbers, there is a zero ordinal 0 and, for each ordinal α, a strictly larger successor ordinal $\mathrm{s}\ \alpha > \alpha$. Going beyond natural numbers: for each family of ordinals $f : X \rightarrow \mathit{Ord}$, there exists an ordinal $\sup_{x:X} f\ x$ which is larger than $f\ x$ for all $x : X$. For instance, we obtain the first infinite ordinal ω as $\sup_{n:\mathbb{N}} \mathrm{s}^n\ 0$.

The supremum sup can be taken as long as X is a *small type*, a type in a universe below the universe containing the type of ordinals. (One can think of X as being a set and *Ord* being a proper class.) Since Rocq's universes are quite expressive, X can refer to countable types like natural numbers, uncountable types like $\mathbb{N} \rightarrow \mathbb{B}$, and much more. In fact, to use ordinals as time credits in §7.2, we put another copy of the ordinal type into a universe below the universe of the ordinals that we use for step-indexing (using universe polymorphism in Rocq). We have denoted it $\mathbb{O}$ in §7.3.2 to avoid confusion.

Step-indexed propositions. Step-indexed propositions are families of ordinary propositions, indexed by ordinals, with the essential "down-closure" property that truth at one step-index implies truth at all lower step-indices:

Definition 56. *The type of step-indexed propositions SProp consists of families $P : Ord \rightarrow Prop$ which are down-closed, meaning if $\alpha \in P$ and $\beta \le \alpha$, then $\beta \in P$.*

Given two step-indexed propositions $P, Q : SProp$, we define entailment between them $P \vdash_{\text{sprop}} Q$ as

$$P \vdash_{\text{sprop}} Q \triangleq \forall \alpha : Ord.\, \alpha \in P \Rightarrow \beta \in Q$$

The familiar logical connectives (conjunction, disjunction, implication, *etc.*) lift to step-indexed propositions and satisfy the standard rules of intuitionistic higher-order logic. For instance, given $\Phi : X \rightarrow SProp$, the step-indexed proposition $\exists x : X.\, \Phi\, x \triangleq \{\alpha \mid \exists x : X.\, \alpha \in \Phi\, x\}$ satisfies the standard logical rules for existential quantification. In addition to the standard connectives, we equip step-indexed propositions with a later modality $\triangleright P$:

$$\triangleright P \triangleq \{\alpha \mid \forall \beta < \alpha.\, \beta \in P\}$$

This definition restricts to the usual model of $\triangleright P$ when applied to natural numbers. It also validates the central rules of step-indexed logics, such as Löb induction (which is now proven by transfinite induction on the step-index) and $P \vdash_{\text{sprop}} \triangleright P$ (which follows directly from down-closure).

The existential property. With step-indexed propositions in hand, we proceed to prove the existential property.

Theorem 57 (Existential Property)**.**
Fix a small type X. If $\vdash_{\text{sprop}} \exists x : X.\, \Phi\, x$, then $\vdash_{\text{sprop}} \Phi\, x$ for some $x : X$.

Proof. Using classical reasoning, we proceed by contradiction and assume that $\vdash_{\text{sprop}} \Phi\, x$ is false for each $x : X$. This unfolds to $\forall x.\, \exists \alpha.\, \alpha \notin \Phi\, x$. Using the axiom of choice, we obtain for each x an ordinal $\alpha_x : Ord$ such that $\alpha_x \notin \Phi\, x$ (*i.e.,* $\forall x.\, \alpha_x \notin \Phi\, x$). From $\vdash_{\text{sprop}} \exists x.\, \Phi\, x$ we know that for each β there is some x such that $\beta \in \Phi\, x$ (*i.e.,* $\forall \beta.\, \exists x.\, \beta \in \Phi\, x$). To obtain a contradiction, we will construct an ordinal β, where $\beta \notin \Phi\, x$ for each x (*i.e.,* it suffices to show $\exists \beta.\, \forall x.\, \beta \notin \Phi\, x$). First, we observe that $\Phi\, x$ is down-closed, so $\alpha \notin \Phi\, x$ for any $\alpha \ge \alpha_x$ (since $\forall x.\, \alpha_x \notin \Phi\, x$). We now define $\beta \triangleq \sup_{x:X} \alpha_x$. For each x, we have $\beta \ge \alpha_x$ by construction, hence $\beta \notin \Phi\, x$. Thus, we have reached a contradiction. ◻

The proof rests crucially on *Ord* containing suprema of small families (*i.e.,* families over small types). In the mechanization,[1] where our definition of ordinals is based on the work of Kirst and Smolka,[2] the notion of "small types" is made precise by Rocq's universe hierarchy; *SProp* satisfies the existential property for any type belonging to a universe smaller than the universe of *Ord*.

Unlike the traditional Iris development, the model of Transfinite Iris—of ordinals and of the existential property—relies on extensionality axioms (*i.e.,* propositional extensionality and functional extensionality) and classical axioms (*i.e.,* the axiom of choice and excluded middle). The soundness of Rocq under this combination of axioms is a consequence of the model of the calculus of inductive constructions in ZFC.[3]

[1] Spies et al., *Transfinite Iris appendix and Rocq development*, 2021 [Spi+21a].

[2] Kirst and Smolka, "Categoricity results and large model constructions for second-order ZF in dependent type theory", 2019 [KS19].

[3] Werner, "Sets in types, types in sets", 1997 [Wer97].

9.2 The Base Logic of Transfinite Iris

Let us now discuss how we integrate transfinite step-indices into the base logic. We stay at the same level of abstraction as in §4.3. In particular, as in Iris, in

$$
\begin{aligned}
\phi &\triangleq \{(\alpha, a) \mid \phi\}\\
P \wedge Q &\triangleq \{(\alpha, a) \mid (\alpha, a) \in P \wedge (\alpha, a) \in Q\}\\
P \vee Q &\triangleq \{(\alpha, a) \mid (\alpha, a) \in P \vee (\alpha, a) \in Q\}\\
P \Rightarrow Q &\triangleq \{(\alpha, a) \mid \forall \beta \leq \alpha. \forall b \succcurlyeq a.\, b \in \mathcal{V} \Rightarrow (\beta, b) \in P \Rightarrow (\beta, b) \in Q\}\\
\forall x : X.\, P(x) &\triangleq \{(\alpha, a) \mid \forall x : X.\, (\alpha, a) \in P(x)\}\\
\exists x : X.\, P(x) &\triangleq \{(\alpha, a) \mid \exists x : X.\, (\alpha, a) \in P(x)\}\\
P * Q &\triangleq \{(\alpha, a) \mid \exists a_1, a_2.\, a = a_1 \cdot a_2 \wedge (\alpha, a_1) \in P \wedge (\alpha, a_2) \in Q\}\\
P \mathrel{-\!\!*} Q &\triangleq \{(\alpha, a) \mid \forall \beta \leq \alpha. \forall b.\, a \cdot b \in \mathcal{V} \Rightarrow (\beta, b) \in P \Rightarrow (\beta, a \cdot b) \in Q\}\\
\Box\, P &\triangleq \{(\alpha, a) \mid (\alpha, |a|_{\mathrm{core}}) \in P\}\\
\triangleright P &\triangleq \{(\alpha, a) \mid \forall \beta < \alpha.\, (\beta, a) \in P\}\\
\mathrel{|\!\!\Rrightarrow} P &\triangleq \big\{(\alpha, a) \,\big|\, \forall \beta \leq \alpha. \forall a_f.\, a \cdot a_f \in \mathcal{V} \Rightarrow \exists b.\, b \cdot a_f \in \mathcal{V} \wedge (\beta, b) \in P\big\}\\
\mathsf{Own}\,(a) &\triangleq \{(\alpha, b) \mid a \preccurlyeq b\}
\end{aligned}
$$

Figure 9.1: Connectives of the Transfinite Iris base logic.

Transfinite Iris, resource algebras are step-indexed using step-indexed types (see §4.5). We will discuss step-indexed types in Transfinite Iris in §9.3.

Analogously to the Iris base logic, we define the type *UPred*(M) as predicates over step-indices—here ordinals—and resources. For this revised definition, the entailment relation $P \vdash Q$ is given by

$$P \vdash Q \triangleq \forall \alpha, a.\, a \in \mathcal{V} \Rightarrow (\alpha, a) \in P \Rightarrow (\alpha, a) \in Q$$

and the definition of the logic connectives is depicted in Fig. 9.1. Their definition mirrors the standard Iris definition, but *crucially* using ordinals α, β instead of natural numbers n, m as step-indices.

Using ordinals instead of natural numbers superficially looks like a small change—indeed, in Fig. 9.1 (compared to Fig. 4.3 in §4.3) effectively only the symbols change. However, it has several important ramifications, which we will discuss below: (1) most importantly, we obtain the existential property; (2) we have to generalize the guarded fixpoint construction; (3) we lose two commuting rules of the later modality; and (4) we have to generalize the solution of the recursive domain equation (from §4.5).[4]

[4] A fifth change that should not be neglected is that this change ripples through the Rocq development of the base logic, leading to changes throughout.

The existential property. Let us start with the existential property. Analogously to Theorem 57, we obtain the following existential property for *UPred* (which, by being stated for *UPred* now, additionally considers resources):

Lemma 58 (Existential Property for Validity)**.**
Fix a small type X. If $\vdash \exists x : X.\, \Phi\, x$, then $\vdash \Phi\, x$ for some $x : X$.

Proof. Analogous to Theorem 57. □

This existential property is stated in terms of validity for the entailment $P \vdash Q$. As such, it requires an empty context for "$\exists x : X.\, \Phi\, x$", which means $\exists x : X.\, \Phi\, x$ must hold in particular for the empty resource. In the presence of resources, it is sometimes convenient to relax this restriction to predicates Φ which are not unconditionally valid (*e.g.*, for situations where we have already allocated some ghost state). For these situations, one can use the following variant of the existential property, where $\mathsf{sat}(P) \triangleq \forall \alpha.\, \exists a.\, a \in \mathcal{V} \land (\alpha, a) \in P$:

Lemma 59 (Existential Property for Satisfiability)**.**
Fix a small type X. If $\mathsf{sat}(\exists x : X.\, \Phi\, x)$, then $\mathsf{sat}(\Phi\, x)$ for some $x : X$.

Proof. The proof is similar to Theorem 57; the presence of valid, potentially non-empty resources has no significant effect on the proof. □

In this version of the existential property, the satisfiability predicate $\mathsf{sat}(P)$ allows P to depend on valid resources. A more detailed discussion on how satisfiability can be used for proving adequacy is given by Spies.[5]

[5] Spies, *Making adequacy of Iris satisfying*, 2024 [Spi24].

Guarded fixpoints. Since we change the underlying step-indexing of the base logic, we also have to the redefine the *guarded fixpoints* $\mu\, f x.\, P\, f\, x$ (see §4.3) in the transfinite setting. Transfinite Iris provides a general construction for fixpoints of step-indexed types, similar to the one of Gianantonio and Miculan.[6] For the sake of simplicity, we restrict our attention below to fixpoints for step-indexed predicates of type $X \to \mathit{UPred}(M)$ for arbitrary types X and non-step-indexed resources M.[7] To obtain such a guarded fixpoint, we must iterate the predicate body $P : (X \to \mathit{UPred}(M)) \to (X \to \mathit{UPred}(M))$ using transfinite recursion. That is, for finite step-indices, simple recursion on natural numbers was enough to do the trick (see §4.3). Here, we now use "strong" transfinite recursion (*i.e.,* at index α recursive results are available for all $\beta < \alpha$). Concretely, we define $(\mu\, f y.\, P\, f\, y)(x) \triangleq \{(\alpha, a) \mid (\alpha, a) \in f_\alpha(x)\}$ where f_α is defined recursively on α as

[6] Gianantonio and Miculan, "Unifying recursive and co-recursive definitions in sheaf categories", 2004 [GM04].

[7] For step-indexed resources, one has to add additional step-indexed validity assumptions to the fixpoint construction. The full definition (generalized beyond fixpoints of *UPred*-predicates) is given in the Transfinite Iris Rocq development.

$$f_\alpha(x) \triangleq P(f_{<\alpha})(x) \qquad f_{<\alpha}(x) \triangleq \left\{(\beta, a) \,\middle|\, \forall \gamma < \alpha.\, \gamma \le \beta \Rightarrow (\gamma, a) \in f_\gamma(x)\right\}.$$

For every ordinal α, the approximation f_α applies P to the predicate $f_{<\alpha}$, and $f_{<\alpha}$ is itself defined using the smaller approximations f_γ for $\gamma < \alpha$. The fixpoint $\mu\, f y.\, P\, f\, y$ is then obtained as the diagonal of the approximations f_α.

We can then show that $\mu\, f y.\, P\, f\, y$ is a guarded fixpoint. More specifically, we obtain a fixpoint of P in the following sense:

Lemma 60. *Let every occurrence of f in P be guarded by a later modality. Then $\mu\, f y.\, P\, f\, y$ is a fixpoint, meaning $\forall x.\, (\mu\, f y.\, P\, f\, y)(x) \dashv\vdash P(\mu\, f y.\, P\, f\, y)(x)$.*

Loss of the commuting rules. We have defined the connectives of the base logic analogous to §4.3. The resulting rules of the Transfinite Iris base logic are depicted in Fig. 9.2.[8] Notably, this base logic is missing two rules compared to the standard rules of Iris (see §4.3): as the price for the existential property, *we lose* the following two commuting rules of the later modality.[9]

[8] As in §4.3, the rule OWN-VALID only holds for non-step-indexed resources, and there is a corresponding version for step-indexed resources.

[9] Transfinite Iris also loses a rule about commuting a later modality with the ownership connective:

$$\triangleright \mathsf{Own}(a) \vdash \exists b.\, \mathsf{Own}(b) \land \triangleright (a \equiv b)$$

Since this rule is used rarely in Iris, we have not discussed it yet (or in §4.3), and its loss does not significantly impact Transfinite Iris.

$$\textsc{later-sep}\quad \triangleright(P * Q) \dashv\vdash \triangleright P * \triangleright Q \qquad\qquad \textsc{later-exists}\quad \frac{X \text{ non-empty}}{\triangleright(\exists x : X.\, P(x)) \dashv\vdash \exists x : X.\, \triangleright P(x)}$$

This is not by accident. Regardless of the step-indexing technique used, the existential property is fundamentally incompatible with the rule LATER-EXISTS. This incompatibility is witnessed by the following theorem:

ent-refl
$$P \vdash P$$

ent-trans
$$\frac{P \vdash Q \qquad Q \vdash R}{P \vdash R}$$

pure-intro
$$\frac{\phi}{P \vdash \phi}$$

from-pure
$$\frac{P \vdash \phi \qquad \phi \Rightarrow (P \vdash Q)}{P \vdash Q}$$

Figure 9.2: Rules of the base logic of Transfinite Iris.

and-elim-l
$$P \wedge Q \vdash P$$

and-elim-r
$$P \wedge Q \vdash Q$$

and-intro
$$\frac{P \vdash Q \qquad P \vdash R}{P \vdash Q \wedge R}$$

or-intro-l
$$P \vdash P \vee Q$$

or-intro-r
$$Q \vdash P \vee Q$$

or-elim
$$\frac{P \vdash R \qquad Q \vdash R}{P \vee Q \vdash R}$$

all-intro
$$\frac{\forall x : X.\, (P \vdash Q(x))}{P \vdash \forall x : X.\, Q(x)}$$

all-elim
$$\frac{a : X}{(\forall x : X.\ P(x)) \vdash P(a)}$$

exist-intro
$$\frac{a : X \qquad P \vdash Q(a)}{P \vdash \exists x : X.\, Q(x)}$$

exist-elim
$$\frac{\forall x : X.\, (P(x) \vdash Q)}{\exists x : X.\, P(x) \vdash Q}$$

wand-intro
$$\frac{P * Q \vdash R}{P \vdash Q \mathrel{-\!\!*} R}$$

wand-elim
$$\frac{P \vdash Q \mathrel{-\!\!*} R}{P * Q \vdash R}$$

sep-weaken
$$P * Q \vdash P$$

sep-true
$$P \vdash P * \mathsf{True}$$

sep-comm
$$P * Q \vdash Q * P$$

sep-split
$$\frac{P \vdash P' \qquad Q \vdash Q'}{P * Q \vdash P' * Q'}$$

sep-assoc
$$P * (Q * R) \dashv\vdash (P * Q) * R$$

pers-elim
$$\Box\, P \vdash P$$

pers-mono
$$\frac{P \vdash Q}{\Box\, P \vdash \Box\, Q}$$

pers-pure
$$\phi \vdash \Box\, \phi$$

pers-all
$$\forall x : X.\ \Box\, P(x) \vdash \Box\, \forall x : X.\, P(x)$$

pers-exists
$$\Box\, \exists x : X.\, P(x) \vdash \exists x : X.\ \Box\, P(x)$$

pers-and-sep
$$(\Box\, P) \wedge Q \vdash (\Box\, P) * Q$$

pers-idemp
$$\Box\, P \vdash \Box\, \Box\, P$$

later-intro
$$P \vdash \triangleright P$$

later-mono
$$\frac{P \vdash Q}{\triangleright P \vdash \triangleright Q}$$

Loeb
$$\frac{\triangleright P \vdash P}{\vdash P}$$

later-all
$$\triangleright(\forall x : X.\, P(x)) \dashv\vdash \forall x : X.\ \triangleright P(x)$$

later-pers
$$\triangleright \Box\, P \dashv\vdash \Box \triangleright P$$

upd-return
$$P \vdash \mathrel{|\!\!\Rrightarrow} P$$

upd-bind
$$\mathrel{|\!\!\Rrightarrow} P * (P \mathrel{-\!\!*} \mathrel{|\!\!\Rrightarrow} Q) \vdash \mathrel{|\!\!\Rrightarrow} Q$$

own-empty
$$\mathsf{True} \vdash \mathsf{Own}\,(\varepsilon)$$

own-upd
$$\frac{a \rightsquigarrow B}{\mathsf{Own}\,(a) \vdash \mathrel{|\!\!\Rrightarrow} \exists b \in B.\, \mathsf{Own}\,(b)}$$

own-valid
$$\mathsf{Own}\,(a) \vdash a \in \mathcal{V}$$

own-pers
$$\mathsf{Own}\,(a) \vdash \Box\, \mathsf{Own}\,(|a|_{\mathrm{core}})$$

own-sep
$$\mathsf{Own}\,(a) * \mathsf{Own}\,(b) \dashv\vdash \mathsf{Own}\,(a \cdot b)$$

Theorem 61. *There exists no consistent logic with a sound later modality (*i.e., $\vdash \triangleright P$ *implies* $\vdash P$*),* Löb *induction (*i.e., $(\triangleright P \vdash P)$ *implies* $\vdash P$*), and the commuting rule* later-exists*, which also enjoys the existential property.*

Proof. By way of contradiction, assume there is a logic satisfying these properties. As we will prove below, the proposition $\exists n : \mathbb{N}.\ \triangleright^n \mathsf{False}$ is provable in such a logic. Using the existential property, we then obtain an n at the meta level such that $\vdash \triangleright^n \mathsf{False}$. By soundness of the later modality, this entails $\vdash \mathsf{False}$, which contradicts consistency (*i.e.,* that $\vdash \mathsf{False}$ is not provable).

We prove $\vdash (\exists n : \mathbb{N}.\ \triangleright^n \mathsf{False})$ by Löb induction. Thus, it suffices to show $(\triangleright \exists n : \mathbb{N}.\ \triangleright^n \mathsf{False}) \vdash (\exists n : \mathbb{N}.\ \triangleright^n \mathsf{False})$. By the commuting rule later-exists, this is reduced to:

$$(\exists n : \mathbb{N}.\ \triangleright\triangleright^n \mathsf{False}) \vdash (\exists n : \mathbb{N}.\ \triangleright^n \mathsf{False})$$

We eliminate the existential quantifier in the assumption to obtain a witness n. We instantiate the existential quantifier in the conclusion by picking $n + 1$. The remaining claim $(\triangleright\triangleright^n \mathsf{False}) \vdash (\triangleright^{n+1} \mathsf{False})$ is immediate. □

The underlying problem is that, in a transfinitely step-indexed logic, the proposition $\triangleright(\exists x : X.\ P(x))$ at step-index ω means that for every natural number n there exists a witness x_n for which P holds at step-index n, but $\exists x : X.\ \triangleright P(x)$ at step-index ω demands one witness x that will work for every natural number n. (In contrast, for step-indexing with natural numbers, the case $n = 0$ is trivial and for all others, one can always choose the witness x_{n-1}, which works for all smaller step-indices $m \leq n - 1$ due to downwards closure.)

Given that later-exists does not hold, it is not surprising that we also lose the rule later-sep (*i.e.,* $\triangleright(P * Q) \vdash (\triangleright P) * (\triangleright Q)$), since in the model of the base logic, the separating conjunction connective $P * Q$ is defined by *existentially quantifying* over the split of resources between P and Q (see Fig. 9.1). Specifically, we lose the direction $\triangleright(P * Q) \vdash \triangleright P * \triangleright Q$.

Avoiding the commuting rules. The absence of the commuting rules means that there are Iris proofs which do not hold verbatim in Transfinite Iris. However, that does not mean that the commuting rules cannot be avoided.

In particular, we tweaked the safety program logic of Iris to avoid the commuting rules—obtaining the safety program logic of Transfinite Iris (see Fig. 7.1). More specifically, we inserted $\blacktriangleright P \triangleq \exists n.\ \triangleright^n P$ (augmented with additional updates) into the safety weakest precondition, which allows one to eliminate an arbitrary number of laters per step.[10] For this program logic, we were able to adapt various existing Iris proofs with only minor changes (for details see the Transfinite Iris Rocq development[11]). In particular, we were able to verify the barrier of Jung et al.:[12] the flagship example for the introduction of higher-order ghost state in Iris. Besides these safety examples, our liveness examples from §8 of course also avoid the commuting rules.

Nevertheless, there are Iris developments which use the commuting rules for proofs where our tweaks do not apply (*e.g.,* the type system of RustBelt[13]). Recovering every Iris development is beyond the scope of this dissertation.

[10] The existential property ensures that the existential in $\blacktriangleright P$ is meaningful (unlike in regular Iris, where one can prove $\blacktriangleright\mathsf{False}$).

[11] Spies et al., *Transfinite Iris appendix and Rocq development*, 2021 [Spi+21a].

[12] Jung et al., "Higher-order ghost state", 2016 [Jun+16].

[13] Jung et al., "RustBelt: Securing the foundations of the Rust programming language", 2018 [Jun+18a].

9.3 Invariants and the Recursive Domain Equation

Let us now turn to how impredicative invariants are integrated into Transfinite Iris. Recall (from §4.5.2) that for impredicative invariants in traditional Iris, we must solve the following recursive domain equation:

$$\begin{aligned}&\mathit{iProp} \triangleq \mathit{UPred}(\mathcal{M})\\ &\quad \mathcal{M} \triangleq \mathit{Map}(\mathbb{P}(\mathbb{N}), \uplus) \times \mathit{Map}(\mathbb{P}^{\text{fin}}(\mathbb{N}), \uplus) \times \mathit{Map}(\mathit{Auth}(\mathbb{N} \xrightharpoonup{\text{fin}} \mathit{Ag}(\blacktriangleright \mathit{iProp}))) \times \cdots\end{aligned}$$

where, through $\mathcal{M}$, the type *iProp* refers to itself. In Transfinite Iris, we must solve the same recursive domain equation for our revised notion of transfinitely step-indexed propositions $\mathit{UPred}(M)$ (§9.2).

Step-indexed types. For finite Iris, the answer to this recursive domain equation is to use (finitely) step-indexed types (see §4.5.2). In Transfinite Iris, we also use step-indexed types to solve this equation, but we alter their definition to use ordinal step-indices instead of natural number step-indices. Specifically, we define a step-indexed type to be an *ordered family of equivalences* (OFE): a pair of a type X and a family of equivalence relations $(\overset{\alpha}{=})$. These equivalence relations must become increasingly coarse as α decreases (i.e., if $x \overset{\alpha}{=} y$ and $\beta \leq \alpha$, then $x \overset{\beta}{=} y$). For instance, *SProp* is an OFE with $P \overset{\alpha}{=} Q \triangleq \forall \beta \leq \alpha.\, \beta \in P \textit{ iff } \beta \in Q$. Besides OFEs, we also generalize the notion of a *complete ordered family of equivalences* (COFE), an OFE with additional limit operations.[14] (COFEs are used for defining fixpoints such as the one in Lemma 60.)

Solving the recursive domain equation. Let us return to the recursive domain equation for *iProp*. It is well-known that domain equations can be solved over *finite* step-indexed types.[15] However, while transfinite variants of step-indexed types have been considered before, it was unclear whether the construction of solutions to domain equations[16] could be adapted to the transfinite setting. In order to complete the model, we have therefore defined a novel construction for solving domain equations, extending the existence of solutions to the transfinite case. The construction is described in the appendix of Transfinite Iris[17] and implemented in the Transfinite Iris Rocq development.

With *iProp* as the solution of the recursive domain equation, we thus have all the parts of Transfinite Iris together to claim the following theorem:

Theorem 62. *Transfinite Iris is consistent and enjoys the existential property.*

[14] Not every type former that is a COFE in the finite setting is also a COFE in the transfinite setting. One notable exception are "*sigma types* $\Sigma(x : X).\, \phi(x)$" that restrict a type X to those elements satisfying ϕ. For them, it depends on which predicate ϕ is chosen. For some predicates ϕ, the resulting type is a COFE in the finite setting but not in the transfinite setting.

[15] America and Rutten, "Solving reflexive domain equations in a category of complete metric spaces", 1989 [AR89].

[16] Gianantonio and Miculan, "Unifying recursive and co-recursive definitions in sheaf categories", 2004 [GM04]; Birkedal et al., "First steps in synthetic guarded domain theory: Step-indexing in the topos of trees", 2012 [Bir+12].

[17] Spies et al., *Transfinite Iris appendix and Rocq development*, 2021 [Spi+21a].

Chapter 10

Related Work

We compare with (1) approaches for proving liveness properties with step-indexed logics, (2) prior separation logics for proving liveness properties, and (3) approaches using transfinitely step-indexed models.

Proving liveness properties via safety properties in step-indexed logics. In the absence of the existential property (and transfinite step-indexing), the typical strategy for proving liveness properties with step-indexed logics is to first prove a safety property (*e.g.,* termination, but with a fixed upper bound on the number of steps; a safety property) and then to show that the safety property implies the liveness property (*e.g.,* termination). Reducing liveness properties to safety properties comes at a cost. All of the approaches below [DH10; DH12; MJP19; TJH17; FKB21a; Tim+24a] end up imposing restrictions that increase the proof overhead (*e.g.,* require determining suitable bounds) and/or exclude programs (*e.g.,* bounded termination excludes programs that terminate in every execution, but where the number of steps has no fixed, static upper bound).

Dockins and Hobor[1] introduce a logic for proving termination with a fixed upper bound, a safety property, which implies the liveness property termination.

Mével, Jourdan, and Pottier[2] extend Iris with time credits, enabling them to prove complexity bounds (*i.e.,* termination with a fixed upper bound). As mentioned in §7.2, while the termination logic of Transfinite Iris, $\mathsf{Termination}_{\mathsf{SHL}}$ (§7.2), also has time credits, it is not suited for proving time complexity results, since it allows for steps that do not require spending a time credit.

Tassarotti, Jung, and Harper[3] extend Iris with support for proving concurrent termination-preserving refinements. However, since they use finite instead of transfinite step-indexing, they can only do so by imposing restrictions on the source language: it can only have bounded non-determinism, and it can only stutter for a fixed number of times.[4] These restrictions effectively turn their termination-preserving refinement into a safety property in the sense that it is determined by finite execution prefixes. More specifically, under these restrictions, one can establish termination-preserving refinement by proving "for every n-step target execution, there is a corresponding m-step source execution where $m \geq n$ (including source-stutter steps)".[5] This finite-prefix characterization of termination-preserving refinement is the underlying reason why Tassarotti, Jung, and Harper can then give a finitely step-indexed simulation relation (akin to the one in §6.3), which is adequate for termination-preserving refinement, yet—in effect—only relates finite execution prefixes. Unfortunately, the restrictions on the source language also have practical consequences: since Tassarotti, Jung, and Harper do not have our no-later stuttering (see §7.1.3) and

[1] Dockins and Hobor, "A theory of termination via indirection", 2010 [DH10]; Dockins and Hobor, "Time bounds for general function pointers", 2012 [DH12].

[2] Mével, Jourdan, and Pottier, "Time credits and time receipts in Iris", 2019 [MJP19].

[3] Tassarotti, Jung, and Harper, "A higher-order logic for concurrent termination-preserving refinement", 2017 [TJH17].

[4] As mentioned in §7.1.3, they encode stuttering by choosing as the source language the lexicographic product of actual source steps and a stutter budget with a fixed, global upper bound: every actual source step resets the stutter budget to its fixed upper bound.

[5] This property is not sufficient when the source language has unbounded non-determinism. For example, consider the case where the target t_∞ is an infinite loop, and the source $s_{<\infty}$ non-deterministically picks a natural number n, and then executes n steps before terminating. The target t_∞ is not a termination-preserving refinement of $s_{<\infty}$, but they satisfy this property.

they only support stuttering the source for a fixed number of steps, their restrictions are too strong for examples like `memo_rec`, where an unbounded number of stutters is required. (The number of steps needed to find a cached result in the lookup table can grow arbitrarily). In contrast to our work, Tassarotti, Jung, and Harper do support concurrency.

Frumin, Krebbers, and Birkedal[6] define a program logic in Iris to prove the security property *timing-sensitive non-interference*. While timing-sensitive non-interference ensures termination preservation, it is expressed as a lock-step program equivalence. As such, Frumin, Krebbers, and Birkedal impose even stronger restrictions than the work of Tassarotti, Jung, and Harper.[7]

After Transfinite Iris was published, Timany et al.[8] have introduced Trillium, a variant of Iris for transferring liveness properties from a source transition system to a target program. The essence of their approach is that they first establish liveness properties at the source level (*i.e.,* about the source transition system) *outside of Iris* and then—via an intensional refinement *inside Iris* (*i.e.,* a lock-step simulation between target and source)—transfer the liveness property to the target program. To ensure that their intensional refinement is "finitely approximable" in a step-indexed setting, they impose the restriction of "relative image-finiteness", a relaxation of the bounded-nondeterminism restriction of Tassarotti, Jung, and Harper.[9] (With relative image-finiteness, it suffices if the source transition system has bounded non-determinism *after* the transitions are filtered using a program-state-dependent relation ξ.) For termination-preserving refinements, Timany et al. only allow stuttering up to a fixed, global bound similar to Tassarotti, Jung, and Harper (see also §7.1.3). For termination, Timany et al. can handle the example "randomly pick a natural number n and then terminate after n steps", but it requires them to instantiate the adequacy theorem of Trillium with a specific state-dependent relation ξ designed for this particular program (to ensure relative image-finiteness). It would be interesting to base Trillium on Transfinite Iris; Timany et al. conjecture that this would remove the relative image-finiteness restriction. In contrast to this work, they do consider liveness properties of concurrent programs, including fair termination-preserving refinement.

Liveness in separation logics. In the literature, a number of separation logics for liveness reasoning have been introduced [LF16; LF18; Roc+16; DOs+21; YHB08; Cha11]: Liang and Feng[10] develop the program logic LiLi, a "rely-guarantee style program logic for verifying linearizability and progress together for concurrent objects under fair scheduling". Rocha Pinto et al.[11] extend the concurrent separation logic TaDA[12] with ordinal time credits to prove program termination. D'Osualdo et al.[13] go further and target more general liveness properties such as "always-eventually" properties. Yoshida, Honda, and Berger[14] and Charguéraud[15] introduce program logics capable of liveness reasoning, even in higher-order stateful settings. In particular, Yoshida, Honda, and Berger verify a termination-preserving refinement of a memoized factorial function.

The fundamental difference to our work is that all of these logics are *not* step-indexed. In Transfinite Iris, we have focused on enabling liveness in a *step-indexed setting*, allowing us to use features like Löb induction, guarded recursion, and impredicative invariants. It was precisely the combination of these features that allowed us to prove a generic specification for `memo_rec` and

[6] Frumin, Krebbers, and Birkedal, "Compositional non-interference for fine-grained concurrent programs", 2021 [FKB21a].

[7] Tassarotti, Jung, and Harper, "A higher-order logic for concurrent termination-preserving refinement", 2017 [TJH17].

[8] Timany et al., "Trillium: Higher-order concurrent and distributed separation logic for intensional refinement", 2024 [Tim+24a].

[9] Tassarotti, Jung, and Harper, "A higher-order logic for concurrent termination-preserving refinement", 2017 [TJH17].

[10] Liang and Feng, "A program logic for concurrent objects under fair scheduling", 2016 [LF16]; Liang and Feng, "Progress of concurrent objects with partial methods", 2018 [LF18].

[11] Rocha Pinto et al., "Modular termination verification for non-blocking concurrency", 2016 [Roc+16].

[12] Rocha Pinto, Dinsdale-Young, and Gardner, "TaDA: A logic for time and data abstraction", 2014 [RDG14].

[13] D'Osualdo et al., "TaDA Live: Compositional reasoning for termination of fine-grained concurrent programs", 2021 [DOs+21].

[14] Yoshida, Honda, and Berger, "Logical reasoning for higher-order functions with local state", 2008 [YHB08].

[15] Charguéraud, "Characteristic formulae for the verification of imperative programs", 2011 [Cha11].

then instantiate it for multiple clients. To the best of our knowledge, verifying `memo_rec` generically is not possible in the above logics.

Some of these logics [LF16; LF18; Roc+16; DOs+21] are, however, capable of proving liveness properties of *concurrent* programs. In this dissertation, to focus on dealing with the "existential dilemma" of step-indexed separation logic, we have explored liveness reasoning in the sequential setting first. That said, Transfinite Iris is compatible with concurrency—our safety program logic includes several representative case studies of concurrent *safety* reasoning, which we have ported from Iris to Transfinite Iris.

After Transfinite Iris was published, Gäher et al.[16] have developed Simuliris, a version of Iris *without step-indexing* for proving fair termination-preserving refinements of concurrent programs in the context of verifying compiler optimizations. They define a simulation relation using least- and greatest fixpoints, but without the later modality (*i.e.,* no Löb induction, no impredicative invariants, no guarded fixpoints, *etc.*).

[16] Gäher et al., "Simuliris: A separation logic framework for verifying concurrent program optimizations", 2022 [Gäh+22].

Transfinite step-indexing. Before Transfinite Iris, transfinite step-indexing had already been used for two purposes in the literature: modeling a separation logic for safety reasoning [Sve+18] and constructing logical relations for a single language [BBS13; BBM14; SSB16; Bir+12; SKD21; BGM21].

First, we discuss the relationship to the only prior separation logic using transfinite step-indexing—SLR by Svendsen et al.,[17] a logic for the promising weak memory model.[18] SLR is restricted to proving *safety properties.* In their work, transfinite indices up to ω^2 are used to handle hypothetical steps of computation called certification steps, which occur between actual steps in the promising model to justify speculative weak memory behavior. However, since their indices only go up to ω^2, their step-indices are too small to obtain the existential property for quantification over infinite sets.

[17] Svendsen et al., "A separation logic for a promising semantics", 2018 [Sve+18].

[18] Kang et al., "A promising semantics for relaxed-memory concurrency", 2017 [Kan+17].

Second, we discuss the relationship to logical relations defined using transfinite step-indexing. Birkedal, Bizjak, and Schwinghammer[19] use step-indexing up to ω_1 (the least uncountable ordinal) to give a logical-relations model for reasoning about must-contextual equivalence for a language with countable nondeterminism. (The inductively-defined must-convergence predicate for the language with countable nondeterminism has ω_1 as closure ordinal.) Bizjak, Birkedal, and Miculan[20] show how to define the logical-relations model for countable nondeterminism in a step-indexed logic, namely in the internal logic of the topos of sheaves over ω_1. We conjecture that their step-indexed logic enjoys the existential property, restricted to countable types, and that they use it implicitly for their adequacy result (see Lemma 4 in [BBM14]).

[19] Birkedal, Bizjak, and Schwinghammer, "Step-indexed relational reasoning for countable nondeterminism", 2013 [BBS13].

[20] Bizjak, Birkedal, and Miculan, "A model of countable nondeterminism in guarded type theory", 2014 [BBM14].

Svendsen, Sieczkowski, and Birkedal[21] use step-indexing up to ω^2 to decouple steps of computation from *logical steps* (such as unfolding an invariant)—*i.e.,* to allow multiple logical steps per physical step of computation. They do not address liveness reasoning. In their work, Svendsen, Sieczkowski, and Birkedal already solve a recursive domain equation (RDE) in COFEs for the ω^2 case. We have extended this result to arbitrary (uncountable) ordinals, which are needed for the existential property.

[21] Svendsen, Sieczkowski, and Birkedal, "Transfinite step-indexing: Decoupling concrete and logical steps", 2016 [SSB16].

Birkedal et al.[22] solve RDEs in the more general category of *sheaves.* For Transfinite Iris, we considered simply working with solutions to RDEs in sheaves, but decided it was impractical as it would have led to problems with

[22] Birkedal et al., "First steps in synthetic guarded domain theory: Step-indexing in the topos of trees", 2012 [Bir+12].

the mechanization of Transfinite Iris—a central concern for its deployment in practice. In particular, were we to use sheaves, functions in Transfinite Iris could no longer be encoded as Rocq functions with a property of non-expansiveness, as they are in the original Iris, but would instead have to be represented as (transfinite) sequences of functions. In the present work, we thus instead solve a RDE in the category of (transfinite) *COFEs*, a subcategory of sheaves. (Note that this is a novel result in relation to the work of Birkedal et al., because their RDE solutions are not guaranteed to be COFEs.) In contrast to sheaves, COFEs are well suited to mechanization in Rocq.

Spies, Krishnaswami, and Dreyer[23] use transfinite step-indexing up to ω^ω to prove termination of a linear language with higher-order channels. As explained in §8.3, the present work subsumes, extends, and mechanizes their work, in the process reducing the size of their proof significantly.

[23] Spies, Krishnaswami, and Dreyer, "Transfinite step-indexing for termination", 2021 [SKD21].

Bahr, Graulund, and Møgelberg[24] define a statically-typed, pure functional language, and use transfinite step-indexing up to $\omega \cdot 2$ to show that well-typed programs enjoy a certain liveness property. Aside from the use of transfinite step-indexing, this work is quite different from ours. It focuses on type systems that ensure liveness properties, rather than techniques for verification of liveness properties in a general-purpose language with features like general recursion and higher-order state.

[24] Bahr, Graulund, and Møgelberg, "Diamonds are not forever: Liveness in reactive programming with guarded recursion", 2021 [BGM21].

Building on top of Transfinite Iris, Guéneau et al.[25] have developed a program logic for reasoning about an idealized version of the OCaml foreign function interface (*i.e.,* the connection between OCaml and C). While they do not consider liveness properties, their program logic is based on Transfinite Iris, because they consider an operational semantics with angelic non-determinism, which leads to existential quantifiers in their definition of the weakest precondition (similar to the existential quantifier in the definition of $\textbf{wp}\ e\ \{v.\ Q(v)\}$ in Fig. 7.6). The existential property of Transfinite Iris enables them to extract these existential quantifiers as part of proving adequacy.

[25] Guéneau et al., "Melocoton: A program logic for verified interoperability between OCaml and C", 2023 [Gué+23].

Part III

Later Credits

Chapter 11

Introduction

In this part of the dissertation, we focus on the *later* ($\triangleright$) *modality*[1] (see §3.2) of step-indexed separation logics like Iris. Though pushed to the margins in much of the literature, as we have already seen in Part I and Part II, the later modality is in fact central to how step-indexed separation logics work, because it makes it possible to do step-indexed reasoning at a higher level of abstraction—without being forced to reason about the underlying step-indices directly and without engaging in tedious "step-index arithmetic" as in earlier formulations of step-indexing.[2]

[1] Appel et al., "A very modal model of a modern, major, general type system", 2007 [App+07].

[2] Dreyer, Ahmed, and Birkedal, "Logical step-indexed logical relations", 2011 [DAB11].

Yet, ironically, the later modality is often viewed as a "necessary evil" by practitioners: In proofs, laters appear in hypotheses (*e.g.,* upon unfolding an implicitly recursive construction or starting a Löb induction), because they serve as "guards" preventing paradoxes of circular reasoning.[3] But from a user perspective, once $\triangleright P$ appears as a hypothesis, the name of the game is figuring out how to *eliminate* the guarding "$\triangleright$" in order to make use of P.

[3] See §3.3 and §8.2 in Jung et al., "Iris from the ground up: A modular foundation for higher-order concurrent separation logic", 2018 [Jun+18b].

This brings us to our main topic: *the later elimination problem.* Although there exist a number of techniques for eliminating laters in step-indexed proofs (which we have already encountered in §3 and will review below), there are several known situations where none of these techniques apply, thus ruling natural proof strategies out of consideration and in some cases making it unclear how to carry out the proof at all. In this part of the dissertation, we propose a new technique for escaping these unfortunate situations by exploiting the fact that we are working in a separation logic. Specifically, *we treat "the right to eliminate a later" as an ownable resource* and then apply standard separation-logic reasoning to that resource. We realize this idea through a new logical mechanism we call **Later Credits**, and we demonstrate its effectiveness on a range of interesting use cases. But before we explain how later credits work and where they shine, let us begin by illustrating the later elimination problem with a concrete example.

The later elimination problem. We have already encountered the later elimination problem *en passant* in the previous parts (*e.g.,* when doing a Löb induction in §3.2 or opening invariants in §3.4). Let us now focus on it by returning to *impredicative invariants* (§3.4.1). Recall that, in Iris, invariants $\boxed{R}$ are used to share state between threads. For example, we can pick $R \triangleq \exists n : \mathbb{N}.\, \ell \mapsto n$ to share access to the location ℓ and, at the same time, constrain ℓ to only store natural numbers. Impredicative invariants are so central to Iris that the rule for *opening* an invariant (*i.e.,* accessing the contents R of the invariant) was presented on page 1 of the original "Iris 1.0" paper,[4] albeit in

[4] Jung et al., "Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning", 2015 [Jun+15].

the following, "simplified for presentation purposes" form (and we will return to what is simplified about it shortly):

$$\frac{\{R * P\}\ e\ \{R * Q\} \qquad e \text{ physically atomic}}{\boxed{R} \vdash \{P\}\ e\ \{Q\}}$$

The rule says that if we open $\boxed{R}$, then we can assume R in our precondition and have to show R holds again after evaluating the atomic expression e. As discussed in §3.4, the rule is restricted to *atomic* expressions to prevent other threads that are interleaved with e from potentially observing inconsistent states in which R does not hold.

The distinguishing feature of Iris's invariants (see §3.4.1) is that they are *impredicative* (*i.e.*, they can store an arbitrary *iProp* proposition R). We have already used this power, for example, to handle examples with higher-order state (in Example 21) and to define a logical relation (in §8.3). The price of impredicativity (as discussed in §4.5) is that their model is cyclic—cyclic to the extent that naive models of $\boxed{R}$ are not well-founded and, to obtain a well-founded model of $\boxed{R}$, the only known approach is to stratify the cyclic construction using step-indexing.[5]

[5] Svendsen and Birkedal, "Impredicative concurrent abstract predicates", 2014 [SB14]; Jung et al., "Iris from the ground up: A modular foundation for higher-order concurrent separation logic", 2018 [Jun+18b].

The side effect of using step-indexing to resolve cycles in the model of invariants is that, when invariants are opened, a *later modality* appears as a "guard" to protect against paradoxically circular reasoning. That is, as we have seen in §3.4.1, Iris's real invariant opening rule is the following:[6]

[6] In line with the invariant opening rule above, we write "e physically atomic" for the side condition $\text{atomic}(e)$ from §3.4.

hoare-inv-open

$$\frac{\{\triangleright R * P\}\ e\ \{v.\ \triangleright R * Q(v)\}_{\mathcal{E}\setminus\mathcal{N}} \qquad \mathcal{N} \subseteq \mathcal{E} \qquad e \text{ physically atomic}}{\boxed{R}^{\mathcal{N}} \vdash \{P\}\ e\ \{v.\ Q(v)\}_{\mathcal{E}}}$$

This rule adds two later later modalities ($\triangleright$) in the pre and post of the premise (as an artifact of the step-indexed model of impredicative invariants; see §4.5.2). Besides later modalities, this rule adds namespaces $\mathcal{N}$ and masks $\mathcal{E}$ to avoid reentrancy and makes the postcondition parametric over the resulting value v.

Thus, after applying this rule, the user needs to *eliminate* the "$\triangleright$" guarding R in the precondition, so that they can use R in verifying e. To do so, Iris presently offers the following three options:

1. *Timeless propositions (see Fig. 3.5).* For the subclass of *timeless* propositions—which include propositions that are pure (*e.g.*, $\text{even}(n)$) or assert only first-order ownership (*e.g.*, $\ell \mapsto 42$)—laters can be eliminated because the model of these propositions ignores the step-index.
2. *Commuting rules (see Fig. 3.2).* The later modality commutes with most logical connectives (*e.g.*, existential quantification and separating conjunction). Thus, in many cases, we can commute the later out of the way.
3. *Program steps (see Fig. 3.2).* With every program step, we can eliminate a guarding later. More precisely, if P is guarded by a later before the step, then the later can be removed *after* the step.

The problem is that in some cases none of these techniques apply. We illustrate such a case with an example: *nested invariants*.[7] Consider the invariant:

[7] We use nested invariants here, because they are one of the simpler examples to illustrate where the existing practices are not enough. In practice, most proofs do not need to use nested invariants. However, plenty of proofs put other (more complicated) step-indexed assertions into invariants, and then guarding laters cause trouble (*e.g.*, see §13.1 and §13.2).

$$\boxed{\exists \ell.\ \boxed{\exists n : \mathbb{N}.\ \ell \mapsto n}^{\mathcal{N}_1} * \gamma \Rightarrow_{1/2} \ell}^{\mathcal{N}_2}$$

Here, the location ℓ is existentially quantified and connected to a logical identifier γ through a ghost variable $\gamma \Rightarrow_{1/2} \ell$ (*i.e.,* ghost state from the resource algebra $((0,1],+) \times Ag(Loc)$; see §4.2.2). If we need the contents of the inner invariant $(\exists n : \mathbb{N}.\, \ell \mapsto n)$ to justify the next step (*e.g.,* to dereference ℓ), then we are in a quandary. If we open the outer invariant, we get

$$\triangleright \left(\exists \ell.\, \boxed{\exists n : \mathbb{N}.\, \ell \mapsto n}^{\mathcal{N}_1} * \gamma \Rightarrow_{1/2} \ell\right).$$

After applying commuting and timelessness rules and eliminating the existential, we are left with $\triangleright \boxed{\exists n : \mathbb{N}.\, \ell \mapsto n}^{\mathcal{N}_1}$ and $\gamma \Rightarrow_{1/2} \ell$. At this point, we have a later guarding $\boxed{\exists n : \mathbb{N}.\, \ell \mapsto n}^{\mathcal{N}_1}$ and are therefore stuck: invariants are not timeless (eliminating the first option), there is nothing to commute (eliminating the second option), and we need $\ell \mapsto n$ *before* (*i.e.,* as a precondition for verifying) the next step (eliminating the third option).

As we will see in this part of the dissertation, the case of nested invariants is not an isolated one. There are a number of realistic scenarios in step-indexed separation logics—not common scenarios exactly, but ones that do occur periodically—where none of the "standard" later elimination options apply. At present, the only way of handling such scenarios is to attempt some non-trivial, non-local refactoring of the proof structure—or to admit defeat.

Later credits. We now introduce a fourth option for later elimination—*later credits*. Later credits support what we call *amortized step-indexed reasoning*—eliminating laters based on *previous* program steps. The basic idea of amortized reasoning is that we decouple the proof steps where laters are eliminated from the proof steps where we execute the program. Instead of eliminating one later after every program step (option 3 above), we obtain a credit £1 after every program step, a *later credit*. This credit can subsequently be used anywhere in the rest of the proof that we want to eliminate a later modality, not just the present step. For example, we can save a credit £1 from one step, keep it for two subsequent steps, and then use it before the next step to eliminate a later guarding an invariant assertion. In particular, the credit £1 can also be used as part of purely logical reasoning where there is no program in sight.

The reader may wonder whether later credits are really a backdoor for reintroducing into the logic the kind of explicit step-index manipulation that the later modality was designed to avoid. The answer is no: because unlike step-indices, later credits are implemented as *resources* in a separation logic, and hence they inherit all the modular reasoning principles associated with resources in separation logic. To wit: if we own £1 (*i.e.,* it is in our precondition), then we alone get to decide how we want to spend it without any interference from other parts of the program/proof. If we want to spend it to eliminate a later, then we can do so with a credit spending rule. If we want to share it with other functions, then we can pass it to them as part of their precondition. If we want to keep it to ourselves during a function call, then we can frame it around the function call. If we want to share it with other threads, then we can put it into an invariant that is shared with those threads. In short, we can reason about later credits using all the standard reasoning patterns that are available for resources in separation logic. None of this is possible with traditional step-index manipulation.

Later credits enable two kinds of applications: First, they can simplify existing proofs. Later credits can help where step-indexing previously got in the way and cluttered the proofs. Second, they can enable proofs which were seemingly not possible with standard later elimination techniques. We will see examples of both kinds of applications in the following.

Contributions. In this part of the dissertation, our main contributions are later credits and the amortized step-indexing technique that they enable. We develop both as an extension of Iris. We explain later credits (in §12), discuss their soundness (in §14), and show how they complement existing approaches to eliminate multiple laters per step (in §15). We focus primarily on later credits in standard Iris (with finite step-indexing), but later credits also work in Transfinite Iris, discussed in §15.2.

We demonstrate the use of later credits (in §13) with two flagship examples (one of each kind):

1. *New proofs.* One interesting application of step-indexed logical relations in prior work has been in proving that expressions in higher-order stateful languages can be *reordered.*[8] However, due to trouble with the later modality, the step-indexed logical relations of prior work can only handle very restricted forms of reordering operations with shared higher-order state (*e.g.,* ones using shared, but immutable state). We show how later credits make it possible to prove much more sophisticated reorderings, in particular for JavaScript-inspired promises (in §13.1).

2. *Proof simplification.* One of the original motivations of Iris was proving so-called *logical atomicity* for concurrent data structures. Sadly, step-indexing has always caused trouble for logical atomicity, sometimes ruling out natural and perfectly valid proof strategies and requiring "ugly" workarounds.[9] We show how to avoid such workarounds by instead using later credits to implement simpler and more intuitive proofs of logical atomicity (in §13.2).

Besides these flagship examples, we develop several smaller case studies to demonstrate usefulness of later credits (in §15). In particular, we develop a form of *prepaid* invariants, which can be opened around physically atomic instructions *without* a guarding later, and we show that later credits can be used to prove the kind of "reverse refinements" introduced by Svendsen, Sieczkowski, and Birkedal[10] without requiring the transfinite step-indexing model that Svendsen, Sieczkowski, and Birkedal needed. Later credits and all of the above examples are mechanized in Rocq using the Iris Proof Mode.[11] See the Later Credits Rocq development for the Rocq proofs.[12] Additionally, see the appendix of the Later Credits paper for further technical details on the examples and case studies.[13]

[8] Krogh-Jespersen, Svendsen, and Birkedal, "A relational model of types-and-effects in higher-order concurrent separation logic", 2017 [KSB17]; Timany et al., "A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of runST", 2018 [Tim+18].

[9] Jung, *Logical atomicity in Iris: The good, the bad, and the ugly*, 2019 [Jun19].

[10] Svendsen, Sieczkowski, and Birkedal, "Transfinite step-indexing: Decoupling concrete and logical steps", 2016 [SSB16].

[11] Krebbers, Timany, and Birkedal, "Interactive proofs in higher-order concurrent separation logic", 2017 [KTB17]; Krebbers et al., "MoSeL: A general, extensible modal framework for interactive proofs in separation logic", 2018 [Kre+18].

[12] Spies et al., *Later Credits Rocq development and technical documentation*, 2022 [Spi+22b].

[13] Spies et al., *Later Credits Rocq development and technical documentation*, 2022 [Spi+22b].

Chapter 12

Later Credits in a Nutshell

Later credits connect the *step-indexing* and *resources* pillars underlying Iris to obtain a simple, yet powerful mechanism for later elimination. In this chapter, we explain the essence of later credits.

CREDIT-SPLIT
$$\pounds(n+m) \dashv\vdash \pounds n * \pounds m$$

CREDIT-TIMELESS
$$\mathsf{timeless}(\pounds n)$$

HOARE-PURE-STEP-CREDIT
$$\frac{\{P * \pounds 1\}\; e_2\; \{v.\, Q(v)\}_{\mathcal{E}} \qquad e_1 \rightarrow_{\text{pure}} e_2}{\{P\}\; e_1\; \{v.\, Q(v)\}_{\mathcal{E}}}$$

HOARE-LE-UPD-EXEC
$$\frac{\{P\}\; e\; \{v.\, Q(v)\}_{\mathcal{E}}}{\left\{\mathrel{|\!\!\Rrightarrow}_{\mathsf{le}} P\right\}\; e\; \{v.\, Q(v)\}_{\mathcal{E}}}$$

LE-UPD-LATER
$$\pounds 1 * \triangleright P \vdash \mathrel{|\!\!\Rrightarrow}_{\mathsf{le}} P$$

LE-UPD-RETURN
$$P \vdash \mathrel{|\!\!\Rrightarrow}_{\mathsf{le}} P$$

LE-UPD-BIND
$$(\mathrel{|\!\!\Rrightarrow}_{\mathsf{le}} P) * (P \mathrel{-\!\!*} \mathrel{|\!\!\Rrightarrow}_{\mathsf{le}} Q) \vdash \mathrel{|\!\!\Rrightarrow}_{\mathsf{le}} Q$$

Figure 12.1: A selection of the core proof rules for later credits.

The later credits mechanism—whose rules are shown in Fig. 12.1—rests on two central pieces: a new resource $\pounds n$, called the *later credits*, and a new update modality $\mathrel{|\!\!\Rrightarrow}_{\mathsf{le}} P$, called the *later elimination update*. Intuitively, one can think of owning $\pounds n$ as the right to eliminate n later modalities, and of the later elimination update $\mathrel{|\!\!\Rrightarrow}_{\mathsf{le}} P$ as an extension of Iris's update modality $\mathrel{|\!\!\Rrightarrow} P$ (§3.6) that additionally allows updating $\triangleright P$ to P using later credits.

The later credits mechanism factors into two parts:

1. We *receive* later credits by taking program steps. For example, we receive one later credit $\pounds 1$ by executing a pure step with HOARE-PURE-STEP-CREDIT. After the program step, the new credit becomes available in the precondition of the Hoare triple of the successor expression e_2. (The proof rules for load, store, allocation, *etc.* similarly generate one credit after the step.) Once we have received credits, we can combine and split them freely with CREDIT-SPLIT.

2. We *spend* later credits through later elimination updates $\mathrel{|\!\!\Rrightarrow}_{\mathsf{le}}$. That is, we can give up a credit $\pounds 1$, and, in exchange, eliminate a later modality by updating $\triangleright P$ to P with LE-UPD-LATER. In particular, we can use this rule to eliminate a guarding later from one of our assumptions. Once we own $\mathrel{|\!\!\Rrightarrow}_{\mathsf{le}} P$, the update can be executed as usual. For example, just like standard updates $\mathrel{|\!\!\Rrightarrow} P$, we can execute $\mathrel{|\!\!\Rrightarrow}_{\mathsf{le}} P$ in the precondition of Hoare triples with HOARE-LE-UPD-EXEC.

Have we just managed to replace one modality ($\triangleright$) with another one ($\mathrel{|\!\!\Rrightarrow}_{\mathsf{le}}$)? No, far from it. There are two key distinctions between the two modalities. The

hoare-frame
$$\frac{\{P\}\ e\ \{v.\ Q(v)\}_{\mathcal{E}}}{\{P * R\}\ e\ \{v.\ Q(v) * R\}_{\mathcal{E}}}$$

upd-return
$$P \vdash \Rrightarrow P$$

upd-bind
$$(\Rrightarrow P) * \ (P \mathrel{-\!\!*} \Rrightarrow Q) \vdash \Rrightarrow Q$$

later-intro
$$P \vdash \triangleright P$$

later-mono
$$\frac{P \vdash Q}{\triangleright P \vdash \triangleright Q}$$

hoare-timeless-strip
$$\frac{\{P * Q\}\ e\ \{v.\ R(v)\}_{\mathcal{E}} \qquad \mathrm{timeless}(Q)}{\{P * \triangleright Q\}\ e\ \{v.\ R(v)\}_{\mathcal{E}}}$$

later-exists
$$\frac{X \text{ non-empty}}{\triangleright \exists x : X.\ \Phi(x) \vdash \exists x : X.\ \triangleright \Phi(x)}$$

later-sep
$$\triangleright(P * Q) \vdash \triangleright P * \triangleright Q$$

hoare-inv-open
$$\frac{\{\triangleright R * P\}\ e\ \{v.\ \triangleright R * Q(v)\}_{\mathcal{E}\setminus\mathcal{N}} \qquad \mathcal{N} \subseteq \mathcal{E} \qquad e \text{ physically atomic}}{\boxed{R}^{\mathcal{N}} \vdash \{P\}\ e\ \{v.\ Q(v)\}_{\mathcal{E}}}$$

Figure 12.2: A selection of Iris's proof rules.

first one is that $\Rrightarrow_{\mathsf{le}} P$ can be executed virtually everywhere in the logic, whereas the elimination of laters is quite restricted. More specifically, to integrate $\Rrightarrow_{\mathsf{le}} P$ into Iris, we replace the update $\Rrightarrow P$ with $\Rrightarrow_{\mathsf{le}} P$ in most of Iris. This modification allows us to execute $\Rrightarrow_{\mathsf{le}} P$ everywhere that we could execute $\Rrightarrow P$ before, including when reasoning about programs—but also in purely logical reasoning (*e.g.,* logical atomicity in §13.2).

The second key distinction is that $\Rrightarrow_{\mathsf{le}}$ is more compositional. Analogous to $\Rrightarrow$ (see upd-return and upd-bind in Fig. 12.2), the update $\Rrightarrow_{\mathsf{le}}$ is a monad with le-upd-return and le-upd-bind, whereas $\triangleright$ is not. (The later modality is only an applicative functor; see later-intro and later-mono in Fig. 12.2.) Since $\Rrightarrow_{\mathsf{le}}$ is a monad, we can use le-upd-bind to accumulate and compose updates. For example, it is trivial to prove transitivity (*i.e.,* $\Rrightarrow_{\mathsf{le}}\Rrightarrow_{\mathsf{le}} P \vdash \Rrightarrow_{\mathsf{le}} P$), or to use one later elimination update to spend *two* credits and eliminate *two* laters. In contrast, the later modality does not satisfy the analogous rule $\triangleright\triangleright P \vdash \triangleright P$, so we cannot fold two laters into one.

Later credits in action. As a first illustration of later credits, let us discuss two small examples. To verify the examples, we use the later credit rules together with standard Iris rules. We have collected the relevant Iris rules (from prior chapters) in Fig. 12.2 to make the examples easier to follow.[1]

[1] Since we will discuss examples centered around invariants, we have annotated all Hoare triple rules with masks here (see §3.4).

In our first example, we show how to save a credit for a few steps to enable a later elimination afterwards. We do not *need* later credits here, because we could use timelessness and later commuting, but it will nevertheless be instructive as a toy example:

Example 63 (Framing Later Credits)**.** In this example, we show:

$$\frac{\forall n.\ \{n \in \mathbb{N}\}\ f\ n\ \{m.\ m \in \mathbb{N}\}}{\boxed{\exists n : \mathbb{N}.\ \ell \mapsto n}^{\mathcal{N}} \vdash \{\mathsf{True}\}\ \ell := f(41+1)\ \{\_.\ \mathsf{True}\}}$$

The way it works is as follows. First, we execute $41 + 1$ with hoare-pure-step-credit and thereby obtain a new later credit $£1$. We are left with proving the entailment $\boxed{\exists n : \mathbb{N}.\ \ell \mapsto n}^{\mathcal{N}} \vdash \{£1\}\ \ell := f(42)\ \{\_.\ \mathsf{True}\}$. We frame $£1$ around the call of f with hoare-frame (in Fig. 12.2), leaving us to prove

$$\boxed{\exists n : \mathbb{N}.\ \ell \mapsto n}^{\mathcal{N}} \vdash \{£1\}\ \ell := m\ \{\_.\ \mathsf{True}\}$$

for some $m \in \mathbb{N}$. After opening the invariant with hoare-inv-open (in Fig. 12.2), we have to show $\{\pounds 1 * \triangleright(\exists n : \mathbb{N}.\, \ell \mapsto n)\}\ \ell := m\ \{\_.\ \triangleright(\exists n : \mathbb{N}.\, \ell \mapsto n)\}_{\top\setminus\mathcal{N}}$. We spend the later credit to eliminate the later modality with le-upd-later, leaving us to prove $\left\{\Rrightarrow_{\mathsf{le}}(\exists n : \mathbb{N}.\, \ell \mapsto n)\right\}\ \ell := m\ \{\_.\ \triangleright(\exists n : \mathbb{N}.\, \ell \mapsto n)\}_{\top\setminus\mathcal{N}}$. Subsequently, we execute the later elimination update with hoare-le-upd-exec, leaving us to prove $\{\exists n : \mathbb{N}.\, \ell \mapsto n\}\ \ell := m\ \{\_.\ \triangleright(\exists n : \mathbb{N}.\, \ell \mapsto n)\}_{\top\setminus\mathcal{N}}$. The rest of the proof is routine, using later-intro (in Fig. 12.2) in the postcondition. •

For our second example, let us return to the example from §11, *nested invariants.* In this example, we will show how one can transfer later credits via invariants to, for example, enable opening a nested invariant:

Example 64 (Later Credits in Invariants)**.** In this example, we show:

$$\boxed{\exists r.\ \boxed{\exists n : \mathbb{N}.\, r \mapsto n}^{\mathcal{N}_1} * \gamma \Mapsto_{1/2} r * \pounds 1}^{\mathcal{N}_2} \vdash \left\{\gamma \Mapsto_{1/2} \ell\right\}\ !\,\ell\ \left\{v.\ v \in \mathbb{N} * \gamma \Mapsto_{1/2} \ell\right\}$$

We abbreviate the contents of the inner invariant as $I(r) \triangleq \exists n : \mathbb{N}.\, r \mapsto n$ and of the outer invariant as $J \triangleq (\exists r.\ \boxed{I(r)}^{\mathcal{N}_1} * \gamma \Mapsto_{1/2} r * \pounds 1)$. First, we open the outer invariant, and we are left to prove:

$$\left\{\gamma \Mapsto_{1/2} \ell * \triangleright J\right\}\ !\,\ell\ \left\{v.\ v \in \mathbb{N} * \gamma \Mapsto_{1/2} \ell * \triangleright J\right\}_{\top\setminus\mathcal{N}_2}$$

Next, we use the commuting rules to commute the later modality inside:

$$\left\{\gamma \Mapsto_{1/2} \ell * (\exists r.\ \triangleright \boxed{I(r)}^{\mathcal{N}_1} * \triangleright \gamma \Mapsto_{1/2} r * \triangleright \pounds 1)\right\}\ !\,\ell\ \left\{v.\ v \in \mathbb{N} * \gamma \Mapsto_{1/2} \ell * \triangleright J\right\}_{\top\setminus\mathcal{N}_2}$$

Then, we use the fact that—ironically—later credits themselves are *timeless* (credit-timeless) to remove the guarding later modality from the later credit (and the ghost variable) with hoare-timeless-strip, leaving us to prove:

$$\left\{\gamma \Mapsto_{1/2} \ell * (\exists r.\ \triangleright \boxed{I(r)}^{\mathcal{N}_1} * \gamma \Mapsto_{1/2} r * \pounds 1)\right\}\ !\,\ell\ \left\{v.\ v \in \mathbb{N} * \gamma \Mapsto_{1/2} \ell * \triangleright J\right\}_{\top\setminus\mathcal{N}_2}$$

As the next step, we use the fact that the ghost variables must agree (*i.e.,* $\gamma \Mapsto_{1/2} \ell * \gamma \Mapsto_{1/2} r \vdash \ell = r$) to deduce that $\ell = r$. This leaves us to prove

$$\left\{\gamma \Mapsto_{1/2} \ell * \gamma \Mapsto_{1/2} \ell * \triangleright \boxed{I(\ell)}^{\mathcal{N}_1} * \pounds 1\right\}\ !\,\ell\ \left\{v.\ v \in \mathbb{N} * \gamma \Mapsto_{1/2} \ell * \triangleright J\right\}_{\top\setminus\mathcal{N}_2}$$

Afterwards, we can spend the credit (as in Example 63) to eliminate the later modality guarding the inner invariant with le-upd-later and hoare-le-upd-exec. Thus, we are left to prove:

$$\left\{\gamma \Mapsto_{1/2} \ell * \gamma \Mapsto_{1/2} \ell * \boxed{\exists n : \mathbb{N}.\, \ell \mapsto n}^{\mathcal{N}_1}\right\}\ !\,\ell\ \left\{v.\ v \in \mathbb{N} * \gamma \Mapsto_{1/2} \ell * \triangleright J\right\}_{\top\setminus\mathcal{N}_2}$$

From here on, the proof is almost routine. There is, however, one interesting aspect about it: how do we restore the later credit inside the invariant J? That is, *someone* prepaid for our later elimination (*i.e.,* whoever closed the invariant the last time), but now—to close the invariant—we must return the later credit. Fortunately, all physical steps produce later credits, including loading from a location (*i.e.,* the corresponding rule is $\{\ell \mapsto v\}\ !\,\ell\ \{w.\ v = w * \ell \mapsto v * \pounds 1\}_{\mathcal{E}}$). Thus, we can simply use the later credit that we obtain from $!\,\ell$ to close the invariant and finish the proof. •

Although they are simple, these examples show how later credits enable reasoning about later eliminations as an ownable resource, which can be passed around using the rules of separation logic. This kind of reasoning is essential in the applications that follow, in which we frame credits for several steps (in §13.1) and exchange them through invariants (in §13.2 and §15.1).

Chapter 13
Applications of Later Credits

Having discussed the essence of later credits, let us now apply them. We use them for proving reordering refinements (§13.1) and to simplify helping for logical atomicity proofs (§13.2).

13.1 Later Credits for Reordering Refinements

For our first application of later credits, we show how they address a limitation of step-indexed logical relations that arises when proving *reordering refinements*. In a reordering refinement, we want to prove that two expressions e_1 and e_2 are independent in the sense that their execution order is not observable. Concretely, this means we want to show:

$$e_2; e_1 \preceq_{\mathsf{ctx}} e_1; e_2 \qquad \textit{and, more generally,} \qquad e_1 \parallel e_2 \preceq_{\mathsf{ctx}} (e_1, e_2)$$

where $e_1 \parallel e_2$ denotes the parallel composition of e_1 and e_2 (returning the pair of their result values). One way to prove such a contextual refinement is with a step-indexed logical relation. That is, for $e_1, e_2 : \tau$ we show

$$e_1 \parallel e_2 \preceq_{\mathsf{log}} (e_1, e_2) : \tau \times \tau$$

where $\preceq_{\mathsf{log}}$ is a step-indexed relation implying $\preceq_{\mathsf{ctx}}$.

However, with step-indexed logical relations, it is difficult to prove reordering refinements involving shared higher-order state (*i.e.,* memory storing functions; see Example 21 in §3.4.1). This is unfortunate, since such state is one of the main reasons for using step-indexing in the first place. The difficulty arises because laters are eliminated in these relations *asymmetrically*: when proving a logical refinement of the form $e \preceq_{\mathsf{log}} e' : \tau$, elimination of laters is only allowed during steps on the left (steps of e). However, for a reordering refinement like $e_2; e_1 \preceq_{\mathsf{log}} e_1; e_2 : \mathbb{1}$, the laters eliminated when stepping e_2 on the left could be too "early" to help with eliminating laters needed for reasoning about e_2 on the right. Later credits resolve this issue by letting us save credits from the execution of e_2 on the left and use them when reasoning about e_2 on the right. This enables us to reorder operations that use shared, mutable, higher-order state, which is beyond the scope of previous work.[1]

[1] Krogh-Jespersen, Svendsen, and Birkedal, "A relational model of types-and-effects in higher-order concurrent separation logic", 2017 [KSB17]; Timany et al., "A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of runST", 2018 [Tim+18].

In the remainder of this section, we will explain in more detail when expressions can be reordered, which step-indexing related issues can arise, and how later credits address these issues. To keep matters concrete, we focus on a motiviating example where later credits will be essential for the proof: *promises*.

```
            promise : $\mathbb{1} \to \mathsf{pr}(\tau)$
          promise() $\triangleq$ (mklock(), ref(None), ref([]))
            resolve : $\mathsf{pr}(\tau) \times \tau \to \mathbb{1}$
resolve(($l, r, c$), $a$) $\triangleq$ lock($l$); match !$r$ with
                          | Some($b$) $\Rightarrow$ unlock($l$); abort()
                          | None $\Rightarrow$ $r$ := Some($a$); let $fs$ = !$c$ in $c$ := [];
                                  unlock($l$); app ($\lambda f.\, f(a)$) $fs$
                          end
               then : $\mathsf{pr}(\tau) \times (\tau \to \mathbb{1}) \to \mathbb{1}$
   then(($l, r, c$), $f$) $\triangleq$ lock($l$); match !$r$ with
                          | Some($a$) $\Rightarrow$ unlock($l$); $f(a)$
                          | None $\Rightarrow$ $c$ := ($f$ :: !$c$); unlock($l$)
                          end
```

Figure 13.1: Promise implementation in HeapLang.

A motivating example. A promise, in languages such as JavaScript,[2] represents the result of a delayed computation—the value is not available right away, but it is "promised" to be there eventually. We can attach continuations to a promise, which are eventually executed once the promise is resolved.

[2] The promises discussed here are similar but not exactly the same as the asynchronous channels from §8.3. The channels of §8.3 are *linear* (and the focus there is on termination), whereas the promises discussed here support attaching more than one continuation (and the focus is on reordering).

We can implement a simplified version of the mechanism in Iris's HeapLang, depicted in Fig. 13.1. We promise a value of type τ with promise, select the value for the promise with resolve, and attach continuations $f : \tau \to \mathbb{1}$ to a promise with then, which will be executed once the promise has been resolved. Internally, a promise consists of a reference for the result r, a list of continuations c, and a lock l to protect the two. When a promise is resolved, the value is stored in r and all continuations in c are executed with the function app. (We explain the abort case shortly.) The operation then adds the continuation to the list if the promise is unresolved or executes it if the promise is resolved.

What is interesting about promises is that (under suitable conditions) their operations can be reordered. For example, if two continuations f and g are reorderable with respect to each other, then the order in which we attach them to a promise using then does not really matter. Similarly, if a promise is only ever resolved once, then the order in which we call then and resolve does not matter either, since the attached callback will eventually be executed with the resolved value of the promise. Thus, for reorderable f and g, we should be able to prove, for example:

$$\begin{array}{lcll} \mathtt{then}(p, f); & & \mathtt{then}(p, g); & \\ \mathtt{then}(p, g); & \preceq_{\mathrm{ctx}} & \mathtt{resolve}(p, a); & : \mathbb{1} \\ \mathtt{resolve}(p, a) & & \mathtt{then}(p, f) & \end{array}$$

To state this refinement precisely, we need to formalize the conditions on the use of then and resolve. To model when two functions f and g are reorderable, we will use a type system (in §13.1.2). And as far as resolve is concerned, resolving a promise twice is typically considered an error. For example, in JavaScript a repeated resolve attempt has no effect. Thus, we simply rule out multiple resolve attempts in the promise implementation: we implement a second call to resolve as "*safe-failure*" (via abort, which just diverges).

But even once we have formalized these constraints, proving such a refinement remains challenging, because the continuations stored in memory are a form of shared, higher-order state. That is where later credits come into the picture. They will enable us to construct a logical relation that can nevertheless prove this refinement. To do so, we start with Iris's standard binary logical relation, ReLoC[3] (in §13.1.1). We extend ReLoC with support for proving reorderings by adapting ideas from Timany et al.[4] (in §13.1.2). Finally, we show that later credits allow us to prove promise reorderings (in §13.1.3).

[3] Frumin, Krebbers, and Birkedal, "ReLoC: A mechanised relational logic for fine-grained concurrency", 2018 [FKB18]; Frumin, Krebbers, and Birkedal, "ReLoC Reloaded: A mechanized relational logic for fine-grained concurrency and logical atomicity", 2021 [FKB21b].

[4] Timany et al., "A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of runST", 2018 [Tim+18].

13.1.1 ReLoC: Concurrent Logical Relations in Iris

ReLoC uses Iris's program logic to define a binary logical relation. To do relational (*i.e.,* binary) reasoning in Iris, ReLoC uses the technique from CaReSL[5] (which we have already encountered in §7.1.1), in which the "specification" or "source" program (on the right-hand side of the refinement) is represented by ghost state. We call it a *ghost program* in the following to emphasize that it is represented as ghost state. ReLoC considers *concurrent* ghost programs. Thus, in this variant, the ghost program has ghost state assertions of the form $j \Mapsto e$, which mean that thread j in the ghost program is executing expression e, and assertions of the form $\ell \mapsto_{\mathrm{g}} v$, which mean that location ℓ stores v in the ghost program's memory (the subscript g here stands for "ghost").

[5] Turon, Dreyer, and Birkedal, "Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency", 2013 [TDB13].

In ReLoC, the ghost program is executed by updating the ghost assertions with Iris's *fancy update* ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P$ (§4.5.1).[6] For example, to perform a store of w to location ℓ in the ghost program, we have the rule

[6] Note that this is different from Transfinite Iris, where we used a dedicated source update modality (*i.e.,* see $\Rrightarrow_{\mathrm{src}} P$ in §7.1.2) instead of one of Iris's built-in update modalities.

GHOST-STORE

$$j \Mapsto (\ell := w) * \ell \mapsto_{\mathrm{g}} v \vdash {}^{\mathcal{N}_{\mathrm{reloc}}}{\Rrightarrow}^{\mathcal{N}_{\mathrm{reloc}}}\, j \Mapsto () * \ell \mapsto_{\mathrm{g}} w\,.$$

Recall from §4.5.1 that fancy updates ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2} P$ are effectively regular updates with the additional ability to interact with invariants. In particular, besides the regular rules of the update $\Rrightarrow P$, they support the following two rules:

INV-OPEN-UPD

$$\frac{P * \triangleright R \vdash {}^{\mathcal{E}\setminus\mathcal{N}}{\Rrightarrow}^{\mathcal{E}\setminus\mathcal{N}} Q * \triangleright R \qquad \mathcal{N} \subseteq \mathcal{E}}{P * \boxed{R}^{\mathcal{N}} \vdash {}^{\mathcal{E}}{\Rrightarrow}^{\mathcal{E}} Q}$$

UPD-MASK-WEAKEN

$$\frac{\mathcal{E}_1 \subseteq \mathcal{E}_2 \qquad P \vdash {}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_1} P}{P \vdash {}^{\mathcal{E}_2}{\Rrightarrow}^{\mathcal{E}_2} P}$$

where INV-OPEN-UPD can be used to open an invariant as part of proving a fancy update. While in the rule GHOST-STORE, the namespace $\mathcal{N}_{\mathrm{reloc}}$ is only an implementation detail of ReLoC,[7] we emphasize the ability to open invariants around fancy updates here, because it will be crucial in our examples (in §13.1.3).

[7] ReLoC allocates an invariant named $\mathcal{N}_{\mathrm{reloc}}$ to track the current state of the ghost program and relate it to the ghost state assertions $j \Mapsto e$ and $\ell \mapsto_{\mathrm{g}} v$.

In ReLoC, to prove a relational property about programs e_1 and e_2, it suffices to prove a Hoare triple for e_1 in which the precondition has a ghost thread running e_2 in an arbitrary evaluation context K. ReLoC defines a binary relation in Iris that expresses this pattern for an arbitrary postcondition $Q : \mathit{Val} \times \mathit{Val} \to \mathit{iProp}$:

$$(e_1 \leq e_2 : Q) \triangleq \forall j, K.\ \{j \Mapsto K[e_2]\}\ e_1\ \{v_1.\ \exists v_2.\ j \Mapsto K[v_2] * Q(v_1, v_2)\}$$

The adequacy theorem of Iris (from §4.4) then ensures that, if $\mathsf{True} \vdash e_1 \leq e_2 : Q$ and e_1 terminates with value v_1, then there exists an execution of e_2 in which it terminates with a value v_2 such that $Q(v_1, v_2)$ holds.

This definition of $(\leq)$ treats the programs e_1 and e_2 asymmetrically. In particular, since the Hoare triple is about e_1, steps of e_1 get to eliminate laters. In

Value Relations

$$
\begin{aligned}
\mathcal{V}[\![\mathbb{1}]\!] &\triangleq \{(v_1, v_2) \mid v_1 = v_2 = ()\} \\
\mathcal{V}[\![\mathsf{int}]\!] &\triangleq \{(v_1, v_2) \mid \exists z \in \mathbb{Z}.\, v_1 = v_2 = z\} \\
\mathcal{V}[\![\tau \times \tau']\!] &\triangleq \left\{(v_1, v_2) \mid \exists v_a, v_a', v_b, v_b'.\, v_1 = (v_a, v_a') * v_2 = (v_b, v_b') * \mathcal{V}[\![\tau]\!](v_a, v_b) * \mathcal{V}[\![\tau']\!](v_a', v_b')\right\} \\
\mathcal{V}[\![\tau + \tau']\!] &\triangleq \left\{(v_1, v_2) \mid \exists w_1, w_2.\, v_1 = \mathsf{inj}_1(w_1) * v_2 = \mathsf{inj}_1(w_2) * \mathcal{V}[\![\tau]\!](w_1, w_2)\right\} \\
&\quad \cup \left\{(v_1, v_2) \mid \exists w_1, w_2.\, v_1 = \mathsf{inj}_2(w_1) * v_2 = \mathsf{inj}_2(w_2) * \mathcal{V}[\![\tau']\!](w_1, w_2)\right\} \\
\mathcal{V}[\![\tau \to \tau']\!] &\triangleq \{(v_1, v_2) \mid \forall w_1, w_2.\ \square(\mathcal{V}[\![\tau]\!](w_1, w_2) \mathrel{-\!\!*} (v_1\ w_1) \le (v_2\ w_2) : \mathcal{V}[\![\tau']\!])\} \\
\mathcal{V}[\![\mathsf{ref}\ \tau]\!] &\triangleq \left\{(v_1, v_2) \mid \exists \ell_1, \ell_2.\, v_1 = \ell_1 * v_2 = \ell_2 * \boxed{\exists w_1, w_2.\, \ell_1 \mapsto w_1 * \ell_2 \mapsto_{\mathsf{g}} w_2 * \mathcal{V}[\![\tau]\!](w_1, w_2)}^{\mathcal{N}.\ell_1.\ell_2}\right\}
\end{aligned}
$$

Open Expressions

$$
(\Gamma \vDash e \le_{\log} e' : \tau) \triangleq \forall \gamma_1, \gamma_2.\ (\gamma_1, \gamma_2) \in \mathcal{G}[\![\Gamma]\!] \vdash \gamma_1(e_1) \le \gamma_2(e_2) : \mathcal{V}[\![\tau]\!]
$$
$$
\textit{where}\ \ \mathcal{G}[\![\Gamma]\!] \triangleq \{(\gamma_1, \gamma_2) \mid \textstyle\mathop{\ast}_{x:\tau\in\Gamma}\ (\gamma_1(x), \gamma_2(x)) \in \mathcal{V}[\![\tau]\!]\}
$$

Figure 13.2: Excerpt of the logical relation of ReLoC.

contrast, e_2 is just a ghost program, so as it is executed, no laters are eliminated. Typically, this asymmetry is not a problem, because we are reasoning about the two programs "in sync": by taking steps of e_1 at the same time as we perform steps of e_2, we can use the physical steps of e_1 to eliminate laters needed for reasoning about e_2. But for reordering, it will become an issue (see §13.1.5).

Having defined the binary relation $e_1 \le e_2 : Q$, defining the logical relation $\Gamma \vDash e_1 \le_{\log} e_2 : \tau$ is relatively straightforward, depicted in Fig. 13.2. First, we define a type interpretation $\mathcal{V}[\![\_]\!] : \mathit{Type} \to (\mathit{Val} \times \mathit{Val}) \to \mathit{iProp}$, which maps every type τ to a persistent Iris relation on values $\mathcal{V}[\![\tau]\!]$. Then, we define $(\le_{\log})$ by lifting $(\le)$ to open expressions.[8] To simplify the explanation here, we leave out the details of how this approach scales to polymorphic and recursive types.

[8] Here, $\gamma(e)$ denotes the result of substituting the free variables in e according to the substitution γ.

In the definition of $\mathcal{V}[\![\_]\!]$, the interesting cases are $\tau \to \tau'$ and $\mathsf{ref}\ \tau$. The former says that two values are related at type $\tau \to \tau'$ if, whenever they are applied to values related at type τ, the resulting application expressions are related at the interpretation of τ'. In this case, we use Iris's *persistency* modality "$\square$" (§3.3) to ensure values of type $\tau \to \tau'$ can be used multiple times.

For $\mathsf{ref}\ \tau$, the relation says that the two values must be locations, and we use an Iris invariant that requires the two locations to always point to values that are related at type τ. (As in §8.3, the invariant here is implicitly making use of Iris's step-indexing, allowing us to avoid the usual circularity issues that arise in defining logical relations for systems with higher-order mutable state.[9])

[9] Ahmed, "Semantics of types for mutable state", 2004 [Ahm04]; Birkedal et al., "Step-indexed Kripke models over recursive worlds", 2011 [Bir+11].

The logical relation has the following two key properties:

Theorem 65 (Soundness). *If* $\Gamma \vDash e_1 \le_{\log} e_2 : \tau$*, then* $\Gamma \vdash e_1 \le_{\mathrm{ctx}} e_2 : \tau$*.*

Theorem 66 (Fundamental Property). *If* $\Gamma \vdash e : \tau$*, then* $\Gamma \vDash e \le_{\log} e : \tau$*.*

The soundness theorem ensures that the logical relation is useful for proving contextual refinements, and it follows from the adequacy of Iris. Meanwhile, the fundamental property lets us automatically deduce that a syntactically well-typed term is logically related to itself. This theorem is proven by showing that the logical relation is a congruence with respect to all typing rules.

13.1.2 Reorderability Extension

Next, we add reorderability. The type system of ReLoC is not rich enough to state that a function is reorderable. To reason about reorderings, we extend the type system with a new type $\tau_1 \rightarrow_{\mathsf{re}} \tau_2$ of reorderable functions. The typing rule for this type uses a new relation, $\Gamma \vdash^{\mathsf{re}} e : \tau$, which implies that e is a *reorderable* expression of type τ. This judgment is a restriction of the standard typing judgment $\vdash$ that removes the rules for operations that have side effects. We then extend the standard typing judgment $\vdash$ with two new rules for introducing and eliminating terms of type $\rightarrow_{\mathsf{re}}$:

$$\frac{\Gamma, x : \tau_1, f : \tau_1 \rightarrow_{\mathsf{re}} \tau_2 \vdash^{\mathsf{re}} e : \tau_2}{\Gamma \vdash (\mathsf{fix}\, f\, x.\, e) : \tau_1 \rightarrow_{\mathsf{re}} \tau_2} \qquad \frac{\Gamma \vdash f : \tau_1 \rightarrow_{\mathsf{re}} \tau_2 \qquad \Gamma \vdash e : \tau_1}{\Gamma \vdash f\ e : \tau_2}$$

The fragment $\vdash^{\mathsf{re}}$ is fairly limited, since expressions may not contain instructions with side effects. However, it is possible to bend this limitation. In the following, we will develop a logical relation for $\vdash^{\mathsf{re}}$, which admits additional terms that cannot be typed syntactically in the side-effect free fragment $\vdash^{\mathsf{re}}$, but are *semantically reorderable.* For example, $\lambda\_.\ \mathsf{let}\ r = \mathsf{ref}(41)\ \mathsf{in}\ !r + 1$ and `then` are semantically reorderable even though they have side effects. To extend the logical relation to support the reorderability type judgment ($\vdash^{\mathsf{re}}$) and reorderable function type ($\rightarrow_{\mathsf{re}}$), we take inspiration from Timany et al.[10] to define a reorderable form of $e_1 \leq e_2 : Q$ as follows:

[10] Timany et al., "A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of runST", 2018 [Tim+18].

$$(e_1 \leq^{\mathsf{re}} e_2 : Q) \triangleq \{\mathsf{True}\}\ e_1\ \{v_1.\ \exists v_2.\ Q(v_1, v_2) * e_2 \rightsquigarrow_{\mathrm{ghost}} v_2\}$$
$$\textit{where}\ \ e \rightsquigarrow_{\mathrm{ghost}} e' \triangleq \forall j, K.\ j \Rrightarrow K[e] \mathrel{-\!\!*} {}^{\top}\!\!\Rrightarrow_{\mathsf{le}}^{\top}\ j \Rrightarrow K[e']$$

The key difference between $\leq$ and $\leq^{\mathsf{re}}$ is that in the latter, the execution of the ghost program is moved entirely to the postcondition, as captured by the $\rightsquigarrow_{\mathrm{ghost}}$ assertion.[11] That is, instead of executing the ghost program and the implementation "in sync" (as with the usual $\leq$ in ReLoC), we wait to run e_2 until *after* e_1 finishes running. This means e_1 executes "independently" of e_2, and subsequently e_2 executes independently of e_1. Disentangling e_1 and e_2 makes their executions reorderable, as we will see below. However, it also means physical steps of e_1 no longer directly eliminate laters that come up when reasoning about e_2. This could pose a problem if e_2 needs to take non-timeless resources out of an invariant. Fortunately, because our notion of ($\rightsquigarrow_{\mathrm{ghost}}$) uses a *later elimination update*[12] "${}^{\top}\!\!\Rrightarrow_{\mathsf{le}}^{\top}$" instead of a standard update "${}^{\top}\!\!\Rrightarrow^{\top}$", we can spend later credits generated by e_1 as we execute steps of e_2.

[11] The ghost execution $e \rightsquigarrow_{\mathrm{ghost}} e'$ is similar to the ghost executions in §8.1.2. However, the ghost executions here only have to be re-orderable and not repeatable, so they do not occur underneath a persistency modality.

[12] To support fancy updates (*i.e.,* mask-changing updates), we define a version ${}^{\mathcal{E}_1}\!\!\Rrightarrow_{\mathsf{le}}^{\mathcal{E}_2}$ of the fancy update ${}^{\mathcal{E}_1}\!\!\Rrightarrow^{\mathcal{E}_2}$ from §4.5.1 that uses the later elimination update $\Rrightarrow_{\mathsf{le}}$ in place of the regular one $\Rrightarrow$. It has effectively the same rules as the original with the additional ability to spend later credits to eliminate laters similar to LE-UPD-LATER.

We integrate reorderability into ReLoC by defining

$$\mathcal{V}[\![\tau \rightarrow_{\mathsf{re}} \tau']\!] \triangleq \{(v_1, v_2) \mid \forall u_1, u_2.\ \Box(\mathcal{V}[\![\tau]\!](u_1, u_2) \mathrel{-\!\!*} (v_1\ u_1) \leq^{\mathsf{re}} (v_2\ u_2) : \mathcal{V}[\![\tau']\!])\}$$

and lifting $e_1 \leq^{\mathsf{re}} e_2 : Q$ to a version on open expressions $\Gamma \vDash e_1 \leq^{\mathsf{re}}_{\mathsf{log}} e_2 : \tau$ analogous to ($\leq_{\mathsf{log}}$). The new relation $\Gamma \vDash e_1 \leq^{\mathsf{re}}_{\mathsf{log}} e_2 : \tau$ ties neatly in with the standard ReLoC setup (from §13.1.1):

Lemma 67.

1. *If* $\Gamma \vdash^{\mathsf{re}} e : \tau$, *then* $\Gamma \vDash e \leq^{\mathsf{re}}_{\mathsf{log}} e : \tau$.

2. *If* $\Gamma \vDash e_1 \leq^{\mathsf{re}}_{\mathsf{log}} e_2 : \tau$, *then* $\Gamma \vDash e_1 \leq_{\mathsf{log}} e_2 : \tau$.

Moreover, reorderability ($\leq^{\text{re}}_{\text{log}}$) is strong enough to show the reorderings that we are after (Lemma 68 and Lemma 69). The proofs of both statements use the fact that a reorderable expression can run *unconditionally* on the left (since the precondition of ($\leq^{\text{re}}_{\text{log}}$) is True) and that we can then delay its execution on the right arbitrarily. We start with the sequencing reordering:

Lemma 68 (Sequencing Reordering)**.**
If $\Gamma \vDash e_1 \leq_{\text{log}} e_1' : \mathbb{1}$ *and* $\Gamma \vDash e_2 \leq^{\text{re}}_{\text{log}} e_2' : \mathbb{1}$*, then* $\Gamma \vDash e_2; e_1 \leq_{\text{log}} e_1'; e_2' : \mathbb{1}$.

Proof. We focus on the case where e_1, e_1', e_2, and e_2' are closed (*i.e.,* $\Gamma = \emptyset$). The general case follows analogously. We show

$$e_2; e_1 \leq e_1'; e_2' : \mathcal{V}[\![\mathbb{1}]\!]$$

under the assumptions $e_1 \leq e_1' : \mathcal{V}[\![\mathbb{1}]\!]$ and $e_2 \leq^{\text{re}} e_2' : \mathcal{V}[\![\mathbb{1}]\!]$. Thus, unfolding the definition of ($\leq$), we have to show:

$$\{j \Rrightarrow K[e_1'; e_2']\}\ e_2; e_1\ \{v_1.\, \exists v_2.\, j \Rrightarrow K[v_2] * (v_1, v_2) \in \mathcal{V}[\![\mathbb{1}]\!]\}$$

Here, we benefit from the reorderability of e_2. Note that the ghost program executes e_1' first and the actual program executes e_2 first. However, since e_2 is reorderable, its verification does not require any ghost code resource in its precondition. Instead, it produces the *delayed* ghost execution $e_2' \rightsquigarrow_{\text{ghost}} ()$ in its postcondition (using the fact that all values in $\mathcal{V}[\![\mathbb{1}]\!]$ are $()$). Concretely, binding on e_2 and using $e_2 \leq^{\text{re}} e_2' : \mathcal{V}[\![\mathbb{1}]\!]$, we are left to prove:

$$\begin{array}{l}\{j \Rrightarrow K[e_1'; e_2'] * e_2' \rightsquigarrow_{\text{ghost}} ()\} \\ \quad (); e_1 \\ \{v_1.\, \exists v_2.\, j \Rrightarrow K[v_2] * (v_1, v_2) \in \mathcal{V}[\![\mathbb{1}]\!]\}\end{array}$$

After executing "();" in the actual program, we can then execute e_1 in the actual program and e_1' in the ghost program "in sync". Concretely, using the assumption $e_1 \leq e_1' : \mathcal{V}[\![\mathbb{1}]\!]$ for $K' \triangleq K[\bullet; e_2']$, we are left to prove:

$$\{j \Rrightarrow K[(); e_2'] * e_2' \rightsquigarrow_{\text{ghost}} ()\}\ ()\ \{v_1.\, \exists v_2.\, j \Rrightarrow K[v_2] * (v_1, v_2) \in \mathcal{V}[\![\mathbb{1}]\!]\}$$

We can execute "();" in the ghost program, and are left to execute e_2' in the ghost program. Fortunately, since we have the ghost execution $e_2' \rightsquigarrow_{\text{ghost}} ()$, we can do so, leaving us to prove:

$$\{{}^{\top}\!\Rrightarrow^{\top}_{\text{le}} j \Rrightarrow K[()]\}\ ()\ \{v_1.\, \exists v_2.\, j \Rrightarrow K[v_2] * (v_1, v_2) \in \mathcal{V}[\![\mathbb{1}]\!]\}$$

We can execute the update in the precondition of the Hoare triple as usual (see hoare-le-upd-exec in Fig. 12.1), which concludes the proof. □

Similarly, we can prove the following lemma for reordering with a parallel composition, because e_2 can be executed independently in the actual program from e_2' in the ghost program:

Lemma 69.
If $\Gamma \vDash e_1 \leq_{\text{log}} e_1' : \tau_1$ *and* $\Gamma \vDash e_2 \leq^{\text{re}}_{\text{log}} e_2' : \tau_2$*, then* $\Gamma \vDash e_2 \,||\, e_1 \leq_{\text{log}} (e_2', e_1') : \tau_2 \times \tau_1$.

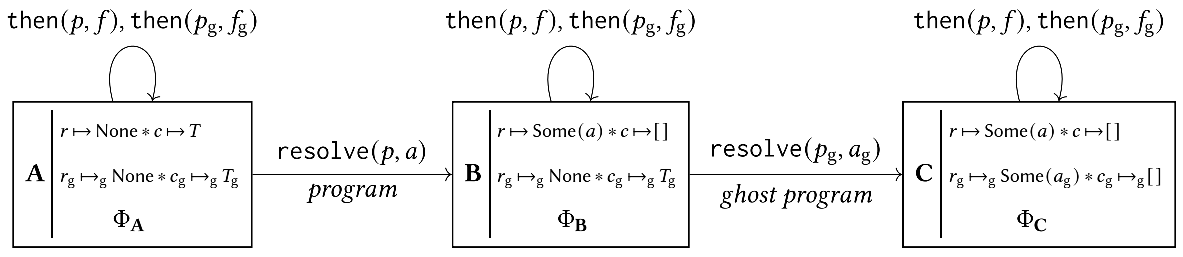

then(p, f), then(p_g, f_g)
then(p, f), then(p_g, f_g)
then(p, f), then(p_g, f_g)
A
r ↦ None ∗ c ↦ T
r_g ↦_g None ∗ c_g ↦_g T_g
Φ_A
resolve(p, a)
program
B
r ↦ Some(a) ∗ c ↦ [ ]
r_g ↦_g None ∗ c_g ↦_g T_g
Φ_B
resolve(p_g, a_g)
ghost program
C
r ↦ Some(a) ∗ c ↦ [ ]
r_g ↦_g Some(a_g) ∗ c_g ↦_g [ ]
Φ_C

Figure 13.3: Transition system describing the different states a promise can be in.

13.1.3 Promises with Later Credits

Equipped with the notion of reorderability, we return to our motivating example: *reordering promise operations*. Our main result for promises will be that their operations are in the logical relation, as stated in Lemma 70.

Lemma 70 (Promise Typing).

1. $\vDash \texttt{promise} \leq_{\text{log}} \texttt{promise} : \mathbb{1} \to \text{pr}(\tau)$
2. $\vDash \texttt{resolve} \leq_{\text{log}}^{\text{re}} \texttt{resolve} : \text{pr}(\tau) \times \tau \to_{\text{re}} \mathbb{1}$
3. $\vDash \texttt{then} \leq_{\text{log}}^{\text{re}} \texttt{then} : \text{pr}(\tau) \times (\tau \to_{\text{re}} \mathbb{1}) \to_{\text{re}} \mathbb{1}$

The proof of this lemma is *challenging* in terms of reasoning about different interleavings of the promise operations, but *simple* in terms of the later credits reasoning. Since we are mainly interested in later credits, we will give a detailed description of the use of later credits (in §13.1.6), and only a high-level description of the rest of the proof (in §13.1.4 and §13.1.5). Let us start with an example of the kind of reorderings that we can prove with this lemma and Lemma 68:

Corollary 71.

$$\frac{\Gamma \vDash e \leq_{\text{log}} e : \mathbb{1} \qquad p : \text{pr}(\tau) \in \Gamma \qquad f : \tau \to_{\text{re}} \mathbb{1} \in \Gamma}{\Gamma \vDash \texttt{then}(p, f); e \leq_{\text{log}} e; \texttt{then}(p, f) : \mathbb{1}}$$

There are two things to note about this corollary. First, when we prove that an expression like then is reorderable, then we can move its execution *earlier*. This includes moving it across an arbitrary expression e, which could include additional calls to then or to resolve. Second, the corollary demonstrates the higher-order nature of the promise operations: then takes an arbitrary reorderable function f as its argument. In particular, f could resolve another promise q or (reentrantly) attach an additional continuation to p, since the operations then and resolve are themselves reorderable.

13.1.4 Promise Extension

Before we can prove Lemma 70, we have to extend the type interpretation $\mathcal{V}[\![\_]\!]$ to include promises $\text{pr}(\tau)$. The definition of $\mathcal{V}[\![\text{pr}(\tau)]\!]$ consists of a lock and an invariant, which together encode a transition system for each promise. We focus on the transition system,[13] depicted in Fig. 13.3. Initially, in state **A**, the promise is unresolved in the program and the ghost program (marked by the subscript g)—the reference r is None, and the reference c is amassing

[13] The full definition can be found in the appendix of the later credits paper. [Spi+22b]

continuations. The proposition Φ in each state stores some additional ownership that we need for the verification (*e.g.*, $\Phi_{\mathbf{A}}$ stores $(f, f_g) \in \mathcal{V}[\![\tau \rightarrow_{\text{re}} \mathbb{1}]\!]$ for each $f \in T$). We will return to Φ in §13.1.5. From state **A**, we transition to state **B** when `resolve`(p, a) is executed in the program. We store a in the promise in the program but not yet in the ghost program, and we execute the continuations in T. From state **B**, we transition to state **C** when `resolve`(p_g, a_g) is executed in the ghost program. We store a_g in the reference and execute the continuations in T_g. If `then`(p, f) (resp. `then`(p_g, f_g)) is executed in either the program or the ghost program, we do not change the state in the transition system. Depending on whether the promise has been resolved yet or not, the function is either stored in the continuation list (*i.e.*, in T or T_g) or directly executed.

Notably, there is no state **B'** in the transition system, where `resolve`(p_g, a_g) has been executed first in the ghost program. The reason is that if we look at the definition of $e \preceq^{\text{re}} e' : Q$, then e is always executed in the program first before e' is executed in the ghost program. (For the same reason, `then`(p, f) always executes in the actual program first, before the corresponding `then`(p_g, f_g) executes in the ghost program. As a result, it is impossible in state **A** that f_g is in T_g but the corresponding continuation f is not in T.)

13.1.5 The Continuation Exchange

Let us return to Lemma 70, specifically `resolve` and `then`. (The `promise` operation is straightforward.) We will focus on the "continuation exchange", the key step in the proof where later credits will become necessary (in §13.1.6). In the continuation exchange, the continuation f is executed by one operation in the program and by the other in the ghost program, requiring us pass ownership of the execution of the continuation from one operation to the other. This happens, for example, in the following reordering, a corollary of Lemma 70:

$$p : \mathsf{pr}(\tau), f : \tau \rightarrow_{\text{re}} \mathbb{1}, a : \tau \vDash \begin{array}{l}\texttt{then}(p, f);\\ \texttt{resolve}(p, a)\end{array} \preceq_{\log} \begin{array}{l}\texttt{resolve}(p, a);\\ \texttt{then}(p, f)\end{array} : \mathbb{1}$$

Here, f is executed by `resolve` in the program but by `then` in the ghost program. When `resolve` executes in the ghost program, it only stores the value a inside the promise without executing f.

The resolve case. Let us start with the `resolve` case of Lemma 70. We have to show that for $(p, p_g) \in \mathcal{V}[\![\mathsf{pr}(\tau)]\!]$ and $(a, a_g) \in \mathcal{V}[\![\tau]\!]$:

$$\texttt{resolve}(p, a) \preceq^{\text{re}} \texttt{resolve}(p_g, a_g) : \mathcal{V}[\![\mathbb{1}]\!]$$

We focus on the most interesting case, which is when `resolve` executes a continuation in the program but—as in the example above—stores the value in the promise when executed in the ghost program. In this case, for the execution of `resolve`(p, a) in the program, we start in state **A** and we find a continuation $f \in T$. (For the sake of simplicity, let us focus on a single continuation in this explanation.) In this state, albeit not shown in Fig. 13.3, $\Phi_{\mathbf{A}}$ contains $(f, f_g) \in \mathcal{V}[\![\tau \rightarrow_{\text{re}} \mathbb{1}]\!]$. (We will put $(f, f_g) \in \mathcal{V}[\![\tau \rightarrow_{\text{re}} \mathbb{1}]\!]$ into $\Phi_{\mathbf{A}}$ in the `then` case.) We store a in r, transition to state **B**, and proceed to execute $f(a)$ in the program using $(f, f_g) \in \mathcal{V}[\![\tau \rightarrow_{\text{re}} \mathbb{1}]\!]$. Afterwards, having just executed $f(a)$ in the program, we now own the ghost execution $f_g(a_g) \rightsquigarrow_{\text{ghost}} ()$ from $(f, f_g) \in \mathcal{V}[\![\tau \rightarrow_{\text{re}} \mathbb{1}]\!]$ (analogously to the proof of Lemma 68).

We use it on the side of the ghost program. For the ghost program, we must prove $\texttt{resolve}(p_\text{g}, a_\text{g}) \rightsquigarrow_\text{ghost} ()$. Recall that we are considering the interesting case where, in the ghost program, p_g has no continuation attached to it yet. Thus, we are still in state **B** and T_g is currently empty. The ghost program stores a_g in r_g, and we advance in the transition system to state **C**. Since there are no continuations attached yet, we can prove $\texttt{resolve}(p_\text{g}, a_\text{g}) \rightsquigarrow_\text{ghost} ()$ without needing $f_\text{g}(a_\text{g}) \rightsquigarrow_\text{ghost} ()$. We put the ghost execution $f_\text{g}(a_\text{g}) \rightsquigarrow_\text{ghost} ()$ into $\Phi_\mathbf{C}$ in the transition system (to use it in the then case).

The then case. Let us use now turn to the then case of Lemma 70. We have to show that for $(p, p_\text{g}) \in \mathcal{V}[\![\text{pr}(\tau)]\!]$ and $(f, f_\text{g}) \in \mathcal{V}[\![\tau \rightarrow_\text{re} \mathbb{1}]\!]$:

$$\texttt{then}(p, f) \leq^\text{re} \texttt{then}(p_\text{g}, f_\text{g}) : \mathcal{V}[\![\mathbb{1}]\!]$$

We again focus on the most interesting case, which is when then stores f in the promise in the program but—as in the example—executes f_g in the ghost program. In this case, for the execution of $\texttt{then}(p, f)$ in the program, we are in state **A**. We store f in T, and we store $(f, f_\text{g}) \in \mathcal{V}[\![\tau \rightarrow_\text{re} \mathbb{1}]\!]$ in $\Phi_\mathbf{A}$. (This matches the resolve case, where we take $(f, f_\text{g}) \in \mathcal{V}[\![\tau \rightarrow_\text{re} \mathbb{1}]\!]$ out of $\Phi_\mathbf{A}$.)

On the side of the ghost program, when $\texttt{then}(p_\text{g}, f_\text{g})$ is executed, we have to prove $\texttt{then}(p_\text{g}, f_\text{g}) \rightsquigarrow_\text{ghost} ()$. Since we are focusing on the interesting case where then will execute the continuation, we are in state **C**. This means p_g already stores a value a_g, and $\texttt{then}(p_\text{g}, f_\text{g})$ will execute $f_\text{g}(a_\text{g})$. Thus, as part of proving $\texttt{then}(p_\text{g}, f_\text{g}) \rightsquigarrow_\text{ghost} ()$, we have to establish the ghost execution $f_\text{g}(a_\text{g}) \rightsquigarrow_\text{ghost} ()$. Fortunately, we have put the ghost execution $f_\text{g}(a_\text{g}) \rightsquigarrow_\text{ghost} ()$ into $\Phi_\mathbf{C}$ when resolving the promise (see the previous case). We now want to use it to finish the proof. This *almost* works, but a "$\triangleright$" gets in the way …

13.1.6 Using Later Credits

In the last step, we run into a "$\triangleright$" when we try to use the $f_\text{g}(a_\text{g}) \rightsquigarrow_\text{ghost} ()$ stored in $\Phi_\mathbf{C}$. That is, recall from the beginning of §13.1.4 that the transition system is encoded in an *invariant*, which is shared between the different promise operations via the interpretation of the promise type $\mathcal{V}[\![\text{pr}(\tau)]\!]$. We will denote the invariant with $\boxed{\text{TS}}^{\mathcal{N}}$ in the following. In short, when we open the invariant in the final step, we only get $\triangleright f_\text{g}(a_\text{g}) \rightsquigarrow_\text{ghost} ()$ out, but we need $f_\text{g}(a_\text{g}) \rightsquigarrow_\text{ghost} ()$ to finish the proof, so (without later credits) we are stuck.[14]

Let us zoom in on that last proof step. Formally, at that point in the proof, we have to show:

$$\boxed{\text{TS}}^{\mathcal{N}} \vdash \texttt{then}(p_\text{g}, f_\text{g}) \rightsquigarrow_\text{ghost} ()$$

and since the promise is resolved in the ghost program at that point, we will be in state **C**. We may open $\boxed{\text{TS}}^{\mathcal{N}}$ because of the fancy update in $(\rightsquigarrow_\text{ghost})$, but when we open $\boxed{\text{TS}}^{\mathcal{N}}$, we will only get access to $\triangleright\,\text{TS}$ (see INV-OPEN-UPD in §13.1.1), which effectively means we only get $\triangleright f_\text{g}(a_\text{g}) \rightsquigarrow_\text{ghost} ()$. To prove $\texttt{then}(p_\text{g}, f_\text{g}) \rightsquigarrow_\text{ghost} ()$, we need to eliminate the later, but there are no *program* steps around to justify an elimination—only ghost program steps (which do not allow later elimination). We are stuck![15]

With later credits, the solution is simple. When we reason about then (*i.e.*, when we prove $\vDash \texttt{then} \leq^\text{re}_\text{log} \texttt{then} : \text{pr}(\tau) \times (\tau \rightarrow_\text{re} \mathbb{1}) \rightarrow_\text{re} \mathbb{1}$), the execution *in the program* has plenty of steps that generate credits that we do not need

[14] The same issue does not arise for the symmetrical side, where we take out $(f, f_\text{g}) \in \mathcal{V}[\![\tau \rightarrow_\text{re} \mathbb{1}]\!]$ in the resolve-case because steps of the program ordinarily can eliminate later modalities.

[15] One may wonder if we can get $f_\text{g}(a_\text{g}) \rightsquigarrow_\text{ghost} ()$ out of the invariant during execution of $\texttt{then}(p, f)$ in the program, so that there are still program steps around. The answer is no, because during that execution, the promise may still be in state **A** (*e.g.*, in the example in §13.1.5). Thus, f will be in the list, not executed, and $f_\text{g}(a_\text{g}) \rightsquigarrow_\text{ghost} ()$ is not yet available, since it only enters the invariant in state **C**.

(*e.g.,* the initial β-reduction step of $\texttt{then}(p, f)$). We can frame one of these credits $£1$ to the postcondition, such that it becomes available when we need to prove $\texttt{then}(p_g, f_g) \rightsquigarrow_{\text{ghost}} ()$. That is, instead of $\boxed{\text{TS}}^{\mathcal{N}} \vdash \texttt{then}(p_g, f_g) \rightsquigarrow_{\text{ghost}} ()$, we now prove $\boxed{\text{TS}}^{\mathcal{N}} * £1 \vdash \texttt{then}(p_g, f_g) \rightsquigarrow_{\text{ghost}} ()$. Thus, when we open $\boxed{\text{TS}}^{\mathcal{N}}$ this time, we can use the later credit $£1$ to eliminate the later and obtain $f_g(a_g) \rightsquigarrow_{\text{ghost}} ()$, finishing the proof.

13.2 Later Credits for Logical Atomicity

In this section, we demonstrate another use of later credits, namely for eliminating a lingering pain point in one of Iris's specialties: *logical atomicity proofs.*[16] Inspired originally by the TaDA logic,[17] logical atomicity is Iris's technique for proving functional correctness of (fine-grained) concurrent data structures. Akin to the standard notion of linearizability,[18] a logically atomic specification of a concurrent operation says that the operation *appears* to take effect atomically, even though it may actually take multiple physical steps. As a consequence, clients can reason about logically atomic operations (almost) as if they were physically atomic instructions—in particular, they can open invariants around them.

[16] Jung et al., "Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning", 2015 [Jun+15].

[17] Rocha Pinto, Dinsdale-Young, and Gardner, "TaDA: A logic for time and data abstraction", 2014 [RDG14].

[18] Herlihy and Wing, "Linearizability: A correctness condition for concurrent objects", 1990 [HW90].

Logical atomicity has been successfully applied to a variety of challenging concurrent data structures.[19] Unfortunately, in verifying logical atomicity for data structures that exhibit a common pattern known as "helping", step-indexing has always caused trouble. "Helping" refers to the situation where one thread helps another thread complete its operation. In previous work, proving logical atomicity for data structures with helping necessitated the use of an "ugly" workaround (to quote its inventor[20]) called "make-laterable", which made logical atomicity harder to prove and harder to use for clients.

[19] Jung et al., "Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning", 2015 [Jun+15]; Jung et al., "The future is ours: Prophecy variables in separation logic", 2020 [Jun+20]; Birkedal et al., "Theorems for free from separation logic specifications", 2021 [Bir+21]; Frumin, Krebbers, and Birkedal, "ReLoC Reloaded: A mechanized relational logic for fine-grained concurrency and logical atomicity", 2021 [FKB21b]; Carbonneaux et al., "Applying formal verification to microkernel IPC at Meta", 2022 [Car+22].

[20] Jung, *Logical atomicity in Iris: The good, the bad, and the ugly*, 2019 [Jun19].

With later credits, we can avoid the need for "make-laterable" entirely, along with its limitations. To explain how, we will use a concrete example of a concurrent data structure that involves helping: *a counter with a backup.* We explain the counter (in §13.2.1), the intuitive argument for proving its logical atomicity (in §13.2.2), the reason we cannot implement that intuitive proof argument with "make-laterable" (in §13.2.3), and finally how later credits save the day (in §13.2.4).

13.2.1 A Counter with a Backup

Our motivating example is a counter with a backup (Fig. 13.4). This counter is basically a regular monotone counter (as described in §3.6) with methods `incr` to increment the counter by 1 and `get` to get the current value of the counter. But there is a twist: the value of the counter is stored in two locations—the *primary* p and the *backup* b—and these two can get out of sync: the operation `incr` eagerly updates the primary p, but leaves updating the backup to *a background thread* (`bg_thread`). Clients can directly access the backup b through a third operation, `get_backup`, so it may seem like they can observe the difference between the primary and the backup. What makes this counter interesting is that they *cannot*, because `incr` and `get` wait for the backup to catch up.

Implementation

$$\begin{aligned}
\texttt{new}() &\triangleq \mathsf{let}\ (b, p) = (\mathsf{ref}(0), \mathsf{ref}(0))\ \mathsf{in}\ \mathsf{fork}\{\texttt{bg\_thread}(b, p)\}; (b, p) \\
\texttt{incr}(b, p) &\triangleq \mathsf{let}\ n = \mathsf{FAA}(p, 1)\ \mathsf{in}\ \texttt{await\_backup}(b, n+1); n \\
\texttt{get}(b, p) &\triangleq \mathsf{let}\ n = {!}\, p\ \mathsf{in}\ \texttt{await\_backup}(b, n); n \\
\texttt{get\_backup}(b, p) &\triangleq {!}\, b
\end{aligned}$$

Helper Functions

$$\begin{aligned}
\texttt{bg\_thread}(b, p) &\triangleq \mathsf{let}\ n = {!}\, p\ \mathsf{in}\ b := n; \texttt{bg\_thread}(b, p) &&\text{// copy primary to backup, in a loop} \\
\texttt{await\_backup}(b, n) &\triangleq \mathsf{if}\ {!}\, b < n\ \mathsf{then}\ \texttt{await\_backup}(b, n)\ \mathsf{else}\ () &&\text{// loop until } {!}\, b \text{ reaches } n
\end{aligned}$$

Specification

$$\begin{aligned}
&\vdash \{\mathsf{True}\}\ \texttt{new}()\ \{c.\ \exists \gamma.\ \mathsf{is\_counter}^{\mathcal{N}}_{\gamma}(c) * \mathsf{value}_{\gamma}(0)\} \\
\mathsf{is\_counter}^{\mathcal{N}}_{\gamma}(c) &\vdash \langle n.\ \mathsf{value}_{\gamma}(n)\rangle\ \texttt{incr}(c)\ \langle m.\ m = n * \mathsf{value}_{\gamma}(n+1)\rangle_{\mathcal{N}} \\
\mathsf{is\_counter}^{\mathcal{N}}_{\gamma}(c) &\vdash \langle n.\ \mathsf{value}_{\gamma}(n)\rangle\ \texttt{get}(c)\ \langle m.\ m = n * \mathsf{value}_{\gamma}(n)\rangle_{\mathcal{N}} \\
\mathsf{is\_counter}^{\mathcal{N}}_{\gamma}(c) &\vdash \langle n.\ \mathsf{value}_{\gamma}(n)\rangle\ \texttt{get\_backup}(c)\ \langle m.\ m = n * \mathsf{value}_{\gamma}(n)\rangle_{\mathcal{N}}
\end{aligned}$$

Figure 13.4: Counter with a backup.

To understand this counter better, let us consider a concrete example:

$$\begin{aligned}
e_{\text{count}} \triangleq\ &\mathsf{let}\ c = \texttt{new}()\ \mathsf{in}\ \mathsf{fork}\{\texttt{incr}(c)\}; \\
&\mathsf{let}\ x = \texttt{get}(c)\ \mathsf{in}\ \mathsf{let}\ y = \texttt{get\_backup}(c)\ \mathsf{in}\ (x, y)
\end{aligned}$$

Depending on when the increment occurs and when the background thread updates the backup b, this expression has three possible values: $(0, 0)$, $(0, 1)$, and $(1, 1)$. One value that it does not have is $(1, 0)$. The outcome $(1, 0)$ is impossible, even though `get` reads the primary p and `get_backup` reads the backup b. The reason is that `get` *waits* for the backup to catch up before returning its result, so we can be sure that any subsequent `get_backup` cannot read "outdated" values.

The counter with a backup is clearly a contrived example. However, it originates from an issue arising in real data structures[21] that need to be durable. For example, a key-value server will store the current mapping of keys to values on disk, but also keep an in-memory copy of that mapping to quickly reply to read requests. Updating the data on disk is inefficient, so a background thread batches concurrent writes to be able to write them to disk in one go. At any time, the system can crash and the in-memory copy disappears; after reboot and recovery, the state of the key-value server is restored from what was stored on disk at the time of the crash. Since the in-memory copy can be lost, the operations working on it need to wait for their changes to become permanent, so they avoid returning data that is later lost in a crash.[22] To avoid all the complexities of crashes and durable state, we have condensed this problem down to its core. The key-value store is replaced by a single counter, the durable disk is replaced by a second copy of the counter in memory, and we use `get_backup` to model the fact that this second copy is observable by clients through crashes.

[21] In fact, after the later credits paper was published, Chang et al. [Cha+23] have used the helping pattern described in the following, "unsolicited helping", to verify a high-performance transaction library using multi-version concurrency control.

[22] The reader may wonder what the point of the in-memory copy is when accesses have to wait for the updates to be written to disk anyway. The answer is that the in-memory copy can be used to quickly respond to read requests for keys that have no pending changes. In these cases, the value can be returned instantly instead of having to load it from disk.

Helping. The most interesting thing about the counter with a backup is the interaction between the background thread and the operations `get` and `incr`. Take `incr` for example. `incr` modifies the primary p, but its effect only becomes observable (through the counter operations) once the background thread updates the backup b. In other words, to complete its action, `incr` needs assistance from the background thread, which is typically called "helping". In general, helping means that the point in time when the action of one operation appears to clients to "take effect"—also known as the *linearization point* of the operation—is actually performed by *another* "helping" thread. For `incr`, the linearization point is the update of b to n in the background thread, because that is when the new counter value actually becomes observable to other `get` and `get_backup` operations. For `get`, in cases where the operation observes the primary p to be larger than the backup b, the linearization point is *also* the update of b in the background thread, because only then can other `get` and `get_backup` operations also observe the new value (see the e_{count} example).

What is particularly interesting about the helping in this example—and what makes it challenging to verify in existing Iris—is that at the point when the background thread updates b, it may have to help (an arbitrary number of) `get` operations complete, but it does not know which operations those are in advance because the `get` operations do nothing to explicitly communicate their need to be helped. In fact, the background thread may have to help `get` operations which only began *immediately before* the update of b to n. We call this phenomenon *unsolicited* helping, in contrast to the *solicited* helping that occurs in the `incr` operation (since the latter communicates explicitly to the background thread by incrementing p). As we will soon see, helping (especially unsolicited helping) makes it difficult to verify data structures like this one in existing Iris, but later credits offer a simpler way.

13.2.2 Logical Atomicity

Let us attempt to verify the counter. We want to prove the standard specification of a logically atomic counter, meaning `get` (and `get_backup`) observe the value of the counter at the linearization point and return it, while `incr` increments the value at the linearization point and returns the old value. In the language of logical atomicity, we express this with the specification shown in Fig. 13.4. Except for the initialization (where atomicity does not matter), the specification consists of several *logically atomic triples* $\langle x.\, P(x)\rangle\, f(a)\, \langle y.\, Q(x,y)\rangle_{\mathcal{E}}$. These are special Hoare triples that describe the atomic action of f at the linearization point. For example, for `incr`, the logically atomic specification is

$$\langle n.\, \mathsf{value}_\gamma(n)\rangle\ \mathtt{incr}(c)\ \langle m.\, m = n * \mathsf{value}_\gamma(n+1)\rangle_{\mathcal{N}}$$

indicating that `incr` updates the value of the counter (identified by the logical name γ) from n to $n+1$. Here, the number n is supposed to be the value of the counter *at the linearization point*. Since the number n is typically not known before the execution of `incr` (and potentially changes during its execution), logically atomic triples have an additional binder "n." in their precondition. This binder can relate the value of n at the linearization point to the result of the triple "m." in the postcondition. In the case of `incr`, we thus know that it returns the value of the counter at the linearization point similar to a fetch-and-add. The rest of the specification is bookkeeping: we keep some state of the counter in an invariant $\mathsf{is\_counter}^{\mathcal{N}}_{\gamma}(c)$, which means that the proof of `incr` needs access to invariant namespace $\mathcal{N}$ and that is reflected in the specification. We express the value of the counter with the (non-duplicable) predicate $\mathsf{value}_\gamma(n)$, and we connect both pieces through the name γ.

Logically atomic triples. Fully explaining how one proves and uses logically atomic specifications is not necessary to understand the step-indexing troubles that arise. (See *Logical atomicity in Iris: The good, the bad, and the ugly*[23] for a detailed discussion.) It suffices to know a little bit more about the definition of a logically atomic triple:

$$\langle x.\, P(x)\rangle\; e\; \langle y.\, Q(x,y)\rangle_{\mathcal{E}} \triangleq \forall R.\, \left\{\mathbf{AU}(x.\, P(x), y.\, Q(x,y))^{\mathcal{E}}_{R}\right\}\; e\; \{y.\, R(y)\}$$

[23] Jung, *Logical atomicity in Iris: The good, the bad, and the ugly*, 2019 [Jun19].

Logically atomic triples are ordinary Hoare triples with a special *atomic update* $\mathbf{AU}$ in their precondition. The atomic update describes the "atomic action" of the operation. For example, in the case of `incr`, the atomic update would be $\mathbf{AU}_{\mathrm{inc}}(R) \triangleq \mathbf{AU}(n.\, \mathsf{value}_\gamma(n), m.\, m = n * \mathsf{value}_\gamma(n+1))^{\mathcal{N}}_{R}$, describing the abstract state change that we want `incr` to perform.

When we prove a logically atomic triple, it is our job to make sure the atomic update $\mathbf{AU}$ is executed, meaning we have to update the program state and ghost state atomically in the way described by the abstract action. We can update our state *atomically* by either (1) performing a physically atomic operation that corresponds to the update or (2) by calling another logically atomic operation. Executing the atomic update yields the "result" $R(y)$ in exchange. The fact that *we have to execute the update* is encoded somewhat implicitly: to prove the triple $\left\{\mathbf{AU}(x.\, P(x), y.\, Q(x,y))^{\mathcal{E}}_{R}\right\}\; e\; \{y.\, R(y)\}$, we eventually have to establish the postcondition $R(y)$. The postcondition R is *universally quantified* and the only way to obtain ownership of the result $R(y)$ is executing the atomic update.

Proving logical atomicity in the presence of helping. Let us return to helping. In the world of logical atomicity, helping means the helpee (*e.g.,* `incr`) transfers its atomic update to the helper (*e.g.,* the background thread). The helper then executes the atomic update at the linearization point of the helpee and afterwards returns the result (*e.g.,* $R(n)$) to the helpee. We refer to this mechanism as the *helping exchange.*

To understand the helping exchange better, we discuss helping the `incr` operation. We start with an *idealized* version of the exchange, because step-indexing sadly makes the matter more complicated. To initiate the exchange, `incr` sets up the following invariant:

$$I_{\mathrm{inc}}(n,R) \triangleq \boxed{(\mathbf{AU}_{\mathrm{inc}}(R) * \mathsf{pending}) \vee (R(n) * \mathsf{executed}) \vee \mathsf{received}}^{\,\mathcal{N}.\mathrm{inc}}$$

and shares it with the background thread through *another* invariant $\mathcal{N}$.main (not spelled out here)[24] that is used to define $\mathsf{is\_counter}^{\mathcal{N}}_{\gamma}(c)$. Here, we use the propositions pending, executed, and received to distinguish the different stages of the helping exchange.[25] Initially, in the pending stage, `incr` stores its update in I_{inc} and then waits for the background thread. The background thread eventually reads p and then updates b. In the step where it updates b, it linearizes the pending increment `incr`. To do so, it opens the invariant I_{inc}, takes out the atomic update $\mathbf{AU}_{\mathrm{inc}}$, executes it (similar to how one executes $\Rrightarrow P$), and puts the result $R(n)$ back into the invariant (advancing to the executed stage). Finally, `incr` observes the change of b, opens I_{inc}, and takes out the result $R(n)$ (advancing to the received stage).

[24] How exactly the invariant $\mathcal{N}$.main stores the `incr` invariants I_{inc} (and the `get` invariants I_{get}) is not important to explain the step-indexing issues that arise. The details can be found in the appendix of the later credits paper [Spi+22b].

[25] We keep pending, executed, and received abstract here to simplify the presentation. Internally, they use ghost state to encode the different stages of the helping mechanism.

13.2.3 Helping without Later Credits

Sadly, without later credits, the helping exchange for `incr` is more complicated, because step-indexing gets in the way. To execute AU_{inc} in the background thread, we first have to obtain ownership of the atomic update, which means taking it out of the invariant I_{inc}. Unfortunately, there are some hurdles: AU_{inc} is not timeless and is stored in an invariant (*i.e.,* I_{inc}), which itself is stored in *another* invariant (*i.e.,* the invariant $\mathcal{N}$.main behind is_counter$^{\mathcal{N}}_{\gamma}(c)$)—a step-indexing nightmare. So we cannot just take AU_{inc} out of the invariant I_{inc}, execute it, and put $R(n)$ back into the invariant, all in one step.

To escape this nightmare, the typical solicited helping proof is a play in three acts; we use `incr` to illustrate it. In the first act, the helper discovers the helpees it is helping (*e.g.,* through reading p). At this point, it gains access to a *witness*[26] $\triangleright W_{\text{inc}}$ for the waiting helpee—a resource relevant for the helping exchange, but guarded by a "$\triangleright$". In the second act, the helper takes a bookkeeping step of execution to eliminate the later (*e.g.,* the reduction of let). In the third act, the helper reaches the linearization point (*e.g.,* the update of b), and uses the witness W_{inc} to obtain access to AU_{inc} and execute it.

[26] It is not so important here what this witness is, only that we need to access it early. Later credits make this delicate play obsolete, so we keep it abstract and brief.

This strategy is suboptimal for several reasons. First, the helping exchange is more complicated than the intuition that we previously outlined and requires additional foresight. Second, making the dance with the witness W_{inc} work requires delicate step-indexing tricks behind the scenes. These tricks, known as "make-laterable", are so cumbersome that even its inventor called them "ugly".[27] (With later credits, "make-laterable" becomes obsolete, so we spare the reader the details here.) Third, "make-laterable" comes at a cost: clients of logically atomic specifications are faced with additional proof obligations when they want to use them. And last but not least, this strategy does not work for *unsolicited* helping: we now use `get` to illustrate why not.

[27] Jung, *Logical atomicity in Iris: The good, the bad, and the ugly*, 2019 [Jun19].

When the background thread reads the primary p, it cannot gain access to the witnesses W_{get} of all the `get` operations it linearizes. The reason is that they might not be there yet. That is, after the read of p, a new `get` operation could arrive. This `get` will be linearized with the update to b, but the background thread could not observe W_{get} yet when it read p. Thus, even with the established bag of tricks, we cannot realize the standard three-part play for `get`.

13.2.4 Helping with Later Credits

Enter later credits. With later credits, there is no need for a complicated three-act play, because we can instead just eliminate the requisite number of laters right at the linearization point. Thus, we can avoid relying on the ugly "make-laterable" trick in the definition of AU (which in turn means fewer proof obligations for clients of logically atomic triples), and we can implement the idealized helping exchange as originally envisioned.

To enable helping proofs, we set up the following scheme with later credits: if a helpee wants help from a helper, it pays the helper with *a later credit* $£1$, which is sent along with the atomic update AU in the shared invariant. The credit remains in the invariant while the update is pending, and can be removed when the atomic update has been executed. For example, in the troubling case of `get`, the invariant becomes:

$$I_{\mathrm{get}}(n, R) \triangleq \boxed{(\mathbf{AU}_{\mathrm{get}}(R) * £1 * \mathsf{pending}) \lor (R(n) * \mathsf{executed}) \lor \mathsf{received}}^{\mathcal{N}.\mathrm{get}}$$

where $\mathbf{AU}_{\mathrm{get}}(R) \triangleq \mathbf{AU}(n.\ \mathsf{value}_{\gamma}(n), m.\ m = n * \mathsf{value}_{\gamma}(n))^{\mathcal{N}}_{R}$ is the atomic update of `get`.

And that is it! In the presence of later credits, the idealized helping exchange just works. The later troubles vanish, since the helper always has a credit in hand when it needs to access an atomic update. For example, if the background thread needs to access the atomic update $\mathbf{AU}_{\mathrm{get}}$, then it can use the later credit stored along side with $\mathbf{AU}_{\mathrm{get}}$ to eliminate a guarding later from $\mathbf{AU}_{\mathrm{get}}$. Afterwards, it can execute the atomic update and return the result $R(n)$. Since the update is no longer pending, it does not have to put any credits back.

For the helpee, producing the later credit is straightforward. In a non-trivial logically atomic operation, there are plenty of bookkeeping steps around that have nothing to do with the linearization point (*e.g.,* the first step of beta reduction, let bindings, arithmetic, etc.), which all generate credits (see HOARE-PURE-STEP-CREDIT in Fig. 12.1). Since each of these steps generates a credit, but there is only one linearization point per operation, there are typically plenty of credits available.

To validate this point, we used later credits to reprove the major benchmarks for logical atomicity (*e.g.,* the elimination stack of Jung et al.[28]) with our simplified definition of **AU**, replacing the three-act play by the idealized helping exchange. Make-laterable, be gone!

[28] Jung et al., "Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning", 2015 [Jun+15].

Chapter 14

Soundness of Later Credits

We have now seen several applications of later credits, but we have yet to discuss how to justify their soundness. When we add later credits, the main challenge is that we have to ensure that the program logic remains *adequate*. That is, recall the main adequacy statement of Iris (from §4.4 in Part I):

Theorem 72 (Adequacy). *If* $\vdash \{\mathsf{True}\}\ e\ \{v.\ \phi(v)\}$, *then* e *is safe to execute in any heap* h *and all possible return values* v *satisfy the pure postcondition* $\phi(v)$.

The adequacy theorem turns proofs of Hoare triples $\vdash \{\mathsf{True}\}\ e\ \{v.\ \phi(v)\}$ into a correctness property of e, namely that e is safe to execute and only terminates in values satisfying ϕ. The adequacy theorem in the presence of later credits will be the main result of this chapter. To explain why it remains sound even with later credits (in §14.2), we first review (in more detail than in §4.4) how adequacy is proven traditionally in Iris (in §14.1). For our discussion, we stay at the same level of abstraction as in §4.1[1] and, to ease the presentation, we set concurrency aside. (Of course, in Rocq, we have proven adequacy also for concurrent programs.)

[1] That is, we set masks and fancy updates aside. In Rocq, we have of course proven the adequacy theorem for the more general version with fancy updates. To do so, we have replaced the regular update $\Rrightarrow$ in the definition of the fancy update ${}^{\mathcal{E}_1}{\Rrightarrow}^{\mathcal{E}_2}$ (see §4.5.1) with our new later elimination update $\Rrightarrow_{\mathsf{le}}$.

14.1 Adequacy in Iris

Let us first recall the relevant definitions. Recall that (in §3.3) Hoare triples are defined as $\{P\}\ e\ \{v.\ Q(v)\} \triangleq \Box(P \mathrel{-\!\!*} \mathbf{wp}\ e\ \{v.\ Q(v)\})$ in Iris, and that (in §4.1) the weakest precondition without concurrency is defined as:

$$\begin{aligned}
\mathbf{wp}\ v\ \{w.\ Q(w)\} &\triangleq {\Rrightarrow} Q(v) \\
\mathbf{wp}\ e\ \{w.\ Q(w)\} &\triangleq \forall h.\ \mathrm{SI}(h) \mathrel{-\!\!*} {\Rrightarrow}\mathrm{progress}(e,h) \qquad\qquad \text{if } e \notin \mathit{Val} \\
&\qquad * \ \forall e', h'.\ (e,h) \rightsquigarrow (e',h') \mathrel{-\!\!*} \triangleright{\Rrightarrow}(\mathrm{SI}(h') * \mathbf{wp}\ e'\ \{w.\ Q(w)\})
\end{aligned}$$

In the value case, we prove the postcondition (after an update). In the proper expression case, we prove that e can make progress in the current heap, and that for every successor expression e', we can re-establish the weakest precondition.

Proving adequacy. Let us now turn to the adequacy theorem. To simplify the proof sketch, we prove a slightly weaker form (which ignores the "safety" part of Theorem 72 and instead focuses on the postcondition ϕ). We start directly with the underlying weakest precondition:

Lemma 73. *If* $\vdash \mathbf{wp}\ e_0\ \{v.\ \phi(v)\}$ *and* $(e_0, h_0) \rightsquigarrow^n (e_n, h_n)$ *where* e_n *is a value, then* $\phi(e_n)$ *holds.*

Proof Sketch. We initialize the state interpretation SI. In the case of HeapLang, this means allocating the ghost state for the underlying heap in the state interpretation (see the definition of $\mathrm{SI}(h)$ in §4.1) using the heap ghost theory from §4.2.2. We obtain

$$\vdash \Rrightarrow \mathrm{SI}(h_0) * \mathbf{wp}\ e_0\ \{v.\ \phi(v)\}.$$

Consider the case where $n > 0$ and $(e_0, h_0) \leadsto (e_1, h_1) \leadsto^{n-1} (e_n, h_n)$. We unfold the weakest precondition and thus obtain:

$$\vdash \Rrightarrow \mathrm{SI}(h_0) * \begin{pmatrix} \forall h.\ \mathrm{SI}(h) \mathrel{-\!\!*} \Rrightarrow \mathrm{progress}(e_0, h) * \\ (\forall e', h'.\ (e_0, h) \leadsto (e', h') \mathrel{-\!\!*} \triangleright \Rrightarrow \mathrm{SI}(h') * \mathbf{wp}\ e'\ \{v.\ \phi(v)\}) \end{pmatrix}$$

Since we have the the state interpretation SI and a step of e_0, we can instantiate the assumptions, drop[2] $\mathrm{progress}(e_0, h_0)$, and obtain

$$\vdash \Rrightarrow \triangleright \Rrightarrow \mathrm{SI}(h_1) * \mathbf{wp}\ e_1\ \{v.\ \phi(v)\}.$$

(The two initial update modalities can be folded into one with upd-trans.) Doing the same again for the next step under the modalities "$\Rrightarrow \triangleright \Rrightarrow$" yields

$$\vdash (\Rrightarrow \triangleright \Rrightarrow)^2\ \mathrm{SI}(h_2) * \mathbf{wp}\ e_2\ \{v.\ \phi(v)\}.$$

We can then inductively repeat this process until we reach the value e_n. At that point, we obtain $\vdash (\Rrightarrow \triangleright \Rrightarrow)^n \Rrightarrow \phi(e_n)$ after dropping the state interpretation. Thus, the weakest precondition yields our desired postcondition $\phi(e_n)$, albeit under a number of updates and later modalities. (We arrive at the same conclusion for $n = 0$.) As the last step, to obtain $\phi(e_n)$ without the updates and laters, we then use Lemma 74 below. □

[2] The progress assertion here ensures that e_0 is not stuck. In the definition of $\mathbf{wp}\ e\ \{v.\ Q(v)\}$, it takes care of the "safety" part of Theorem 72.

Concretely, if one proves a pure proposition ϕ under an interleaving of updates and laters, then ϕ must hold, since it depends neither on step-indexing nor on resources:[3]

Lemma 74. *Let ϕ be a pure proposition. If $\vdash (\Rrightarrow \triangleright \Rrightarrow)^n \Rrightarrow \phi$, then ϕ holds.*

Proof Sketch. Recall the model of Iris's propositions from §4.3 and the entailment $P \vdash Q$. We pick the step-index $n+1$. Then we obtain the pure proposition ϕ at step-index 1. Thus ϕ must hold. □

[3] This lemma is a variant of Lemma 38. This version contains more update modalities (to match the interleaving we obtain in Lemma 73), but the two versions are interchangeable: since one can always introduce later modalities with later-intro and update modalities with upd-return, Lemma 38 can be used to derive Lemma 74 and vice versa.

With Lemma 74, we conclude our discussion of the original Iris adequacy proof.

14.2 Modeling Later Credits

With later credits, we introduce an additional layer in between program steps and laters. In the definition of the weakest precondition above, the later "$\triangleright$" enables later eliminations (see also §4.1). It tightly couples them to program steps, since after each program step, we get to eliminate another later. Later credits relax this connection. With later credits, a credit $£1$ becomes available with every program step. We can spend it immediately to eliminate a later, or we can save it for another proof step. This "amortized" form of reasoning about later eliminations works, because all that matters for adequacy is that in an n-step execution at most n laters are eliminated.[4] It matters less (as we will see below) *when* these laters are eliminated, so we delegate the responsibility to "track" later eliminations to a new modality, the later elimination update $\Rrightarrow_{\mathsf{le}} P$.

[4] Technically, the number of aggregate laters does not have to be exactly the same as the number of steps. One can, for example, allow eliminating $f(n)$ laters where f is a monotone function. This has been implemented in Iris by Matsushita et al. [Mat+22] (see §15).

The weakest precondition. Staying at the same level of abstraction as above, we redefine the weakest precondition as:

$$\begin{aligned}
\mathbf{wp}\ v\ \{w.\ Q(w)\} &\triangleq {\Rrightarrow_{\mathsf{le}}}\ Q(v) \\
\mathbf{wp}\ e\ \{w.\ Q(w)\} &\triangleq \forall h.\ \mathrm{SI}(h) \mathrel{-\!\!*} {\Rrightarrow_{\mathsf{le}}} \mathrm{progress}(e, h) \qquad\qquad \text{if } e \notin \mathit{Val} \\
&\quad * \forall e', h'. (e, h) \leadsto (e', h') \mathrel{-\!\!*} \pounds 1 \mathrel{-\!\!*} {\Rrightarrow_{\mathsf{le}}} (\mathrm{SI}(h') * \mathbf{wp}\ e'\ \{w.\ Q(w)\})
\end{aligned}$$

The changes are highlighted in blue: we use later elimination updates instead of standard updates, and we make a new credit available after every step.[5] In this updated definition, the connection between program steps and later eliminations is relaxed, because physical steps yield later credits which can subsequently be used for later eliminations virtually anywhere in the the rest of a proof.

[5] One change that is not highlighted is that this definition is no longer a guarded fixpoint. One option would be to define it as a least fixpoint (*e.g.,* as in §7.3.1). Another is to add an additional later to the definition and then still use a guarded fixpoint. Since the additional later modality does not hurt, in the Rocq implementation, we chose the latter for easier backwards compatibility.

Adequacy. Let us return to adequacy. For simplicity, we focus again on the special case (*i.e.,* on Lemma 73 but now for our updated weakest precondition):

Lemma 75. *If* $\vdash \mathbf{wp}\ e_0\ \{v.\ \phi(v)\}$ *and* $(e_0, h_0) \leadsto^n (e_n, h_n)$ *where* e_n *is a value, then* $\phi(e_n)$ *holds.*

Proof Sketch. The proof starts virtually unchanged, meaning we unfold the weakest precondition n times and we instantiate the execution of e_0. We obtain:

$$\vdash ({\Rrightarrow_{\mathsf{le}}}(\pounds 1 \mathrel{-\!\!*} {\Rrightarrow_{\mathsf{le}}}))^n {\Rrightarrow_{\mathsf{le}}} \phi(e_n)$$

Instead of updates and laters, here we iterate later elimination updates and credit assumptions "$\pounds 1 \mathrel{-\!\!*}$". We can pull all the credit assumptions out and obtain $\pounds n \vdash ({\Rrightarrow_{\mathsf{le}}}{\Rrightarrow_{\mathsf{le}}})^n {\Rrightarrow_{\mathsf{le}}}\phi(e_n)$. Using transitivity of "${\Rrightarrow_{\mathsf{le}}}$", this can be simplified to $\pounds n \vdash {\Rrightarrow_{\mathsf{le}}}\phi(e_n)$. From here, the desired goal $\phi(e_n)$ follows by chaining soundness of the later elimination update, Lemma 76 below, and Lemma 74. □

In the proof of Lemma 75, we can see that later credits delegate the responsibility of later management to the later elimination update. The proof boils down to $\pounds n \vdash {\Rrightarrow_{\mathsf{le}}}\phi(e_n)$, and from there we can retrieve the "amortized" interleaving of laters through the following soundness lemma:

Lemma 76 (Soundness of ${\Rrightarrow_{\mathsf{le}}}$)**.** If $\pounds n \vdash {\Rrightarrow_{\mathsf{le}}}\phi$, then $\vdash ({\Rrightarrow} \triangleright {\Rrightarrow})^n {\Rrightarrow} \phi$.

In other words, the later elimination update aggregates all of our step-index decreases and ghost state updates. We first define the later elimination update ${\Rrightarrow_{\mathsf{le}}}$ and then return to the proof of Lemma 76 below.

The later elimination update. The later elimination update can be defined using the existing connectives of Iris, so we never have to touch the underlying model (from §4.3). In fact, we have discussed almost all the pieces that are needed to define it. There is just one piece missing, *the credit supply* $\pounds_\bullet\, m$. The credit supply $\pounds_\bullet\, m$ is a piece of ghost state tracking the total number of available credits. That is, its value m is, at any time, the sum of all the credits $\pounds n$ distributed in the logic. The resources are an instance of the credit ghost theory from §4.2.2.[6] Thus, it satisfies the following rules:

[6] To be precise, the later credits $\pounds n \triangleq \boxed{\circ n}^{\gamma_{\mathsf{lc}}}$ are the fragments and the supply $\pounds_\bullet\, m \triangleq \boxed{\bullet m}^{\gamma_{\mathsf{lc}}}$ is the authoritative element of the resource algebra $\mathit{Auth}(\mathbb{N}, +)$ from §4.2.1. Both pieces are connected through the ghost name γ_{lc}, which is chosen globally.

SUPPLY-BOUND
$$\pounds_\bullet\, m * \pounds n \vdash m \geq n$$

SUPPLY-DECR
$$\pounds_\bullet\, (n + m) * \pounds n \vdash {\Rrightarrow} \pounds_\bullet\, m$$

The rule supply-bound ensures that $£_\bullet\, m$ is an upper bound. The rule supply-decr allows us to decrement the supply by giving up some credits.

We now turn to the later elimination update. Based on what we have seen, $\Rrightarrow_{\mathsf{le}}$ must be a monad, must connect later credits to later eliminations, and must enable ghost state updates. We achieve all of these properties with the following definition:

$$\Rrightarrow_{\mathsf{le}} P \triangleq \forall n.\, £_\bullet\, n \mathrel{-\!\!*} \Rrightarrow((£_\bullet\, n \ast P) \vee (\exists m < n.\, £_\bullet\, m \ast \triangleright \Rrightarrow_{\mathsf{le}} P))$$

Let us break it down. First, the definition quantifies over the current credit supply $£_\bullet\, n$. As a consequence, when we prove $\Rrightarrow_{\mathsf{le}} P$, we can make use of the rules supply-decr and supply-bound to (potentially) decrease the credit supply if we are willing to give up a later credit $£1$. Second, after an update to the ghost state, the later elimination update offers a choice: we can either (1) return the supply and prove P (turning it into a standard update), or (2) decrease the supply and correspondingly wrap the goal with a later, thereby enabling one later to be eliminated from any assumption in the context. (Note that the latter case is analogous to how $\triangleright$ was used to support later elimination in Iris's original definition of the weakest precondition from §4.1.) Finally, the definition is recursive, so we can repeat both ghost state updates and later eliminations (if we have additional credits). This recursion is handled with Iris's guarded fixpoints (see §4.3).

Let us now return to the soundness statement of the later elimination update:

Lemma 76. *If* $£n \vdash \Rrightarrow_{\mathsf{le}}\phi$, *then* $\vdash (\Rrightarrow \triangleright \Rrightarrow)^n \Rrightarrow \phi$.

Proof Sketch. We allocate $£_\bullet\, n$ and $£n$, meaning we obtain $\vdash \Rrightarrow(£_\bullet\, n \ast £n)$.[7] Using our assumption $£n \vdash \Rrightarrow_{\mathsf{le}}\phi$, we obtain $\vdash \Rrightarrow(£_\bullet\, n \ast \Rrightarrow_{\mathsf{le}}\phi)$. Analogous to the adequacy proof of the weakest precondition, we then unroll the later elimination update. That is, after unfolding "$\Rrightarrow_{\mathsf{le}}$", we have:

$$\vdash \Rrightarrow £_\bullet\, n \ast (\forall n.\, £_\bullet\, n \mathrel{-\!\!*} \Rrightarrow((£_\bullet\, n \ast \phi) \vee (\exists m < n.\, £_\bullet\, m \ast \triangleright \Rrightarrow_{\mathsf{le}}\phi)))$$

which can be simplified to $\vdash \Rrightarrow\Rrightarrow(£_\bullet\, n \ast \phi) \vee (\exists m < n.\, \triangleright(£_\bullet\, m \ast \Rrightarrow_{\mathsf{le}}\phi))$. In the left branch of the disjunction, we are done (using upd-return, upd-bind, and later-intro in Fig. 12.2). In the right branch, we are in a similar situation as before: we have a separating conjunction of the credit supply and a later elimination update (*i.e.,* $£_\bullet\, m \ast \Rrightarrow_{\mathsf{le}}\phi$). Thus, we can repeat the unfold-then-simplify step. Every time we consider the right branch, the credit supply decreases (*e.g.,* from n to some $m < n$) and, since n is finite, this decrease can happen at most n times. Consequently, after n unfold-then-simplify steps, we know the left branch must have been chosen. □

[7] At this point, the global ghost name γ_{lc} is chosen and, technically, the assumption $£n \vdash \Rrightarrow_{\mathsf{le}}\phi$ must hold for all choices of the global ghost name γ_{lc}. To avoid cluttering the presentation, we omit it here.

Backwards compatibility. In large parts of Iris, including the weakest precondition and the definition of fancy updates (*i.e.,* mask-changing updates; see §4.5.1), we replace the standard update modality "$\Rrightarrow$" with the later elimination update "$\Rrightarrow_{\mathsf{le}}$". Since $\Rrightarrow_{\mathsf{le}}$ and $\Rrightarrow$ satisfy nearly identical rules, the later credits mechanism is mostly backwards compatible. In fact, since the later credits paper was published, later credits have been integrated into the main Iris development. The only rules that the new update modality does not satisfy are interaction rules with Iris's "plainly modality" $\blacksquare\, P$. These rules were introduced by Timany et al.[8] for a logical relation for Haskell's ST monad, but are rarely used elsewhere in the Iris ecosystem.

[8] Timany et al., "A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of runST", 2018 [Tim+18].

Chapter 15

Extensions of Later Credits

In this chapter, we discuss prior techniques to generalize step-indexing and explain how later credits complement them. We give an overview of the techniques in the following table:

	Later eliminations per program step	
Traditional step-indexing	$1 \leadsto 1 \leadsto 1 \leadsto 1 \leadsto 1 \ldots$	
Folklore extension	$k \leadsto k \leadsto k \leadsto k \leadsto k \ldots$	for fixed k
Flexible step-indexing	$1 \leadsto 2 \leadsto 3 \leadsto 4 \leadsto 5 \ldots$	
Transfinite step-indexing	$n_1 \leadsto n_2 \leadsto n_3 \leadsto n_4 \leadsto n_5 \ldots$	for arbitrary n_i

Traditional step-indexing allows for the elimination of exactly one later per program step. It is folklore that this can be relaxed to k laters per step. Matsushita et al.[1] show that the number k does not have to be fixed upfront, but can depend on the program execution, so one can eliminate 1 later after the first step, 2 after the second, 3 after the third, *etc.* This extension of step-indexing uses *time receipts*[2] to keep track of the number of program steps. It is also possible to eliminate an *arbitrary* number of laters per step as shown by Svendsen, Sieczkowski, and Birkedal[3] using transfinite step-indexing. In particular, as briefly mentioned in §9.2, the safety program logic of Transfinite Iris (from Part II) allows one to eliminate an arbitrary number of laters per step.[4]

In all of these techniques, later elimination remains coupled to program steps—*i.e.,* a later can only be eliminated if the goal is a weakest precondition. Later credits are fundamentally different, because they turn the right to eliminate a later into an ownable resource $£\,1$ that can be saved and used even when the goal is merely an update modality. This decoupling is crucial for proofs where there is no program in sight when a later needs to be eliminated (*e.g.,* the examples from §13.1 and §13.2).

That said, later credits can be *combined* with these techniques to unlock additional, interesting applications. We combine flexible step-indexing and later credits (in §15.1) and use them for two examples: prepaid invariants and reverse refinements. We then discuss the extension of later credits to transfinite step-indexing and point out trade-offs compared to flexible step-indexing (§15.2).

[1] Matsushita et al., "RustHornBelt: A semantic foundation for functional verification of Rust programs with unsafe code", 2022 [Mat+22].

[2] Mével, Jourdan, and Pottier, "Time credits and time receipts in Iris", 2019 [MJP19].

[3] Svendsen, Sieczkowski, and Birkedal, "Transfinite step-indexing: Decoupling concrete and logical steps", 2016 [SSB16].

[4] The liveness logic of Transfinite Iris (see §7.3) allows one later elimination per target *and* source step. As in the safety logic, this could be generalized to an arbitrary number of laters. But also in the liveness logic, later eliminations remain coupled to program steps (see, *e.g.,* Fig. 7.3).

15.1 Flexible Step-Indexing

Similar to Matsushita et al.[5] and Mével, Jourdan, and Pottier,[6] we use time receipts to reflect the number of previous program steps into Iris. The rules of our extension are shown in Fig. 15.1. Each execution step produces a *time*

[5] Matsushita et al., "RustHornBelt: A semantic foundation for functional verification of Rust programs with unsafe code", 2022 [Mat+22].

[6] Mével, Jourdan, and Pottier, "Time credits and time receipts in Iris", 2019 [MJP19].

receipt-split
$$⧗(n+m) \dashv\vdash ⧗n * ⧗m$$

receipt-timeless
$$\mathsf{timeless}(⧗n)$$

hoare-pure-step-receipt
$$\frac{\{P * £1 * ⧗1\}\ e_2\ \{v.\, Q(v)\}_{\mathcal{E}} \qquad e_1 \rightarrow_{\mathrm{pure}} e_2}{\{P\}\ e_1\ \{v.\, Q(v)\}_{\mathcal{E}}}$$

hoare-receipt-credits-post
$$\frac{\{P\}\ e\ \{v.\, Q(v)\}_{\mathcal{E}} \qquad e \notin \mathit{Val}}{\{P * ⧗n\}\ e\ \{v.\, Q(v) * £n * ⧗n\}_{\mathcal{E}}}$$

Figure 15.1: The proof rules for later credits with flexible step-indexing.

receipt $⧗1$ and a credit $£1$ (hoare-pure-step-receipt). The receipts can be used to generate later credits (hoare-receipt-credits-post). That is, if we own n receipts $⧗n$, then we can leverage these receipts to generate an additional n credits $£n$ after the next step of execution.

We now briefly sketch two applications of this extension.

Prepaid invariants. Prepaid invariants $\boxed{R}^{\mathcal{N}}_{\mathsf{pre}} \triangleq \boxed{R * £1 * ⧗1}^{\mathcal{N}}$ store a later credit and a time receipt. Their rules are as follows:

inv-pre-alloc
$$\frac{\left\{P * \boxed{R}^{\mathcal{N}}_{\mathsf{pre}}\right\}\ e\ \left\{v.\, Q(v)\right\}_{\mathcal{E}}}{\{P * £1 * ⧗1 * \triangleright R\}\ e\ \{v.\, Q(v)\}_{\mathcal{E}}}$$

inv-pre-open
$$\frac{\{R * P\}\ e\ \{v.\, R * Q(v)\}_{\mathcal{E}\setminus\mathcal{N}} \qquad \mathcal{N} \subseteq \mathcal{E} \qquad e \text{ physically atomic}}{\left\{\boxed{R}^{\mathcal{N}}_{\mathsf{pre}} * P\right\}\ e\ \left\{v.\, Q(v)\right\}_{\mathcal{E}}}$$

Their distinguishing feature is that they can be opened around physically atomic instructions *without a guarding later* (inv-pre-open). The trick is that when we open the underlying invariant the later credit $£1$ can be used to eliminate the guarding later from R, and the time receipt $⧗1$ can be used to restore the later credit in the postcondition (with hoare-receipt-credits-post). A consequence of this trick is that we need to provide a credit and receipt when allocating the invariant (inv-pre-alloc). Since there is no “$\triangleright$” in inv-pre-open, we do not have to worry about later elimination when we work with, for example, nested invariants. In particular, with prepaid invariants, the nested invariant from §11 would not be an issue. However, we *cannot* open prepaid invariants around updates without a guarding later (see inv-open-upd in §13.1.1), because updates do not generate later credits.

Reverse refinements. Later credits with flexible step-indexing can also solve an issue with step-indexed logical relations described by Svendsen, Sieczkowski, and Birkedal.[7] The problem they highlight involves proving a contextual equivalence $F\,e \equiv_{\mathrm{ctx}} e : \tau$ for all expressions $e : \tau$ given a function $F : \tau \rightarrow \tau$. One strategy to show such an equivalence is to split proving the equivalence into the two contextual refinements $F\,e \leq_{\mathrm{ctx}} e : \tau$ and $e \leq_{\mathrm{ctx}} F\,e : \tau$. To prove these contextual refinements, one can prove the expressions logically refine each other, according to a step-indexed logical relation $\leq_{\log}$ (analogous to §13.1). In a logical refinement of the form $e_1 \leq_{\log} e_2 : \tau$, steps of e_1 allow elimination of laters. Thus, in the direction $F\,e \leq_{\log} e : \tau$, evaluating $F\,e$ takes steps that provide opportunities to eliminate laters. In the *reverse refinement* $e \leq_{\log} F\,e : \tau$,

[7] Svendsen, Sieczkowski, and Birkedal, “Transfinite step-indexing: Decoupling concrete and logical steps”, 2016 [SSB16].

that is not the case—we need to prove $e \leq_{\mathrm{log}} F\, e : \tau$ for all $e : \tau$, meaning that e could be a value, which takes no steps.

Svendsen, Sieczkowski, and Birkedal use a *transfinite* step-indexed logical relation to address this. But transfinite step-indexed models come with trade-offs (see §9.2 and §15.2), so we show that later credits provide an alternate solution. We do this by extending ReLoC and proving the reverse refinement example of Svendsen, Sieczkowski, and Birkedal, as well as a new and more difficult example involving concurrent memoization. The details can be found in the appendix accompanying the later credits paper.[8]

[8] Spies et al., *Later Credits Rocq development and technical documentation*, 2022 [Spi+22b].

15.2 Transfinite Step-Indexing

When combining Transfinite Iris (from Part II) with later credits, we obtain a rule that allows us to allocate an arbitrary number of credits in the postcondition of an expression:

$$\textsc{hoare-credits-post}\quad \frac{\{P\}\; e\; \{v.\, Q(v)\} \qquad e \notin \mathit{Val}}{\{P\}\; e\; \{v.\, Q(v) * \pounds n\}}$$

At first glance, this may seem like a strict improvement over flexible step-indexing since it lets us obtain n credits in each step without the need for time receipts (*cf.* hoare-receipt-credits-post). But there is a trade-off: As discussed in §9.2, the later commuting rules for existential quantification and separating conjunction (later-sep and later-exists in Fig. 12.2) are not sound in a transfinite model and are used in existing Iris proofs (*e.g.*, for logical atomicity). Conversely, the rule hoare-credits-post is unsound in a finitely step-indexed model. Thus, advanced users of later credits have a choice: work in the flexible finite model with time receipts, or work in the transfinite model without the commuting rules.

On the flip side, if one works in a transfinitely step-indexed setting, then later credits also provide a way to mitigate the missing commuting rules. For example, if we need the contents of an invariant *before* a step—instead of of using the commuting rules—one could use a later credit to remove the guarding later right away. As the rule hoare-credits-post shows, in principle, one has ample later credits available in Transfinite Iris. In fact, one can even define a transfinite version of later credits $\pounds\,\alpha$ in Transfinite Iris with ordinals instead of natural numbers and Hessenberg ordinal addition (discussed in §7.2) instead of regular natural number addition. With these credits one can obtain an ordinal number of credits per step (*e.g.*, ω credits to eliminate an arbitrary, finite number of laters subsequently). The definition of the later elimination update $\models\!\!\Rrightarrow_{\mathrm{le}}$ generalizes naturally to the transfinite case.

Chapter 16

Related Work

Multiple later eliminations per step. Svendsen, Sieczkowski, and Birkedal[1] and Matsushita et al.[2] have proposed techniques to generalize the traditional approach of "one later per step" to allow multiple step-index decreases per step. We have discussed these techniques in §15. As explained there, later credits are not an alternative to these techniques, they *complement* them. In particular, for the applications presented in this work—especially those in §13—it is vital that we can eliminate later modalities even when we are not proving a Hoare triple, which neither approach supports.

Steel. Steel[3] is a shallow embedding of concurrent separation logic in $F^\star$. Inspired by Iris, Steel supports dynamically allocated invariants but unlike Iris, opening an invariant in Steel does not introduce a later. Nevertheless, the underlying soundness argument crucially relies on program steps [Swa+20, Page 18], as in Iris. The difference arises because Steel treats ghost operations such as opening invariants as *explicit ghost code* that can take steps (which can then be erased before execution), allowing them to hide the later modality from the rule for opening invariants. The price for this more convenient interface is a loss in expressiveness—there is no Steel connective corresponding to Iris's update modality ("ghost actions without code", which logically atomic specifications are built on), and the authors of Steel say that "contextual refinement proofs are beyond what is possible in Steel" [Fro+21, Page 27].

Logical atomicity and linearizability. The main point of logical atomicity is to put user-defined, linearizable operations on (almost) the same footing as physically atomic instructions. In particular, users can open invariants around logically atomic operations. Prior work on logical atomicity either does not support helping[4] or relies on impredicative invariants in a step-indexed separation logic[5] with its suite of later elimination challenges. Later credits thus represent a significant step forward for logical atomicity proofs in general.

As Birkedal et al.[6] show, logical atomicity can also be used to prove the more traditional notion of linearizability.[7] To express and prove linearizability, many alternative approaches have been studied in prior work.[8] Those alternatives do not rely on impredicative invariants and, hence, they do not suffer from the step-indexing problems that later credits help solve. Instead, they have other means of expressing and establishing linearizability (*e.g.,* specifications that expose the effects of linearizable operations using a PCM of event histories[9]). What these approaches cannot do, compared to logical atomicity, is allow clients to treat linearizable operations "as if" they were physically atomic, meaning

[1] Svendsen, Sieczkowski, and Birkedal, "Transfinite step-indexing: Decoupling concrete and logical steps", 2016 [SSB16].

[2] Matsushita et al., "RustHornBelt: A semantic foundation for functional verification of Rust programs with unsafe code", 2022 [Mat+22].

[3] Swamy et al., "SteelCore: An extensible concurrent separation logic for effectful dependently typed programs", 2020 [Swa+20]; Fromherz et al., "Steel: Proof-oriented programming in a dependently typed concurrent separation logic", 2021 [Fro+21].

[4] Rocha Pinto, Dinsdale-Young, and Gardner, "TaDA: A logic for time and data abstraction", 2014 [RDG14].

[5] Svendsen and Birkedal, "Impredicative concurrent abstract predicates", 2014 [SB14]; Jung et al., "Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning", 2015 [Jun+15]; Jung et al., "The future is ours: Prophecy variables in separation logic", 2020 [Jun+20].

[6] Birkedal et al., "Theorems for free from separation logic specifications", 2021 [Bir+21].

[7] Herlihy and Wing, "Linearizability: A correctness condition for concurrent objects", 1990 [HW90].

[8] Elmas et al., "Simplifying linearizability proofs with reduction and abstraction", 2010 [Elm+10]; Liang and Feng, "Modular verification of linearizability with non-fixed linearization points", 2013 [LF13]; Turon, Dreyer, and Birkedal, "Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency", 2013 [TDB13]; Sergey, Nanevski, and Banerjee, "Specifying and verifying concurrent algorithms with histories and subjectivity", 2015 [SNB15]; Chakraborty et al., "Aspect-oriented linearizability proofs", 2015 [Cha+15]; Khyzha, Gotsman, and Parkinson, "A generic logic for proving linearizability", 2016 [KGP16]; Nanevski et al., "Specifying concurrent programs in separation logic: Morphisms and simulations", 2019 [Nan+19].

[9] Sergey, Nanevski, and Banerjee, "Specifying and verifying concurrent algorithms with histories and subjectivity", 2015 [SNB15]; Nanevski et al., "Specifying concurrent programs in separation logic: Morphisms and simulations", 2019 [Nan+19].

clients cannot open invariants around such user-defined operations, which is the main selling point of logical atomicity.

Reordering refinements. For stateful programming languages, Benton et al. investigated reordering refinements based on type-and-effect systems in a series of papers.[10] Their work was extended by Birkedal et al. [Bir+16][11] to a language with higher-order state and effect-masking. We focus on the work of Krogh-Jespersen, Svendsen, and Birkedal [KSB17][12] (which generalizes the work of Birkedal et al. [Bir+16]) and on the work of Timany et al. [Tim+18],[13] because they consider languages with cyclic features such as higher-order state and recursive types. These cyclic features are typically what motivate the use of step-indexing in a logical relation, but they also add an additional layer of complexity in reordering proofs.

Krogh-Jespersen, Svendsen, and Birkedal [KSB17] prove reorderings in a concurrent stateful language with an effect type system; their model supports reorderings of operations that write to *disjoint* parts of the heap and thus does not scale to the promise operations in §13.1.

Timany et al. [Tim+18] prove reorderings for a sequential language in the presence of Haskell's ST monad. They verify reorderings of pure computations (with stateful subcomputations encapsulated using ST), but for stateful computations they only show the expected monadic rules for the state monad. As mentioned earlier, our definition of reordering refinement is inspired by the work of Timany et al. The crucial difference is that we "bake in" support for later credits. As a consequence, we can prove the higher-order stateful reorderings of the promise operations (in §13.1), which are beyond the model of Timany et al. The key step in the proof, where we use an impredicative invariant to transfer the source execution between operations, is only possible because we can use later credits to eliminate the irksome laters that pop up.

[10] Benton et al., "Reading, writing and relations", 2006 [Ben+06]; Benton and Buchlovsky, "Semantics of an effect analysis for exceptions", 2007 [BB07]; Benton et al., "Relational semantics for effect-based program transformations with dynamic allocation", 2007 [Ben+07]; Benton et al., "Relational semantics for effect-based program transformations: Higher-order store", 2009 [Ben+09].

[11] Birkedal et al., "A Kripke logical relation for effect-based program transformations", 2016 [Bir+16].

[12] Krogh-Jespersen, Svendsen, and Birkedal, "A relational model of types-and-effects in higher-order concurrent separation logic", 2017 [KSB17].

[13] Timany et al., "A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of runST", 2018 [Tim+18].

Part IV

Quiver

Chapter 17

Introduction

Verifying *functional correctness* of large, stateful programs is one of the oldest challenges in computer science, tracing back to the work of Hoare[1] and Floyd.[2] Over the last two decades, a number of *deductive verification tools* based on separation logic have made remarkable progress in this direction, including VeriFast,[3] CFML,[4] Bedrock,[5] GRASShopper,[6] VST,[7] Viper,[8] Gillian,[9] Perennial,[10] RefinedC,[11] and CN[12]. They provide exceptionally strong verification guarantees (*e.g.,* memory safety and functional correctness in a pointer-manipulating language like C) by working with rich forms of separation logic, featuring, *e.g.,* custom predicates, user-defined functions, detailed programming language semantics, and large mathematical theories. In many cases, they even establish these guarantees *foundationally*—by producing machine checked proofs in proof assistants like Rocq.

Sadly, the overhead of functional correctness verification is considerable. For example, for manual verification in a proof assistant, the proofs are typically an order of magnitude larger than the code (see, *e.g.,* the work of Chajed et al. [Cha+21, Fig. 12] and Cao et al. [Cao+18, §11]). Hence, one of the longstanding goals has been to reduce this overhead—to scale verification to larger code bases, lower the entry barrier, and reduce the overall effort. Toward this goal, many of the aforementioned techniques have made great strides by developing *proof automation* (*e.g.,* [Jac+11; PWZ14; MSS17; Sam+21; San+20; Pul+23]), which often shrinks the overhead to—or even below—the code size (see, *e.g.,* the work of Pulte et al. [Pul+23, §5]). Instead of manual correctness proofs, these tools take as input the program code and one specification per function and then validate the program against the specification. To be precise, they offer the following verification paradigm:

code + specification (+ annotations & proofs) ⇒ OK/failure *(verification)*

where the user provides *code*, *specification*, and in some cases additional *annotations* (*e.g.,* to guide the proof search) and *proofs* (*e.g.,* lemmas in a proof assistant) and, then, the tool outputs *OK* (possibly with a proof) or *failure* (possibly with an error message), depending on whether the verification succeeded or not.

However, for verification of functional correctness to truly scale one day, proof automation alone might not be enough. The reason is that, in the current verification paradigm, even if the number of *annotations* decreases in the future and the *proof* component shrinks due to better proof automation, the resulting tools still require their users to *provide specifications manually.* And these specifications are rarely small. For a recent example, take RefinedC [Sam+21]. RefinedC targets proof automation for foundational verification of C code. Cor-

[1] Hoare, "An axiomatic basis for computer programming", 1969 [Hoa69].

[2] Floyd, "Assigning meanings to programs", 1967 [Flo67].

[3] Jacobs et al., "VeriFast: A powerful, sound, predictable, fast verifier for C and Java", 2011 [Jac+11].

[4] Charguéraud, "Characteristic formulae for the verification of imperative programs", 2011 [Cha11].

[5] Chlipala, "Mostly-automated verification of low-level programs in computational separation logic", 2011 [Chl11].

[6] Piskac, Wies, and Zufferey, "GRASShopper - complete heap verification with mixed specifications", 2014 [PWZ14].

[7] Appel, "Verified Software Toolchain", 2012 [App12]; Cao et al., "VST-Floyd: A separation logic tool to verify correctness of C programs", 2018 [Cao+18].

[8] Müller, Schwerhoff, and Summers, "Viper: A verification infrastructure for permission-based reasoning", 2017 [MSS17].

[9] Santos et al., "Gillian, Part I: A multi-language platform for symbolic execution", 2020 [San+20]; Maksimovic et al., "Gillian, Part II: Real-world verification for JavaScript and C", 2021 [Mak+21].

[10] Chajed et al., "Verifying concurrent, crash-safe systems with Perennial", 2019 [Cha+19a].

[11] Sammler et al., "RefinedC: Automating the foundational verification of C code with refined ownership types", 2021 [Sam+21].

[12] Pulte et al., "CN: Verifying systems C code with separation-logic refinement types", 2023 [Pul+23].

respondingly, regarding *proof overhead*, its "relative annotation overhead is moderate—less than 0.7 for all examples that do not involve complex side conditions" [Sam+21, §7]. But this statistic does not include the *specification overhead*, which is considerable in its own right: specifications contribute an additional 30-50 percent of the code size to the total verification overhead [Sam+21, Fig. 7]. Moreover, specifications often impose as much of a burden on the verification effort as do annotations, forcing the user to supply tedious side conditions about integer arithmetic, nontrivial preconditions about pointers, error cases, and conditionals over the possible return values.

Specification inference. So how can we meaningfully reduce the *specification overhead* of deductive separation logic techniques? The answer that we explore in this part of the dissertation is to fundamentally change the verification paradigm. Instead of treating the specification as an *input*, make it an *output* of the verification:

$$\text{code (+ specification sketches)} \Rightarrow \text{specification/failure} \qquad \textit{(inference)}$$

That is, given the code together with possibly additional hints in the form of specification sketches (explained below), our goal is to infer a specification based on the code. Working in separation logic, this specification can then be used (1) compositionally in the verification of other code, (2) to infer specifications of clients, or (3) by humans to compare the specification against their expectations.

Specification inference is an even harder problem than traditional program verification—it decreases the user-provided input and increases the desired output. Accordingly, it is not solved all at once. With **Quiver**, we embark on the crucial next steps of this journey. Quiver is the first technique for inferring—and foundationally verifying—functional correctness specifications in separation logic. It takes in sketches of function specifications and completes them to a full separation logic specification—adding missing preconditions, inferring postconditions, and filling out user-determined holes. To achieve this goal, Quiver proposes a new verification approach, *abductive deductive verification*, which integrates ideas from abductive inference with deductive separation logic verification to infer specifications in separation logic.

Automating separation logic. For deductive separation logic verification and automation, we follow in the footsteps of RefinedC.[13] RefinedC is a recently developed separation logic verification technique—built on top of Iris—for establishing functional correctness of C code. Its distinguishing feature is that it is foundational and, additionally, automated: Embedded into the Rocq proof assistant, RefinedC (1) provides powerful automation of separation logic, (2) inherits support for a large variety of functional correctness reasoning from Rocq's ambient meta-logic, and (3) is proven sound against Caesium, a detailed model of the C semantics in Rocq. For Quiver, we take inspiration from RefinedC's approach to separation logic proof automation (*i.e.,* goal-directed proof search for weakest preconditions; see §18), its separation logic-based type system for handling the complexities of C, and its embedding into Rocq to support a large variety of mathematical theories.

[13] Sammler et al., "RefinedC: Automating the foundational verification of C code with refined ownership types", 2021 [Sam+21].

As mentioned above, a weak spot of RefinedC is that it—like other deductive verification tools—requires considerable amounts of specification. To illustrate

```
 1 [[rc::parameters(n: Z)]]
 2 [[rc::args(n @ int<size_t>)]]
 3 [[rc::exists(l : loc)]]
 4 [[rc::returns(l @ &own<uninit<n>>)]]
 5 [[rc::ensures(block l n)]]
 6 void *xmalloc(size_t size) {
 7   void *ptr = malloc(size);
 8   if (ptr == NULL) abort();
 9   return ptr;
10 }
```

```
11 [[rc::parameters(n: Z)]]
12 [[rc::args(n @ int<size_t>)]]
13 [[rc::exists(l : loc)]]
14 [[rc::returns(l @ &own<zeros<n>>)]]
15 [[rc::ensures(block l n)]]
16 void *xzalloc(size_t size) {
17   void *ptr = xmalloc(size);
18   memset(ptr, 0, size);
19   return ptr;
20 }
```

Figure 17.1: Wrappers for memory allocation in C.

$$\{n \in \texttt{size\_t}\}\ \texttt{malloc}(n)\ \{v.\, v = \mathsf{NULL} \lor (\exists \ell, w.\, v = \ell * \ell \mapsto w * \mathsf{uninit}(w, n) * \mathsf{block}\,\ell\,n)\}$$

$$\{\ell \mapsto v * \mathsf{uninit}(v, n) * n \in \texttt{size\_t}\}\ \texttt{memset}(\ell, 0, n)\ \{\_.\, \exists w.\, \ell \mapsto w * \mathsf{zeros}(w, n)\} \qquad \{\mathsf{True}\}\ \texttt{abort}()\ \{\_.\, \mathsf{False}\}$$

$$\{n \in \texttt{size\_t}\}\ \texttt{xmalloc}(n)\ \{v.\, \exists \ell, w.\, v = \ell * \ell \mapsto w * \mathsf{uninit}(w, n) * \mathsf{block}\,\ell\,n\}$$

$$\{n \in \texttt{size\_t}\}\ \texttt{xzalloc}(n)\ \{v.\, \exists \ell, w.\, v = \ell * \ell \mapsto w * \mathsf{zeros}(w, n) * \mathsf{block}\,\ell\,n\}$$

Figure 17.2: Memory operations and wrappers in separation logic.

this point, let us consider a poster child example for specification inference. The functions `xmalloc` and `xzalloc`, depicted in Fig. 17.1, are simple helper functions for wrapping memory allocation in C (inspired by similar wrappers in popular open source projects[14]). They encapsulate common patterns such as (1) handling the case that allocation fails and `malloc` returns `NULL` (`xmalloc`) and (2) initializing freshly allocated memory with zeros (`xzalloc`). The implementations of the two functions are dead simple. Yet, when verifying them in RefinedC, we end up writing as many lines of specification (Lines 1-5 and Lines 11-15) as there are lines of code. And for no good reason: as we will see below, the specifications of `xmalloc` and `xzalloc` can be inferred from those of `malloc`, `memset`, and `abort`.

[14] Cyrus IMAPD, *Cyrus IMAPD memory wrapper operations*, 2025 [Cyr25]; Git, *Git memory wrapper operations*, 2025 [Git25]; Redis, *Redis memory wrapper operations*, 2025 [Red25a].

Abductive inference. A key building block for us in reducing the specification burden is the idea of *abductive inference* in the sense that we infer the specification for a piece of code by “puzzling together” existing specifications for its component parts. To illustrate this idea, let us assemble the specifications of `xmalloc` and `xzalloc` from the auxiliary operations `malloc`, `memset`, and `abort`. The specifications of all operations are depicted in Fig. 17.2. (For simplicity, we phrase these specifications in a separation logic instead of RefinedC’s type system.) The operation `malloc` takes a `size_t` integer n and returns either `NULL` or a pointer ℓ to a memory block of size n (denoted $\mathsf{block}\,\ell\,n$) whose contents w are uninitialized (denoted $\mathsf{uninit}(w, n)$). The operation `memset`, called with zero, initializes the contents of a pointer with zeros (denoted $\mathsf{zeros}(w, n)$), and the operation `abort` never returns (postcondition False). Using these specifications, we can assemble the specification of `xmalloc` (and analogously `xzalloc`) as follows: The precondition $n \in \texttt{size\_t}$ is inherited from `malloc`. The postcondition is derived from the post of `malloc` knowing that, in the `NULL`-case, we never return due to `abort`.

The idea of using abductive inference in separation logic is not new. It was first pioneered by *bi-abduction*,[15] a landmark technique for compositional shape analysis based on separation logic. Bi-abduction is one of the cornerstones of Meta's Infer tool for detecting bugs in millions of lines of code[16] and also inspired a line of research on bug finding using incorrectness logic.[17] It takes as input the code of a function and generates a separation logic specification that summarizes the footprint of the code via abductive inference. However, in the interest of supporting "push-button" automation, bi-abduction focuses on fixed, restricted fragments of separation logic. For example, the original work of Calcagno et al.[18] restricts attention to points-to assertions $\ell \mapsto v$, list segments $\mathsf{lseg}(\ell, r)$, and equalities $v = v'$. As such, it cannot express—or abductively infer—for example, the specifications of `xmalloc` and `xzalloc` in Fig. 17.2, since they go beyond this fragment.[19]

[15] Calcagno et al., "Compositional shape analysis by means of bi-abduction", 2009 [Cal+09]; Calcagno et al., "Compositional shape analysis by means of bi-abduction", 2011 [Cal+11].

[16] Calcagno et al., *Go Huge or Go Home: POPL'19 Most Influential Paper Retrospective*, 2019 [Cal+19].

[17] O'Hearn, "Incorrectness Logic", 2020 [OHe20]; Raad et al., "Local reasoning about the presence of bugs: Incorrectness Separation Logic", 2020 [Raa+20]; Le et al., "Finding real bugs in big programs with incorrectness logic", 2022 [Le+22].

[18] Calcagno et al., "Compositional shape analysis by means of bi-abduction", 2009 [Cal+09]; Calcagno et al., "Compositional shape analysis by means of bi-abduction", 2011 [Cal+11].

[19] Modifying this fragment is a challenging feat. Considerable follow-on work has gone into adding *individual extensions* (*e.g.*, linear integer arithmetic [Tri+13] or low-level pointer representation [Hol+22]). See §23 for an overview.

Abductive deductive verification. With Quiver, we pursue a fundamentally different approach. Rather than trying to build push-button automation by restricting the separation logic fragment, we instead aim to integrate abductive inference into deductive verification approaches that already handle rich fragments of separation logic. To do so, we introduce a new technique we call *abductive deductive verification*. The main abductive deductive verification judgment $\Delta * [R] \vdash \mathbf{wp}\ e\ \{\Phi\}$ (where we abbreviate $\mathbf{wp}\ e\ \{\Phi\} \triangleq \mathbf{wp}\ e\ \{v.\ \Phi(v)\}$) marries deductive separation logic verification, via the *weakest precondition* connective $\mathbf{wp}\ e\ \{\Phi\}$ (see §3.1), with abductive inference of a precondition R, via the *abduction judgment* $\Delta * [R] \vdash G$. Concretely, deriving $\Delta * [R] \vdash \mathbf{wp}\ e\ \{\Phi\}$ corresponds to deductively verifying the expression e in the context Δ while, simultaneously, abductively inferring any missing resources R that are needed to do so. By combining both styles of reasoning, we maintain the ability to deductively verify programs with rich separation logic specifications while additionally benefiting from the advantages of specification inference (*e.g.*, inferring the specification of `xmalloc` while verifying it; see §22.2).

Specification sketches. Since Quiver targets rich separation logics, *fully automatically* puzzling together specifications is not always the right choice (or even feasible). Consider the following extension of the previous example—a function that allocates a vector initialized with zeros:

```
vec_t mkvec(int n) {
  size_t s = sizeof(int) * (size_t)n;
  vec_t vec = xmalloc(sizeof(struct vector));
  vec->data = xzalloc(s);
  vec->len = n;
  [[q::type(? @vec_t)]] return vec;
}
```

(For now, we ask the reader to ignore the annotation "`[[q::...]]`".) A standard functional correctness specification for `mkvec` would be

$$\{n \in \mathtt{int} * n \geq 0\}\ \mathtt{mkvec}(n)\ \{v.\ \mathsf{vec}(v, 0^n)\}$$

where $\mathsf{vec}(v, xs)$ is an abstract predicate for vectors with contents xs (a list of integers) and 0^n is a list filled with n zeros. If we simply "puzzle together" a specification for `mkvec` based on the specifications of `xmalloc` and `xzalloc` (in Fig. 17.2), however, we would arrive at a low-level specification in terms of points-to assertions and the zeros-predicate—not a high-level specification about vectors. It would look like something along the lines of:

$$\{n \in \texttt{int} * n \geq 0\}$$
$$\texttt{mkvec}(n)$$
$$\left\{v.\begin{array}{l} \exists \ell, r.\, v = \ell * \ell \mapsto \{\texttt{data} = r, \texttt{len} = n\} * \mathsf{block}\, \ell\, sz_{\texttt{vec}} \\ * \exists w.\, r \mapsto w * \mathsf{zeros}(w, sz_{\texttt{int}} \cdot n) * \mathsf{block}\, r\, (sz_{\texttt{int}} \cdot n) \end{array}\right\}$$

where $sz_{\texttt{int}} \triangleq \textbf{sizeof}(\texttt{int})$ and $sz_{\texttt{vec}} \triangleq \textbf{sizeof}(\textbf{struct}\ \texttt{vector})$. The underlying issue is that a single function can have multiple specifications at different levels of abstraction. By only puzzling together known specifications, we may end up with one at a lower level of abstraction. To find "the right one", we have to take the *intent* of the developer into account. This is why, in Quiver, we guide the inference using *specification sketches*.

Guided specification inference à la Quiver. That is, Quiver explores the middle ground in between (a) taking a complete specification as user input and verifying the code (as in RefinedC) and (b) taking only the code as input and inferring the entire specification (as in bi-abduction). We take a *specification sketch* as input, use it to resolve ambiguity, and complete it to a full specification—but without requiring the user to provide every little detail.

Quiver works in three steps:

1. *Data type declarations.* First, the user defines their custom data types that are used in the code (*e.g.,* arrays, linked lists, maps, buffers, vectors, *etc.*). This step includes choosing mathematical domains, imposing invariants on values, and relating mathematical and physical representations.

2. *Function specification sketches.* Second, the user can provide sketches for functions. These sketches are similar to separation logic specifications (*e.g.,* describing the abstract predicates for arguments). There is, however, a crucial difference: they are *incomplete* with holes for, *e.g.,* arguments of abstract predicates, additional constraints, and missing ownership.

3. *Specification inference.* Finally, Quiver takes this sketch and completes it into a specification for the entire function using abductive deductive verification. This includes adding missing preconditions, making imprecise annotations precise, adding constraints for unspecified function arguments, and figuring out the postcondition of the function.

In the resulting system, users control how much specification they want to provide. By default, if the inference is successful, the resulting specification closely follows the code. If the user decides to "sprinkle in" some annotations that constrain function arguments or local variables to a certain data type, Quiver takes these into account and adjusts the specification accordingly. And if the user provides the complete function specification, Quiver turns into a traditional technique for verifying functional correctness. For example, the specifications of `xmalloc` and `xzalloc` can be derived fully automatically without any sketches. For `mkvec`, we only add the sketch in Line 26: it instructs Quiver that the return value is a vector, which results in the high-level vec-specification.

Contributions. Our key theoretical contribution in this part of the dissertation is the approach of **Abductive Deductive Verification** (§18), which provides a powerful basis for specification inference in rich separation logics. With the abductive deductive verification judgment $\Delta * [R] \vdash \textbf{wp}\ e\ \{\Phi\}$, we marry traditional deductive verification via the *weakest precondition* $\textbf{wp}\ e\ \{\Phi\}$ with abductive inference via the *abduction judgment* $\Delta * [R] \vdash G$.

Our key technical contribution is that we realize abductive deductive verification in the form of **Quiver**, which consists of four parts:

- The *abduction engine* **Argon** (§19), which automates the abduction judgment $\Delta * [R] \vdash G$. Its key contribution is a *goal-directed proof search procedure* for abductive inference. It supports predicate-transformer style reasoning, necessitated by the weakest precondition $\textbf{wp}\ e\ \{\Phi\}$, provides extensible proof search, and has powerful support for instantiating existential quantifiers.
- The *type system* **Thorium** (§20), which uses types in separation logic (*à la* RefinedC) to scale automated reasoning about the weakest precondition $\textbf{wp}\ e\ \{\Phi\}$ to the complexities of C. Its key contribution is that it works under *incomplete information* about the proof context Δ, meaning it works even when the types that are supposed to guide the proof search are yet to be determined.
- A proof-of-concept **Implementation** (§21) in the Rocq proof assistant—with a frontend for C, building on Iris and RefinedC's Caesium semantics for C. The implementation infers specifications and, at the same time, proves them correct in Rocq, which includes proving the absence of out-of-bounds accesses, use-after-free, and integer overflows.
- An **Evaluation** (§22), applying Quiver to several interesting case studies, including a dynamically-allocated vector data type and code from popular open-source libraries.

The implementation and all inferred specifications are provided in the Quiver Rocq development. The Quiver paper is additionally accompanied by an appendix containing further details on the inferred specifications.[20]

[20] Spies et al., *Quiver: Guided abductive inference of separation logic specifications in Coq (Rocq development and appendix)*, 2024 [Spi+24b].

Limitations. Quiver does not infer loop invariants, but supports manually provided invariants. Similarly, Quiver does not infer or complete specifications of recursive functions. Quiver does not infer specifications of function pointers and only handles sequential code. Moreover, while Quiver builds on the detailed Caesium C semantics, Quiver does not handle all features of C. In particular, it does not enforce that pointer accesses are aligned, it does not support unions, it does not support integer-pointer casts, and it inherits the limitations of Caesium (*e.g.*, no floating point numbers).

Chapter 18

Abductive Deductive Verification

In this chapter, we explain our approach of *abductive deductive verification*. Concretely, we discuss the abductive deductive verification judgment $\Delta * [R] \vdash \mathbf{wp}\ e\ \{\Phi\}$ (in §18.1), the treatment of existential quantification (in §18.2), and how we steer the inference via specification sketches (in §18.3). To avoid getting bogged down in the details of C, we focus on a simple, expository language λ_{expo} for this explanation. From §19 onwards, we will then explain how we scale the approach to actual C code.

$$\begin{array}{lcll} \text{Expressions} & e & ::= & x \mid v \mid e_1 \odot e_2 \mid f(\vec{e}) \mid e{\rightarrow}m \mid e_1{\rightarrow}m := e_2 \mid \mathsf{new}() \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \cdots \\ \text{Assertions} & P, Q & ::= & \ell \mapsto t \mid \mathrm{loc}(v, \ell) \mid \mathrm{int}(v, n) \mid \mathrm{P}(v, \vec{x}) \mid n_1 \leq n_2 \mid n_1 = n_2 \mid \cdots \end{array}$$

Figure 18.1: The exposition language λ_{expo}.

A running example. The language λ_{expo} is depicted in Fig. 18.1. Inspired by HeapLang, it is a simple, substitution-based language with heap-allocated data structures.[1] In λ_{expo}, heap-allocated data structures are modeled as mutable, finite maps t from fields to values (similar to objects in JavaScript). We write $\mathrm{dom}\, t$ for the fields of t and ϵ for the empty map. Values v can be unit $()$, locations ℓ, and integers n. The expression $\mathsf{new}()$ allocates an empty struct, $e{\rightarrow}m := e'$ assigns e' to the field m of e, and $e{\rightarrow}m$ dereferences field m of e. We abbreviate $e_1; e_2 \triangleq \mathsf{let}\ \_ = e_1\ \mathsf{in}\ e_2$.

[1] The language λ_{expo} is only meant to serve as an exposition to abductive deductive verification, so we have not mechanized it. We return to the range example discussed here in §20.3, where it is implemented in C.

To reason about λ_{expo} in separation logic, we use *resources* and *pure assertions*. The resources are *points-to assertions* $\ell \mapsto t$, which assert *ownership* of a struct at location ℓ with at least the fields t, and *abstract predicates* $\mathrm{P}(v, \vec{x})$ (see §18.3). The pure assertions include $\mathrm{loc}(v, \ell)$ for "v is the location ℓ" and $\mathrm{int}(v, n)$ for "v is the integer n".

As a running example, we consider *a data type for integer ranges* $[n_{\text{s}}, n_{\text{e}})$, implemented in λ_{expo}, in the remainder of this chapter. This range data type is represented by a struct with two integer fields: s (for the start of the range) and e (for one past the end of the range). We define three operations operating on ranges, depicted in Fig. 18.2: `init` for initializing a previously allocated range r with bounds a and b, `mkrange` for allocating and initializing a new range from a to b, and `size` to determine the size of a range r.

```
init(r, a, b)    := r→s := a; r→e := b; assert(range(r, ?, ?))
mkrange(a, b)    := let r = new() in init(r, a, b); r
size(r)          := assert(range(r, ?, ?)); r→e − r→s; assert(range(r, ?, ?))
```

Figure 18.2: The implementation of the range data type. Quiver assertion annotations in blue.

Weakest Precondition Rules (Language-Specific)

$$\begin{array}{ll}
\mathbf{wp}\; v\; \{\Phi\} \;\dashv\; \Phi\, v & (\textsc{wp-val}) \\
\mathbf{wp}\; (\mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2)\; \{\Phi\} \;\dashv\; \mathbf{wp}\; e_1 \left\{v.\, \mathbf{wp}\; (e_2[v/x])\; \{\Phi\}\right\} & (\textsc{wp-let}) \\
\mathbf{wp}\; (v{\to}m := w)\; \{\Phi\} \;\dashv\; \ell \mapsto t * (\ell \mapsto t[m := w] \mathrel{-\!\!*} \Phi\,())\; \textit{when}\; \mathrm{loc}(v, \ell) * \ell \mapsto t & (\textsc{wp-assign}) \\
\mathbf{wp}\; (v{\to}m := w)\; \{\Phi\} \;\dashv\; \exists \ell, t.\, \mathrm{loc}(v, \ell) * \ell \mapsto t * (\ell \mapsto t \mathrel{-\!\!*} \mathbf{wp}\; (v{\to}m := w)\; \{\Phi\}) & (\textsc{wp-assign-def}) \\
\mathbf{wp}\; (v{\to}m)\; \{\Phi\} \;\dashv\; \ell \mapsto t * (\ell \mapsto t \mathrel{-\!\!*} \Phi(t(m)))\; \textit{when}\; \mathrm{loc}(v, \ell) * \ell \mapsto t * m \in \mathrm{dom}\, t & (\textsc{wp-read}) \\
\mathbf{wp}\; \mathsf{new}()\; \{\Phi\} \;\dashv\; \forall \ell, v.\, \mathrm{loc}(v, \ell) * \ell \mapsto \epsilon \mathrel{-\!\!*} \Phi\, v & (\textsc{wp-new}) \\
\mathbf{wp}\; f(\vec{v})\; \{\Phi\} \;\dashv\; \mathbf{apply}(T\, \vec{v})\{\Phi\}\; \textit{when}\; \mathrm{spec}(f, T) & (\textsc{wp-call}) \\
\mathbf{wp}\; (\mathsf{assert}(\exists x.\, P\, x))\; \{\Phi\} \;\dashv\; \mathbf{assert}(x.\, P\, x)\{\Phi\,()\} & (\textsc{wp-assert})
\end{array}$$

Abduction Rules (Generic)

$$\textsc{abd-embed}\quad \dfrac{E \dashv G \;\textit{when}\; P \qquad \Delta \vdash P \qquad \Delta * [R] \vdash G}{\Delta * [R] \vdash E} \qquad \textsc{abd-res-ctx}\quad \dfrac{\Delta = \Delta', M \qquad \Delta' * [R] \vdash G}{\Delta * [R] \vdash M * G} \qquad \textsc{abd-res-missing}\quad \dfrac{\Delta * [R] \vdash G}{\Delta * [M * R] \vdash M * G}$$

$$\textsc{abd-pure-prove}\quad \dfrac{\Delta \vdash \phi \qquad \Delta * [R] \vdash G}{\Delta * [R] \vdash \phi * G} \qquad \textsc{abd-pure-missing}\quad \dfrac{\Delta, \phi * [R] \vdash G}{\Delta * [\phi * R] \vdash \phi * G} \qquad \textsc{abd-wand-res}\quad \dfrac{\Delta, M * [R] \vdash G}{\Delta * [R] \vdash M \mathrel{-\!\!*} G} \qquad \textsc{abd-wand-pure}\quad \dfrac{\Delta, \phi * [R] \vdash G}{\Delta * [R] \vdash \phi \mathrel{-\!\!*} G}$$

$$\textsc{abd-exists}\quad \dfrac{\forall x.\, (\Delta * [R x] \vdash G x)}{\Delta * [\exists x.\, R x] \vdash \exists x.\, G x} \qquad \textsc{abd-all}\quad \dfrac{\forall x.\, (\Delta * [R] \vdash G x)}{\Delta * [R] \vdash \forall x.\, G x} \qquad \textsc{abd-end}\quad \Delta * [\forall \vec{x}.\, \Delta \mathrel{-\!\!*} \Phi v] \vdash \Phi v \qquad \textsc{abd-true}\quad \Delta * [\mathsf{True}] \vdash \mathsf{True}$$

Figure 18.3: Weakest precondition rules for λ_{expo} and generic abduction rules. Overlapping weakest precondition rules are applied top-to-bottom, and overlapping abduction rules are applied left-to-right.

18.1 The Essence of Abductive Deductive Verification

Before we dive into the details of the range example, we start by explaining the abductive deductive verification judgment $\Delta * [R] \vdash \mathbf{wp}\; e\; \{\Phi\}$. It consists of two parts: the *weakest precondition* $\mathbf{wp}\; e\; \{\Phi\}$ (§3.1) and an *abduction judgment* $\Delta * [R] \vdash G$ where Δ is a separation logic context, R is an additional *inferred* precondition, and G is the current goal. The basic idea is that when we derive $\Delta * [R] \vdash \mathbf{wp}\; e\; \{\Phi\}$, we prove that Δ together with R is a sufficient precondition for e to satisfy the postcondition Φ. That is:

$$\Delta * [R] \vdash \mathbf{wp}\; e\; \{v.\, \Phi\, v\} \quad \textit{implies} \quad \{\Delta * R\}\; e\; \{v.\, \Phi\, v\}$$

We explain how $\Delta * [R] \vdash \mathbf{wp}\; e\; \{\Phi\}$ works with the rules in Fig. 18.3. To stage the explanation, we present it in three steps, moving from verification to inference. In Version 1, we use the judgment for ordinary verification—without any inference—and explain the proof search strategy underlying our automation: goal-directed proof search for weakest preconditions. In Version 2, we extend the judgment to infer preconditions, by incorporating abduction into our goal-directed proof search. In Version 3, we extend it further to infer complete specifications. In this last version, we explain why we infer so-called "predicate transformer specifications" instead of Hoare triples.

To keep the explanation concrete, we focus on the operation `init` of the range data type. For now, we ask the reader to ignore the blue assertion in the code of `init` (in Fig. 18.2). We will infer the following specification for `init`:

$$\{\mathrm{loc}(v_r,\ell) * \ell \mapsto t\}\ \texttt{init}(v_r,v_a,v_b)\ \{\_.\ \mathrm{loc}(v_r,\ell) * \ell \mapsto t[\texttt{s} := v_a,\ \texttt{e} := v_b]\}$$

The precondition assumes that v_r is a location ℓ and that this location ℓ stores a struct with contents t. The postcondition ensures that ℓ stores an updated struct with v_a in its `s`-field and v_b in its `e`-field.

Version 1: Deductive verification. We begin by supposing we are *given* the above specification $\{P_{\texttt{init}}\}\ \texttt{init}(v_r,v_a,v_b)\ \{\_.\ Q_{\texttt{init}}\}$, where we abbreviate $P_{\texttt{init}} \triangleq \mathrm{loc}(v_r,\ell) * \ell \mapsto t$ and $Q_{\texttt{init}} \triangleq \mathrm{loc}(v_r,\ell) * \ell \mapsto t[\texttt{s} := v_a,\ \texttt{e} := v_b]$, and we want to deductively verify it. We aim to deductively verify it via the judgment $\Delta * [R] \vdash \mathbf{wp}\ e\ \{\Phi\}$, where we instantiate $\Delta \triangleq P_{\texttt{init}}$, $\Phi(\_) \triangleq Q_{\texttt{init}}$, and e with the body of $\texttt{init}(v_r,v_a,v_b)$, and for now we ignore the precondition R. (The reader can pretend it is True.) That is, we set out to *prove*:

$$\mathrm{loc}(v_r,\ell),\ \ell \mapsto t * [\_] \vdash \mathbf{wp}\ (v_r \texttt{->s} := v_a;\ v_r \texttt{->e} := v_b)\ \{\_.\ Q_{\texttt{init}}\}$$

Our proof strategy for deductive verification—following in the footsteps of RefinedC[2]—is to employ *goal-directed proof search for weakest preconditions.* It is goal-directed in the sense that, to derive $\Delta * [\_] \vdash G$, we iteratively inspect the current goal G and then apply a matching rule that transforms G into a new goal G'. (If multiple rules match, we use the first one whose side conditions can be proven; the order is described in the figures.) And to reason about weakest preconditions, we use the rule ABD-EMBED to embed deductive proof rules of the form "$E \dashv G$ *when* P": here, E is the weakest-pre goal we are trying to solve, and G is the new subgoal that implies it under the (optional) side condition P.

[2] Sammler et al., "RefinedC: Automating the foundational verification of C code with refined ownership types", 2021 [Sam+21].

In the case of $\mathbf{wp}\ (v_r \texttt{->s} := v_a;\ v_r \texttt{->e} := v_b)\ \{\_.\ Q_{\texttt{init}}\}$, we start by using ABD-EMBED to apply WP-LET (similar to WP-BIND in §3.1). It breaks up sequential composition $e_1; e_2 \triangleq \mathrm{let}\ \_ = e_1\ \mathrm{in}\ e_2$ by putting the $\mathbf{wp}$ of e_2 into the post of e_1, turning the goal into

$$\mathrm{loc}(v_r,\ell),\ \ell \mapsto t * [\_] \vdash \mathbf{wp}\ (v_r \texttt{->s} := v_a)\ \left\{\_.\ \mathbf{wp}\ (v_r \texttt{->e} := v_b)\ \{\_.\ Q_{\texttt{init}}\}\right\}$$

Next, we apply WP-ASSIGN. It imposes an additional side condition on the context (via ABD-EMBED), namely v_r should be some location ℓ for which we have a points-to assertion $\ell \mapsto t$. Thus, leaving as our new goal:[3]

$$\mathrm{loc}(v_r,\ell),\ \ell \mapsto t * [\_] \vdash \ell \mapsto t * (\ell \mapsto t[\texttt{s} := v_a] \mathrel{-\!\!*} \mathbf{wp}\ (v_r \texttt{->e} := v_b)\ \{\_.\ Q_{\texttt{init}}\})$$

[3] Note that the side condition in ABD-EMBED is only a check. To be sound, the weakest precondition rule still needs to consume (and then return) ownership of the location ℓ.

The rule WP-ASSIGN (similar to WP-STORE in §3.1) has transformed the goal such that we should first give up the ownership of ℓ (with "$\ell \mapsto t *$") and then we get back the updated ownership again (with "$\ell \mapsto t[\texttt{s} := v_a] \mathrel{-\!\!*}$"). We do the former with ABD-RES-CTX and the latter with ABD-WAND-RES, leaving us to prove

$$\mathrm{loc}(v_r,\ell),\ \ell \mapsto t[\texttt{s} := v_a] * [\_] \vdash \mathbf{wp}\ (v_r \texttt{->e} := v_b)\ \{\_.\ Q_{\texttt{init}}\}$$

We proceed in a similar fashion for the second assignment, updating the points-to assertion to $\ell \mapsto t[\texttt{s} := v_a,\ \texttt{e} := v_b]$, which satisfies the desired postcondition.

Version 2: Abducting the precondition. We now turn to *inference*. As before, we deductively verify $\Delta * [R] \vdash \mathbf{wp}\ e\ \{\Phi\}$ in goal-directed fashion—except that now we allow for the possibility that the context Δ was not sufficient to prove the goal, and we infer the missing precondition R. This means that, whereas Δ, e, and Φ are inputs, R is an *output* of $\Delta * [R] \vdash \mathbf{wp}\ e\ \{\Phi\}$.

To stage the presentation, we will make the simplifying assumption in this version that the postcondition Φ is simply True. Thus, to infer the precondition $P_{\texttt{init}}$, we solve

$$\emptyset * [R] \vdash \mathbf{wp}\ (v_r{\to}\mathsf{s} := v_a;\ v_r{\to}\mathsf{e} := v_b)\ \{\_.\ \mathsf{True}\}.$$

Note that now the context is *empty*. The proof search proceeds as in Version 1 until we reach the first assignment:

$$\emptyset * [R] \vdash \mathbf{wp}\ (v_r{\to}\mathsf{s} := v_a)\ \big\{\_.\ \mathbf{wp}\ (v_r{\to}\mathsf{e} := v_b)\ \{\_.\ \mathsf{True}\}\big\}$$

At this point, the rule WP-ASSIGN does not apply anymore, since the ownership of v_r is not in the context. (It is empty!) Instead, the missing ownership should come from the precondition that we want to infer, so we must *add it* to the precondition R. To do so, we proceed in several steps. First, we use a second rule for assignment for cases where the context does not provide enough information, WP-ASSIGN-DEF, resulting in:

$$\begin{aligned}\emptyset * [R] \vdash\ & \exists \ell, t.\ \mathrm{loc}(v_r, \ell) * \ell \mapsto t\ * \\ & \quad \big(\ell \mapsto t \mathrel{-\!\!*} \mathbf{wp}\ (v_r{\to}\mathsf{s} := v_a)\ \big\{\_.\ \mathbf{wp}\ (v_r{\to}\mathsf{e} := v_b)\ \{\_.\ \mathsf{True}\}\big\}\big)\end{aligned}$$

Next, since the goal has become an existential quantifier, we apply the rule for existential quantification, ABD-EXISTS (twice). It adds existential quantifiers for ℓ and t to the precondition R. Subsequently, since $\mathrm{loc}(v_r, \ell)$ and $\ell \mapsto t$ are not contained in the context $\emptyset$, they are also added to the precondition R—with ABD-PURE-MISSING for $\mathrm{loc}(v_r, \ell)$ and ABD-RES-MISSING for $\ell \mapsto t$.

We are left to derive

$$\mathrm{loc}(v_r, \ell) * [R'] \vdash \ell \mapsto t \mathrel{-\!\!*} \mathbf{wp}\ (v_r{\to}\mathsf{s} := v_a)\ \big\{\_.\ \mathbf{wp}\ (v_r{\to}\mathsf{e} := v_b)\ \{\_.\ \mathsf{True}\}\big\}$$

and have already constructed $R \triangleq \exists \ell, t.\ \mathrm{loc}(v_r, \ell) * \ell \mapsto t * R'(\ell, t)$ for some R' that is yet to be determined. From here, the derivation proceeds as in Version 1 until we eventually arrive at the postcondition True:

$$\mathrm{loc}(v_r, \ell),\ \ell \mapsto t[\mathsf{s} := v_a,\ \mathsf{e} := v_b] * [R'] \vdash \mathsf{True}$$

We finish with ABD-TRUE, resolving $R' \triangleq \mathsf{True}$. Thus, we have inferred the precondition $R = (\exists \ell, t.\ \mathrm{loc}(v_r, \ell) * \ell \mapsto t)$ such that $\{R\}\ \texttt{init}(v_r, v_a, v_b)\ \{\_.\ \mathsf{True}\}$.

Version 3: Abducting the postcondition. Lastly, let us additionally infer the *postcondition.* Intuitively, the postcondition should be the context at the end of the derivation (*i.e.,* "Δ" in ABD-TRUE), since it describes all knowledge that we have at the end of the function. However, note that the judgment $\Delta * [R] \vdash \mathbf{wp}\ e\ \{\Phi\}$ does not have an *output* for the postcondition, only for the precondition R. The reason is that such a dedicated postcondition output is not needed—rather, we can encode the postcondition as part of the precondition R by expressing specifications as *predicate transformers.*

A predicate transformer is a function T from postconditions Φ to preconditions $T\,\Phi$ such that $\forall \Phi.\ \{T\,\Phi\}\ e\ \{v.\, \Phi(v)\}$. Predicate transformers are an alternative to Hoare triples for specifying functions, where we use existential quantification (+ separating conjunctions) to express (Hoare triple) preconditions, and universal quantification (+ magic wands) to express (Hoare triple) postconditions. We use colors to highlight the precondition parts (in orange) and postcondition parts (in violet). For example, the desired specification of `init`, namely $\{\mathrm{loc}(v_r, \ell) * \ell \mapsto t\}\ \mathtt{init}(v_r, v_a, v_b)\ \{\_.\mathrm{loc}(v_r, \ell) * \ell \mapsto t[\mathsf{s} := v_a, \mathsf{e} := v_b]\}$, can be expressed equivalently as the predicate transformer:

$$T_{\mathtt{init}}(\Phi) \triangleq \underbrace{\exists \ell, t. \mathrm{loc}(v_r, \ell) * \ell \mapsto t}_{\text{precondition}} * (\underbrace{\mathrm{loc}(v_r, \ell) * \ell \mapsto t[\mathsf{s} := v_a,\ \mathsf{e} := v_b]}_{\text{postcondition}} \twoheadrightarrow \Phi\,())$$

In this case, the value v_r should be a location ℓ storing a struct with contents t (precondition), and afterwards the location ℓ stores the updated contents $t[\mathsf{s} := v_a,\ \mathsf{e} := v_b]$ (postcondition).

To *infer* a predicate transformer specification, we treat the predicate Φ abstractly during the abduction. Then, the resulting precondition $R(\Phi)$ is a predicate transformer, because

$$\forall \Phi.\ \emptyset * [R\,\Phi] \vdash \mathbf{wp}\ e\ \{v.\ \Phi\,v\} \quad \textit{implies} \quad \forall \Phi.\ \{R\,\Phi\}\ e\ \{v.\ \Phi\,v\}$$

In the case of $\mathtt{init}(v_r, v_a, v_b)$, we abduct $\emptyset * [R] \vdash \mathbf{wp}\ \mathtt{init}(v_r, v_a, v_b)\ \{v.\ \Phi\,v\}$ where Φ is abstract. We eventually hit the postcondition $\Phi\,()$ (in place of "True" in Version 2). At this point, we face $\mathrm{loc}(v_r, \ell),\ \ell \mapsto t[\mathsf{s} := v_a,\ \mathsf{e} := v_b] * [R'] \vdash \Phi\,()$. To finish the derivation, we "revert" the context before the post with ABD-END ("$\forall\vec{x}$" explained below), resulting in the solution $R'(\Phi) \triangleq \mathrm{loc}(v_r, \ell) * \ell \mapsto t[\mathsf{s} := v_a,\ \mathsf{e} := v_b] \twoheadrightarrow \Phi\,()$. Plugging this into the top-level inferred specification $R(\Phi) \triangleq \exists \ell, t.\ \mathrm{loc}(v_r, \ell) * \ell \mapsto t * R'(\Phi)$, we see that $R(\Phi)$ coincides exactly with the specification $T_{\mathtt{init}}(\Phi)$. Thus, we have inferred $\{\mathrm{loc}(v_r, \ell) * \ell \mapsto t\}\ \mathtt{init}(v_r, v_a, v_b)\ \{\_.\ \mathrm{loc}(v_r, \ell) * \ell \mapsto t[\mathsf{s} := v_a,\ \mathsf{e} := v_b]\}$ as the specification, stated as a predicate transformer instead of a Hoare triple.

18.2 Existential Quantification

The goal-directed proof search presented above is overly simplistic in a key dimension: the treatment of existential quantification. With the rule ABD-EXISTS, it—so far—always lifts existential quantifiers to the precondition. But there are really two options for existential quantifiers: (1) lift the quantifier to the precondition (as above) or (2) instantiate the quantifier. For abductive deductive verification to work, we need both options. With `init`, we have already seen a case where we *must* lift the existential quantifier to the precondition, because the context $\emptyset$ is empty (in §18.1). To illustrate when we want the second option, we consider the function `mkrange`.

For `mkrange`, we want to infer the following specification:

$$T_{\mathtt{mkrange}}(\Phi) \triangleq \mathrm{True} * (\forall \ell, v_r.\ \mathrm{loc}(v_r, \ell) * \ell \mapsto \{\ \mathsf{s} = v_a; \mathsf{e} = v_b\ \} \twoheadrightarrow \Phi\,v_r)$$

The precondition is True and the postcondition ensures that the return value v_r is a location ℓ storing a correctly initialized range. To understand why inferring this specification requires existential instantiation, we unfold the weakest precondition semantics of $\mathbf{wp}\ \mathtt{mkrange}(v_a, v_b)\ \{\Phi\}$ (*i.e.*, we apply weakest precondition rules such as the ones in Fig. 18.3 in a non-goal directed fashion), because it reveals the existential quantifiers that we must instantiate. To indicate that the reasoning is not goal-directed (as opposed to the entailments in Fig. 18.3), we write $P \vDash Q$ for semantic entailment here (*i.e.,* for the standard Iris entailment $P \vdash Q$ from §4.3). Specifically, we unfold mkrange, the let-binding, the allocation new(), and the init-call:

$$\begin{aligned}
&\mathbf{wp}\ \mathtt{mkrange}(v_a, v_b)\ \{v.\ \Phi\, v\}\\
\dashv\ &\mathbf{wp}\ \mathsf{let}\ \mathsf{r} = \mathsf{new}()\ \mathsf{in}\ \mathtt{init}(\mathsf{r}, v_a, v_b);\ \mathsf{r}\ \{v.\ \Phi\, v\}\\
\dashv\ &\mathbf{wp}\ \mathsf{new}()\ \left\{v_r.\ \mathbf{wp}\ \mathtt{init}(v_r, v_a, v_b)\ \{\_.\ \Phi\, v_r\}\right\}\\
\dashv\ &\underbrace{\forall \ell, v_r.\ \mathrm{loc}(v_r, \ell) * \ell \mapsto \epsilon \mathrel{-\!\!*}}_{\text{from } \mathsf{new}()}\\
&\underbrace{\exists \ell, t.\ \mathrm{loc}(v_r, \ell) * \ell \mapsto t * (\mathrm{loc}(v_r, \ell) * \ell \mapsto t[\mathsf{s} := v_a,\ \mathsf{e} := v_b] \mathrel{-\!\!*} \Phi\, v_r)}_{\text{from } \mathtt{init}(v_r, v_a, v_b)}
\end{aligned}$$

Note the existential "$\exists \ell, t.$" arising from the call to init. We should not lift this quantifier into the precondition, because its value *depends* on the result of allocating a fresh location inside mkrange. Instead, we should *instantiate* this quantifier with the location ℓ obtained from new() (and t with ϵ).

Instantiating existentials. To instantiate existential quantifiers during goal-directed proof search, we will introduce a new goal $\mathbf{ex}(x.\ G\, x)$ in §19 such that, semantically, $\exists x.\ G\, x \vDash \mathbf{ex}(x.\ G\, x)$. The goal **ex** is triggered indirectly for function calls $f(\vec{v})$ to instantiate existential quantifiers in the spec of f.

More specifically, for a function call $f(\vec{v})$, we use WP-CALL. It searches for a specification T for f in the context (with the side condition $\mathrm{spec}(f, T)$) and then creates the function application goal "$\mathbf{apply}(T)\{\Phi\}$" where, semantically, **apply** simply applies the predicate transformer specification T to the postcondition Φ (*i.e.,* semantically $T\,\Phi \vDash \mathbf{apply}(T)\{\Phi\}$). As we will see in §19, the proof search then turns $\mathbf{apply}(T)\{\Phi\}$ roughly into "$\mathbf{ex}(\_.\ T\,\Phi)$" to instantiate the existential quantifiers that occur in the predicate transformer specification T.[4]

For example, in the inference of the specification of mkrange, we eventually encounter the goal:

$$\mathrm{loc}(v_r, \ell),\ \ell \mapsto \epsilon * [R] \vdash \mathbf{ex}(\_.\ \exists x, y.\ \mathrm{loc}(v_r, x) * x \mapsto y * G_{\mathtt{rest}}\, x\, y)$$

where **ex** contains the precondition part of $T_{\mathtt{init}}$ and we summarize the remainder as "$G_{\mathtt{rest}}$". The rules for **ex** then iterate on this goal to instantiate $x \triangleq \ell$, instantiate $y \triangleq \epsilon$, resolve the preconditions $\mathrm{loc}(v_r, \ell)$ and $\ell \mapsto \epsilon$, and leave as the remaining goal $G_{\mathtt{rest}}\, \ell\, \epsilon$. As a result, we arrive at the desired specification $T_{\mathtt{mkrange}}(\Phi)$ in the end (after continuing the abduction for $G_{\mathtt{rest}}\, \ell\, \epsilon$).

[4] When we write $\mathbf{ex}(\_.\ G)$, then the underscore "_" indicates that G is closed and that the list of quantified variables is currently empty. As $\mathbf{ex}(\_.\ G)$ processes the goal G, it adds existentially quantified variables to the binder. For more details on **ex**, see §19.2.

∃∀-specifications. One may wonder why we bother with existential quantifier instantiation and do not use the predicate transformer that results from semantically unfolding $\mathbf{wp}\ \mathtt{mkrange}(v_a, v_b)\ \{v.\ \Phi\, v\}$, namely

$$\begin{aligned}
T'_{\mathtt{init}}(\Phi) \triangleq\ &\forall \ell, v_r.\ \mathrm{loc}(v_r, \ell) * \ell \mapsto \epsilon \mathrel{-\!\!*}\\
&\qquad \exists \ell, t.\ \mathrm{loc}(v_r, \ell) * \ell \mapsto t * (\mathrm{loc}(v_r, \ell) * \ell \mapsto t[\mathsf{s} := v_a,\ \mathsf{e} := v_b] \mathrel{-\!\!*} \Phi\, v_r),
\end{aligned}$$

as the specification for mkrange. The underlying issue is that while predicate transformers can, in principle, contain arbitrary quantifier alternations, a predicate transformer T that goes beyond a single ∃∀-alternation can barely be considered a specification: it alternates preconditions (∃) with postconditions (∀), thus making it difficult to understand what T *means* as a specification.

For example, from a caller-perspective, the predicate transformer $T'_{\texttt{init}}$ allows one to assume ownership of a new location ℓ storing an empty record ϵ, then requires one to give it back for some contents t, and finally allows one to assume it again for updated fields $t[\mathsf{s} := v_\mathsf{a},\ \mathsf{e} := v_\mathsf{b}]$.

To avoid such confusion between pre- and postconditions, a key design decision of Quiver is to restrict ourselves to a single ∃∀-alternation. Thus, in the resulting predicate transformers, preconditions (∃) always appear before postconditions (∀). The ∃∀-alternation is *enforced* by an asymmetry in our quantifier rules: ABD-EXISTS adds existential quantifiers to the precondition, but ABD-ALL does not add any universal quantifiers to the precondition; it only introduces them in the goal. The rule that adds universal quantifiers to the precondition is ABD-END. As we have seen above, it is used only at the very end. It adds those universal quantifiers that have been introduced by ABD-ALL in the goal (because they are potentially now contained in the context Δ).

18.3 Specification Sketches

So far, we have discussed how we can infer specifications *without any user guidance*. The resulting specifications describe “low-level” memory footprints, but they do not yet use any abstract predicates (*i.e.*, user defined predicates for data types). Abstract predicates are, however, a hallmark of separation logic verification (*e.g.*, see the linked-list predicate in §2.2). For instance, for the range data type, a standard approach would be to conceal the implementation details behind a predicate $\mathrm{range}(v_\mathsf{r}, n_\mathsf{s}, n_\mathsf{e})$, which means “$v_\mathsf{r}$ is the range $[n_\mathsf{s}, n_\mathsf{e})$”. To guide the inference towards “high-level” specifications with abstract predicates, we integrate *specification sketches* into abductive deductive verification.

Specification sketches. To explain what sketches are and how we integrate them, we continue with the range example. We define the abstract predicate:

$$\begin{aligned}\mathrm{range}(v_\mathsf{r}, n_\mathsf{s}, n_\mathsf{e}) \triangleq \exists \ell, v_\mathsf{s}, v_\mathsf{e}.\, &\mathrm{loc}(v, \ell) * \ell \mapsto \{\mathsf{s} = v_\mathsf{s}; \mathsf{e} = v_\mathsf{e}\}\\ &* \mathrm{int}(v_\mathsf{s}, n_\mathsf{s}) * \mathrm{int}(v_\mathsf{e}, n_\mathsf{e}) * 0 \le n_\mathsf{s} \le n_\mathsf{e}\end{aligned}$$

The predicate ensures that the s-field and e-field are integers n_s and n_e, and that the integer bounds form a valid, non-negative range by imposing $0 \le n_\mathsf{s} \le n_\mathsf{e}$.

To infer specifications involving the range-predicate, we add sketches to the implementation of the range operations. A sketch is an inline assertion “$\mathrm{assert}(\ldots)$”, which describes *part of the logical state* at the program point, and which may use question marks “?” to leave holes in the description. For example, in `init`, we add the assertion “$\mathrm{assert}(\mathrm{range}(\mathrm{r}, ?, ?))$” to mean r is some range $[?, ?)$ at the end of the `init` function. The idea is that the proof search then takes this sketch into account and adjusts the resulting specification. That is, for `init`, the inferred specification becomes:

$$\begin{aligned}T^{\mathsf{ran}}_{\texttt{init}}(\Phi) \triangleq\ &\exists \ell, t, n_\mathsf{a}, n_\mathsf{b}.\, \mathrm{loc}(v_\mathsf{r}, \ell) * \ell \mapsto t * \mathrm{int}(v_\mathsf{a}, n_\mathsf{a}) * \mathrm{int}(v_\mathsf{b}, n_\mathsf{b}) * 0 \le n_\mathsf{a} \le n_\mathsf{b}\\ &* (\mathrm{range}(v_\mathsf{r}, n_\mathsf{a}, n_\mathsf{b}) \mathrel{-\!\!*} \Phi())\end{aligned}$$

The precondition is extended by three new assumptions, namely $\mathrm{int}(v_\mathsf{a}, n_\mathsf{a})$ and $\mathrm{int}(v_\mathsf{b}, n_\mathsf{b})$ requiring v_a and v_b to be integers n_a and n_b, and $0 \le n_\mathsf{a} \le n_\mathsf{b}$ to impose the range constraint on the integers. The postcondition is changed to indicate that v_r stores the range $[n_\mathsf{a}, n_\mathsf{b})$ after calling the function.

Abducting specification sketches. As far as abductive deductive verification is concerned, every sketch corresponds to a separation logic proposition with existential quantifiers for question marks. For example, the sketch "assert(range(r, ?, ?))" in `init` corresponds to "$\exists x, y.\ \mathrm{range}(v_r, x, y)$". We use the sketch to update the separation logic state at the point of the assertion.

Concretely, to integrate sketches into $\Delta * [R] \vdash G$, we introduce a new goal $\mathbf{assert}(x.\ P\,x)\{G\}$. When we encounter $\mathbf{assert}(x.\ P\,x)\{G\}$, we first (1) *prove* $P\,x$ for some x—abducting additional preconditions where necessary—and, subsequently, (2) *assume* $P\,x$ for the remainder of the inference. To deal with the existential "x" in the sketch "$\exists x.\ P\,x$", we define $\mathbf{assert}(x.\ P\,x)\{G\}$ using **ex**, roughly as "$\mathbf{ex}(x.\ P\,x * (P\,x \twoheadrightarrow G))$" (see §19). Here, we use the same pattern as wp-assign and wp-read: we first produce "$P\,x$" and then assume "$P\,x$" again for the remainder of the inference.

For example, inside the derivation of `init`, we encounter (spelling out the question marks as existential quantifiers) $\mathbf{wp}\ (\mathrm{assert}(\exists x, y.\mathrm{range}(v_r, x, y)))\ \{\Phi\}$. We apply (wp-assert) and are confronted with the new **assert**-goal:

$$\mathrm{loc}(v_r, \ell),\ \ell \mapsto t[\mathsf{s} := v_a,\ \mathsf{e} := v_b] * [R'] \vdash \mathbf{assert}(x, y.\ \mathrm{range}(v_r, x, y))\{\Phi\,()\}$$

which boils down to $\mathbf{ex}(x, y.\ \mathrm{range}(v_r, x, y) * (\mathrm{range}(v_r, x, y) \twoheadrightarrow \Phi\,()))$. The *first part*, "$\mathrm{range}(v_r, x, y)\ *$", is handled by (a) unfolding $\mathrm{range}(v_r, x, y)$ and, then, (b) abducting anything missing for proving the body of $\mathrm{range}(v_r, x, y)$—here $\mathrm{int}(v_a, n_a)$, $\mathrm{int}(v_b, n_b)$, and $0 \le n_a \le n_b$—while also instantiating $x \triangleq n_a$ and $y \triangleq n_b$. The *second part*, "$\mathrm{range}(v_r, x, y) \twoheadrightarrow$ ", after x and y have been instantiated, adds $\mathrm{range}(v_r, n_a, n_b)$ to the context (abd-wand-res), which then eventually ends up in the postcondition of $T^{\mathsf{ran}}_{\mathsf{init}}$ (via abd-end).

Sketches *vs.* specifications. Zooming out to the entire range implementation again (in Fig. 18.2), the other two range operations highlight two important benefits of specification sketches over full-fledged specifications. First, we can provide similar sketches for multiple functions, yet obtain different specifications. For example, for the function `size`, we provide the same assertion sketches as for `init`, yet obtain:

$$\begin{aligned} T^{\mathsf{ran}}_{\mathsf{size}}(\Phi) \triangleq\ & \exists n_s, n_e.\ \mathrm{range}(v_r, n_s, n_e) \\ & * (\forall w.\ \mathrm{range}(v_r, n_s, n_e) * \mathrm{int}(w, n_e - n_s) \twoheadrightarrow \Phi\,w) \end{aligned}$$

The precondition $\mathrm{range}(v_r, n_s, n_e)$ arises, because unlike for `init`, when we encounter the first sketch in `size`, the context contains no information about v_r that could be used to prove $\mathrm{range}(v_r, n_s, n_e)$. Thus, the resource is added as a whole to the precondition via abd-res-missing. The postcondition contains the additional information that the return value w is the integer $n_e - n_s$.

Second, abductive deductive verification is *compositional*. The sketch in `init` not only affects `init`, but also `mkrange`. That is, if we infer a specification of `mkrange` again—against the new specification $T^{\mathsf{ran}}_{\mathsf{init}}$—we obtain the following specification *without any additional sketches*:

$$\begin{aligned} T^{\mathsf{ran}}_{\mathsf{mkrange}}(\Phi) \triangleq\ & \exists n_a, n_b.\ \mathrm{int}(v_a, n_a) * \mathrm{int}(v_b, n_b) * 0 \le n_a \le n_b \\ & * (\forall v_r.\ \mathrm{range}(v_r, n_a, n_b) \twoheadrightarrow \Phi\,v_r) \end{aligned}$$

The precondition changes to incorporate the additional assumptions on v_a and v_b, and the postcondition ensures that the return value v_r is the correctly initialized $\mathrm{range}(v_r, n_a, n_b)$ from `init`.

Chapter 19

The Abduction Engine Argon

Having introduced the idea of abductive deductive verification (§18), let us now focus on the first part of Quiver, the *abduction engine* **Argon**. It provides automation for the abduction judgment $\Delta * [R] \vdash G$. Intuitively, $\Delta * [R] \vdash G$ means G holds under the assumption of the context Δ and the additional precondition R. Accordingly, for a context $\Delta = (\Gamma, \Omega_*, \Omega_\Box)$, we define the abduction judgment $\Delta * [R] \vdash G$ as the entailment

$$\Delta * [R] \vdash G \triangleq (\mathop{\Large\ast}_{\phi \in \Gamma} \phi) * (\mathop{\Large\ast}_{Q \in \Omega_*} Q) * (\mathop{\Large\ast}_{Q \in \Omega_\Box} \Box Q) * R \vDash G$$

The context Δ consists of three parts: pure assertions Γ (*e.g.*, $n \geq 0$ and $\mathrm{loc}(v, \ell)$), ownership assertions Ω_*, and persistent assertions $\Omega_\Box$ (§3.3). The assertions in Ω_* and $\Omega_\Box$ are *resources* M, the basic building blocks to describe the program state (*e.g.*, $\ell \mapsto t$ and $\mathrm{range}(v, n, m)$ in §18). The persistent resources in $\Omega_\Box$ remain in the context forever while the resources in Ω_* can be removed.

$$\begin{array}{rrl}
\text{Contexts} & \Delta & ::= (\Gamma, \Omega_*, \Omega_\Box) \qquad (\Gamma \in \mathsf{List}(\mathit{Prop}),\ \Omega \in \mathsf{List}(\mathrm{Resource})) \\
\text{State Prop.} & S & ::= \phi \mid M \mid S_1 * S_2 \mid \exists x.\, S\, x \qquad\qquad\qquad (M \in \mathrm{Resource}) \\
\exists\forall\text{-Pred. Trans.} & T & ::= \exists \vec{x}.\, S\vec{x} * T\vec{x} \mid T_1 \wedge T_2 \mid \textbf{if } \phi \textbf{ then } T_1 \textbf{ else } T_2 \mid \forall \vec{x}.\, S\vec{x} \mathrel{-\!\!*} \Phi\vec{x} \\
\text{Goal} & G & ::= \mathsf{True} \mid \Phi\, v \mid E \mid S * G \mid S \mathrel{-\!\!*} G \mid G_1 \wedge G_2 \\
 & & \mid\ \textbf{if } \phi \textbf{ then } G_1 \textbf{ else } G_2 \mid \exists x.\, Gx \mid \forall x.\, Gx \\
 & & \mid\ \textbf{simpl}(T)\{G\} \mid \textbf{ex}(x.\, Gx) \mid \textbf{bind}(G_1)\{G_2\} \mid \cdots
\end{array}$$

Figure 19.1: The syntax of Argon.

Goal-directed proof search. The key technique that turns Argon's inference rules into an automated abduction algorithm is *goal-directed proof search*: At each step of the abduction with state $\Delta * [R] \vdash G$, Argon matches on the *goal* G and applies the first rule with a matching conclusion and whose side conditions can be proven. After applying the rule, it then recursively proceeds with the premises. The goals that Argon supports are depicted in Fig. 19.1. Below, we discuss which purpose they serve, and how they are dealt with during goal-directed proof search, using the rules in Fig. 19.2 (which include the rules from Fig. 18.3 for convenience). We start with basic goals in §19.1 and proceed with more advanced goals in §19.2.

19.1 Basic Goals

Embedded goals. *Embedded goals* E sit at the heart of Argon. They embed deductive proof systems into Argon such as a weakest precondition calculus

abd-embed
$$\frac{E \dashv G \textit{ when } P \qquad \Delta \vdash P \qquad \Delta * [R] \vdash G}{\Delta * [R] \vdash E}$$

abd-res-ctx
$$\frac{\Delta = \Delta', M \qquad \Delta' * [R] \vdash G}{\Delta * [R] \vdash M * G}$$

abd-res-missing
$$\frac{\Delta * [R] \vdash G}{\Delta * [M * R] \vdash M * G}$$

abd-pure-prove
$$\frac{\Delta \vdash \phi \qquad \Delta * [R] \vdash G}{\Delta * [R] \vdash \phi * G}$$

abd-pure-missing
$$\frac{\Delta, \phi * [R] \vdash G}{\Delta * [\phi * R] \vdash \phi * G}$$

abd-wand-res
$$\frac{\Delta, M * [R] \vdash G}{\Delta * [R] \vdash M \mathrel{-\!\!*} G}$$

abd-wand-pure
$$\frac{\Delta, \phi * [R] \vdash G}{\Delta * [R] \vdash \phi \mathrel{-\!\!*} G}$$

abd-exists
$$\frac{\forall x.\, (\Delta * [R x] \vdash G x)}{\Delta * [\exists x.\, R x] \vdash \exists x.\, G x}$$

abd-all
$$\frac{\forall x.\, (\Delta * [R] \vdash G x)}{\Delta * [R] \vdash \forall x.\, G x}$$

abd-end
$$\Delta * [\forall \vec{x}.\, \Delta \mathrel{-\!\!*} \Phi v] \vdash \Phi v$$

abd-true
$$\Delta * [\mathsf{True}] \vdash \mathsf{True}$$

abd-if-true
$$\frac{\Delta \vdash \phi \qquad \Delta * [R] \vdash G_1}{\Delta * [R] \vdash \textbf{if } \phi \textbf{ then } G_1 \textbf{ else } G_2}$$

abd-if
$$\frac{\Delta, \phi * [R_1] \vdash G_1 \qquad \Delta, \neg\phi * [R_2] \vdash G_2}{\Delta * [\textbf{if } \phi \textbf{ then } R_1 \textbf{ else } R_2] \vdash \textbf{if } \phi \textbf{ then } G_1 \textbf{ else } G_2}$$

abd-conj
$$\frac{\Delta * [R_i] \vdash G_i \textit{ for } i = 1, 2}{\Delta * [R_1 \wedge R_2] \vdash G_1 \wedge G_2}$$

abd-ex
$$\frac{\textbf{ex}(x.\, G_1 x \mid S\, x) \dashv G_2 \textit{ when } P \qquad \Delta \vdash P \qquad \Delta * [R] \vdash G_2}{\Delta * [R] \vdash \textbf{ex}(x.\, G_1\, x \mid S\, x)}$$

abd-simpl
$$\frac{(\forall \Phi.\, T\, \Phi \Rightarrow T'\, \Phi) \qquad \Delta * [R] \vdash G\, T'}{\Delta * [R] \vdash \textbf{simpl}(T)\{T''.\, G\, T''\}}$$

abd-bind
$$\frac{\forall \Phi.\, \Delta * [T\, \Phi] \vdash G_1\, \Phi \qquad \Delta_{\Box} * [R] \vdash G_2\, T}{\Delta * [R] \vdash \textbf{bind}(\Phi.\, G_1\, \Phi)\{T'.\, G_2\, T'\}}$$

abd-fail
$$\Delta * [\mathsf{False}] \vdash G$$

simplify
$$\frac{P_1 \Rightarrow_{\text{norm}} P_2 \qquad P_2 \Rightarrow_{\text{simp}} P_3 \qquad P_3 \Rightarrow_{\text{exist}} P_4}{P_1 \Rightarrow P_4}$$

Figure 19.2: Abduction rules, including the rules from Fig. 18.3. Overlapping abduction rules are applied top-to-bottom, left-to-right.

(in §18) and the type system Thorium (in §20)—using an extensible set of reasoning rules. Concretely, when Argon encounters an embedded goal E (abd-embed), it searches for a reasoning rule $E \dashv G$ *when* P by matching on E and continues with the goal G if the side condition P is provable in the current context Δ. Embedded goals can be weakest preconditions $\textbf{wp}\ e\ \{\Phi\}$, but also other auxiliary judgments. For example, Thorium introduces a judgment "$\textbf{conv}(v : A)(x.\, B\, x)\{\Phi\}$" for type conversion, discussed in §20.2.

Separating conjunction and magic wand. We turn to *separating conjunction* $S * G$ and the *magic wand* $S \mathrel{-\!\!*} G$. The goal $S * G$ instructs Argon to prove the assertion S and then proceed with G while $S \mathrel{-\!\!*} G$ introduces S into the context and then proceeds with G. Argon avoids ambiguity during the proof search by restricting S to *state propositions*, *i.e.*, assertions over the state of the program consisting of resources and pure assertions (see Fig. 19.1). (For efficient proof automation, Sammler et al.[1] employ a similar restriction, but not in the context of abduction.) For $S * G$, we consider the two interesting cases: If $S = \phi$ is a pure assertion, we either prove ϕ through a pure entailment $\Delta \vdash \phi$ (abd-pure-prove), or we add ϕ to the precondition and the context (abd-pure-missing). If $S = M$ is a resource, we either find the resource M in the context (abd-res-ctx), or we add it as a missing assertion to the precondition (abd-res-missing). The other cases for S (namely, $S_1 * S_2$ and $\exists x.\, S'\, x$) are handled by straightforward rules (not shown) which serve to hoist out existential quantifiers and move a ϕ or M to the left side of the goal using associativity of separating conjunction. We adopt a similar restriction for *magic wands* $S \mathrel{-\!\!*} G$.

[1] Sammler et al., "RefinedC: Automating the foundational verification of C code with refined ownership types", 2021 [Sam+21].

It lets us split S into pure assertions ϕ and resources M that are then added to the context (abd-wand-pure resp. abd-wand-res).

Conditionals and conjunctions. We turn to *conditionals* **if** ϕ **then** G_1 **else** G_2 and *conjunctions* $G_1 \wedge G_2$. For a conditional **if** ϕ **then** G_1 **else** G_2, we first try to eliminate it by proving or disproving the condition ϕ (*i.e.,* by applying abd-if-true or the corresponding rule for $\Delta \vdash \neg\phi$). Otherwise, we lift it to the precondition (abd-if).[2] For *conjunctions* $G_1 \wedge G_2$, we lift the conjunction to the precondition (abd-conj). Conceptually, a conjunction means the reason for the choice between both branches is *internal* to the function, *i.e.,* it cannot be influenced by the caller. For example, a call to `malloc` may return `NULL` or a valid pointer. A client cannot influence which one it is, making the predicate transformer for `malloc` a conjunction of both cases (see $T_{\texttt{malloc}}$ in §20.3).

[2] If the inferred preconditions R_1 and R_2 coincide, the conditional can be removed altogether by simplification. Additionally, in some cases, the inferred preconditions R_1 and R_2 can also be joined using a heuristic of the Thorium type system.

Quantifiers and post conditions. We turn to *existential* and *universal* quantification. Existential quantification $\exists x.\, Gx$ in the goal is resolved by adding an existential quantifier to the precondition R (abd-exists) and universal quantification $\forall x.\, Gx$ is resolved by introducing x in the goal but leaving the precondition R unchanged (abd-all). Universal quantifiers are added to the precondition only when we reach the *postcondition goal* $\Phi\, v$ (abd-end). Concretely, when we encounter $\Phi\, v$, we revert those $\vec{x}$ that have been previously introduced in the goal (with abd-all). They are added in front of the context Δ, since the context might refer to them at this point (*e.g.,* v_{r} and ℓ in $T_{\texttt{mkrange}}$ in §18.2).

19.2 Advanced Goals

Simplification. One of the main ways to integrate reasoning about mathematical theories into the Argon proof search is *simplification.*[3] It comes in two forms: (1) a general-purpose simplification judgment $P \Rightarrow Q$ (semantically $Q \vDash P$), which simplifies P into Q and (2) a goal $\mathbf{simpl}(T)\{T'.\, G\, T'\}$ that uses the judgment $P \Rightarrow Q$ to simplify a predicate transformer T (abd-simpl). The simplification judgment $P \Rightarrow Q$ proceeds in three steps[4] (simplify), $P_1 \Rightarrow_{\text{norm}} P_2 \Rightarrow_{\text{simp}} P_3 \Rightarrow_{\text{exist}} P_4$: First, with $P_1 \Rightarrow_{\text{norm}} P_2$, we normalize P_1 into a normal form analogous to $\exists\forall$-predicate transformers (*e.g.,* by lifting out existentials). Then, with $P_2 \Rightarrow_{\text{simp}} P_3$, we simplify pure propositions in P_2. Finally, with $P_3 \Rightarrow_{\text{exist}} P_4$, we instantiate existentials based on *equalities* $a = b$ in P_3. For example, the simplification proceeds as follows

[3] Additionally, Argon allows integrating solvers for mathematical theories via $\Delta \vdash \phi$.

[4] We may apply simplification multiple times to benefit from existential instantiations for further simplification and vice versa. By default, the implementation of Quiver simplifies twice.

$$\begin{array}{ll} & x \geq 0 * (\exists y.\, 4x = 4y * Ty) \\ \Rightarrow_{\text{norm}} & \exists y.\, x \geq 0 * 4x = 4y * Ty \\ \Rightarrow_{\text{simp}} & \exists y.\, x \geq 0 * x = y * Ty \\ \Rightarrow_{\text{exist}} & x \geq 0 * Tx \end{array}$$

Normalization lifts the existential to the outside, then simplification removes the multiplication with 4 (since $4x = 4y$ *iff.* $x = y$), and finally instantiation resolves $y \triangleq x$ based on the equality.

As illustrated above, the simplification step $P_3 \Rightarrow_{\text{simp}} P_4$ integrates mathematical theories. It uses (a) *pure abduction rules* $\phi \Rightarrow_{\text{simp}} \psi$, (b) *rewriting simplification rules* $\phi[a] \Rightarrow_{\text{simp}} \phi[b]$ if $a = b$, and (c) *solvers* $\phi \Rightarrow_{\text{simp}}$ True if ϕ. We use simplification rules and solvers for, *e.g.,* integers, injective functions, and lists, and the simplification rules can be extended as needed.

$$\mathbf{ex}(x.\, \exists y.\, G\,x\,y \mid S\,x) \dashv \mathbf{ex}(x, y.\, G\,x\,y \mid S\,x) \tag{ex-exists}$$

$$\mathbf{ex}(x.\, \phi * G\,x \mid S\,x) \dashv \phi * \mathbf{ex}(x.\, G\,x \mid S\,x) \tag{ex-pure}$$

$$\mathbf{ex}(x.\, \phi\,x * G\,x \mid S\,x) \dashv \mathbf{ex}(x.\, G\,x \mid \phi\,x * S\,x) \tag{ex-pure-blocked}$$

$$\mathbf{ex}(x, y.\, x = o * G\,x\,y \mid S\,x\,y) \dashv \mathbf{ex}(\_.\, G' \mid \mathsf{True})\ \textit{when}\ (\exists x, y.\, x = o * S\,x\,y * G\,x\,y) \Rightarrow G' \tag{ex-eq}$$

$$\mathbf{ex}(x, y.\, G\,x\,y \mid S\,x\,y) \dashv \exists y.\, \mathbf{ex}(x.\, S\,x\,y * G\,x\,y \mid \mathsf{True}) \tag{ex-lift}$$

$$\mathbf{ex}(\_.\, G \mid \mathsf{True}) \dashv G \tag{ex-done}$$

Figure 19.3: Selection of existential instantiation rules for $\mathbf{ex}(x.\, Gx \mid Sx)$. Overlapping rules are applied top-to-bottom. The underscore "_" indicates that the list of quantified variables is currently empty.

Existential instantiation. As mentioned in §18.2, the goal $\mathbf{ex}(x.\ G\,x)$ is used for existential instantiation. It has the following key characteristics: it is agnostic to the order of existential quantifiers in G; it is agnostic to the order of conjuncts in a separating conjunction; it inherits the simplification of $P \Rightarrow Q$; and it allows us to destruct existential quantifiers in the context. Moreover, similar to embedded goals E, it is extensible in the sense that additional rules can be added. To achieve these characteristics, we generalize $\mathbf{ex}(x.\ G\,x)$ to the form "$\mathbf{ex}(x.\, G\,x \mid S\,x)$", where the state proposition S collects "blocked" assertions (explained below) and define $\mathbf{ex}(x.\ G\,x) \triangleq \mathbf{ex}(x.\, G\,x \mid \mathsf{True})$.

The proof search for $\mathbf{ex}(x.\, G\,x \mid S\,x)$ proceeds by applying existential instantiation rules of the form $\mathbf{ex}(x.\, G_1\,x \mid S\,x) \dashv G_2$ *when* P (abd-ex), analogous to the rule for embedded goals (abd-embed in Fig. 18.3). We discuss the most important rules, depicted in Fig. 19.3 (and omit straightforward rules such as applying associativity for separating conjunction):

1. For *existential quantifiers* $\exists y.\, G\,y$, we add a binding (ex-exists).

2. For *pure propositions* ϕ that do not depend on x, we lift them out of the goal (ex-pure). For pure propositions ϕ that make x precise (*e.g.*, equality), we use simplification to instantiate the existential (ex-eq). For pure propositions that depend on x but do not lead to instantiation, we put them on the "blocked stack", meaning we add them to the state goal S (ex-pure-blocked). The blocked stack allows us to traverse further into the goal G and, thereby, we can be agnostic about the order of conjuncts in a separating conjunction. A blocked assertion may become unblocked when we can instantiate an existential. Thus, ex-eq adds the "blocked stack" back into the goal.

3. For *resources* M, each instantiation of Argon can handle them as desired by extending the rules of $\mathbf{ex}(x.\, G_1\,x \mid S\,x) \dashv G_2$ *when* P. For example, Thorium adds rules for its resources, type assignments (see §20).

Finally, we have rules for when the goal G is *stuck* in the sense that no other rule applies: if there are existentials left to instantiate (ex-lift), we lift one of them outside and, once there are none left (ex-done), we continue with G.

Sequential composition. The goal $\mathbf{bind}(\Phi.\, G_1\,\Phi)\{T.\, G_2\,T\}$ implements *sequential composition* of abduction goals (abd-bind). It works as follows: First, it will abduct G_1 for an arbitrary postcondition Φ. The result of this abduction is a predicate transformer T. Then, it will abduct G_2, passing it the newly abducted predicate transformer T as an argument. (To implement sequential composition, abd-bind has to avoid duplicating ownership, and therefore gives only the persistent part of the context, written $\Delta_{\Box}$, to G_2.) In other words, **bind** makes abduction available *inside of an abduction.*

With **bind**, we have all the pieces needed to define the goals for applying predicate transformers $\mathbf{apply}(T)\{\Phi\}$ (from §18.2) and specification sketches $\mathbf{assert}(x.\, S\, x)\{\Phi\}$ (from §18.3):

$$\mathbf{apply}(T)\{\Phi\} \triangleq \mathbf{bind}(\Psi.\, \mathbf{ex}(\_.\, T\, \Psi))\{T'.\, \mathbf{simpl}(T')\{T''.\, T''\, \Phi\}\}$$
$$\mathbf{assert}(x.\, S\, x)\{\Phi\} \triangleq \mathbf{bind}(\Psi.\, \mathbf{ex}(x.\, S\, x * (S\, x \mathrel{-\!\!*} \Psi)))\{T'.\, \mathbf{simpl}(T')\{T''.\, T''\, \Phi\}\}$$

Both goals have a similar structure. For **apply**, we instantiate the existentials in T using **ex**, and for **assert**, we instantiate the existentials in the sketch "$S\, x$" using **ex**. We wrap the instantiation of existentials in the sequential composition **bind** to "cleanup" after **ex** with a simplification **simpl**. That is, since the rules for existential instantiation are extensible (abd-ex), they can trigger arbitrary auxiliary goals, which may (indirectly) add existential quantifiers to the precondition R. By simplifying afterwards, we can potentially eliminate some of these quantifiers (*e.g.,* one goal might add "$\exists n.\ \cdots$" and another might add "$n = 0$", which is then simplified by picking $n \triangleq 0$).

Loops. Argon does not infer loop invariants, but supports loops with manually provided loop invariants (without sketches) as one would also provide in deductive verification techniques like RefinedC. For a given loop invariant S_{inv}, the proof search proceeds in four steps: (1) when we reach the loop, we abduct the invariant S_{inv} using **ex**; (2) we abduct the body of the loop assuming the loop invariant S_{inv}; (3) before the next iteration, we re-establish S_{inv} again using **ex**; finally, (4) we check that S_{inv} is indeed a loop invariant by ensuring the abduction of the loop body did not require any additional preconditions.

Failure. Lastly, if no other Argon rule applies, the inference *fails* (abd-fail). In this case, Argon terminates by inserting a marker into the precondition and provides the partial inferred precondition to the user. In impossible cases (*e.g.,* a location is `NULL`, but we are supposed to provide ownership), the marker can contain information explaining what went wrong, provided by the Argon instantiation. To remain sound, the marker is semantically interpreted as False, as indicated in the rule abd-fail.

Chapter 20

The Type System Thorium

In §18, we have seen how to apply abductive deductive verification to a simple separation logic for λ_{expo}. To scale to the complexities of C, we instantiate Argon with a richer separation logic below, **Thorium**. Thorium, following in the footsteps of RefinedC,[1] is a separation logic-based type system. We first explain the approach of using refinement types in separation logic (§20.1), then we discuss how they integrate with abductive deductive verification (§20.2), and finally we return to the range example from §18 to illustrate how Thorium enables compositional specification inference for C programs (§20.3).

[1] Sammler et al., "RefinedC: Automating the foundational verification of C code with refined ownership types", 2021 [Sam+21].

The Caesium language. Before we dive into Thorium, let us first briefly discuss Caesium, the language that Quiver and RefinedC use to model C:

$$\begin{array}{rrcl} \text{Left Expr.} & p, q & ::= & \mathtt{x} \mid \ell \mid p.\mathtt{m} \mid p{+}e \mid *p \qquad\qquad (\mathtt{x} \in \mathit{Var}, \ell \in \mathit{Loc}) \\ \text{Right Expr.} & e & ::= & v \mid *p \mid \&p \mid e_1 \odot e_2 \mid \ominus e \mid e_1(e_2) \mid e_1 \mathbin{?} e_2 : e_3 \\ \text{Statements} & s & ::= & p \leftarrow e;\ s \mid e; s \mid \mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \mid \mathbf{return}\ e \mid \mathbf{goto}\ \mathit{lb} \\ \text{Function Body} & b & ::= & \emptyset \mid \mathit{lb} \mapsto s, b \end{array}$$

We distinguish three main syntactic categories: *left expressions* p, which evaluate to a location; *right expressions* e, which evaluate to a value; and *statements* s, which eventually return a value (or diverge). Left expressions support pointer arithmetic $p+e$, field accesses $p.\mathtt{m}$, and dereferences $*p$. Right expressions support dereferencing left expressions $*p$ (since they evaluate to locations), taking the address of a left expression $\&p$ (effectively a no-op directly returning the location p evaluates to), binary operations $e_1 \odot e_2$, unary operations $\ominus e_1$, function applications $e_1(e_2)$, and inline conditionals $e_1 \mathbin{?} e_2 : e_3$. Statements support assignment to a left-expression $p \leftarrow e;\ s$, expression statements $e; s$, conditionals, returns, and gotos. (There is no statement for declaring a new stack variable, because all stack variables are allocated at the beginning of a function call.) Functions—or rather their implementations—are represented as control flow graphs, as finite maps from block-labels to statements. Loops such as `while`-loops and `for`-loops are translated into this control flow graph representation by the Quiver frontend.

20.1 Separation Logic with Refinement Types à la RefinedC

Instead of abstract predicates $\mathrm{P}(v, x)$ and points-to assertions $\ell \mapsto v$ (as we considered for λ_{expo} in §18), the resources in Thorium are *type assignments*. They are of the form $v \triangleleft_v A$ (read "v is an A") and $\ell \triangleleft_l A$ (read "ℓ stores an A"; semantically $(\exists v.\ \ell \mapsto v \ *\ v \triangleleft_v A) \vDash \ell \triangleleft_l A$). For each type A, they have an

$$
\begin{array}{rrcl}
\text{Types} & A, B & ::= & \mathsf{void} \mid \mathsf{null} \mid \mathsf{num}\texttt{[}it\texttt{]}\, n \mid \mathsf{any}\, n \mid \mathsf{zeros}\, n \mid \mathsf{value}\, n\, v \\
 & & \mid & \exists x.\, A\, x \mid A * P \mid \mathsf{own}\, \ell\, A \mid \mathsf{optional}\, \phi\, A \mid \mathsf{fn}\, T \mid \vec{x} \mathbin{@} \mathrm{P} \\
 & & \mid & \mathsf{struct}\texttt{[}s\texttt{]}\, \vec{A} \mid \mathsf{array}[\mathrm{P}]\, xs \mid \cdots \\
\text{Resources} & M, N & ::= & v \triangleleft_v A \mid \ell \triangleleft_l A \mid \mathsf{block}\, \ell\, n \mid \cdots \\
\text{Embed. Goal} & E & ::= & \mathbf{rwp}(e)\{v, A.\, \Phi\, v\, A\} \mid \mathbf{lwp}(p)\{\ell, A.\, \Phi\, \ell\, A\} \\
 & & \mid & \mathbf{swp}(s)\{v, A.\, \Phi\, v\, A\} \mid \mathbf{cast}\, (it_2)_{it_1} (v : A)\{w, B.\, \Phi\, w\, B\} \\
 & & \mid & \mathbf{conv}(v : A)(x.\, B\, x)\{\Phi\} \mid \mathbf{call}(v : A)(w : B)\{\Phi\} \mid \cdots
\end{array}
$$

Figure 20.1: The Thorium types, resources, and embedded goals.

interpretation in terms of more traditional separation logic assertions. For example, the resource $v \triangleleft_v \mathsf{own}\, \ell\, (\mathsf{num}\texttt{[int]}\, n)$ means "v is an owned pointer ℓ storing the `int`-integer n", which boils down to

$$v \triangleleft_v \mathsf{own}\, \ell\, (\mathsf{num}\texttt{[int]}\, n) \Leftrightarrow v = \ell * \ell \mapsto n * n \in \texttt{int}.$$

Types. The types of Thorium are depicted in Fig. 20.1. We explain the most important types by returning to the range data type (from §18.3). In C, it would be declared as

```
typedef struct ran { int low; int high } *range;
```

The analogous Thorium type for the predicate range$(v, n_\mathsf{s}, n_\mathsf{e})$ (from §18.3) is:

$$(n_\mathsf{s}, n_\mathsf{e}) \mathbin{@} \mathrm{range} \equiv_{\mathrm{ty}} \exists \ell. \mathsf{own}\, \ell\, (\mathsf{struct}\texttt{[ran]}\, [A_\mathsf{s}; A_\mathsf{e}]) * 0 \leq n_\mathsf{s} \leq n_\mathsf{e} * \mathsf{block}\, \ell\, sz_{\texttt{ran}}$$
$$\textit{where } A_\mathsf{s} \triangleq \mathsf{num}\texttt{[int]}\, n_\mathsf{s},\ A_\mathsf{e} \triangleq \mathsf{num}\texttt{[int]}\, n_\mathsf{e},\ \textit{and } sz_{\texttt{ran}} \triangleq \mathbf{sizeof}(\mathbf{struct}\ \texttt{ran})$$

Types of the form $\vec{x} \mathbin{@} \mathrm{P}$ correspond to user-defined abstract predicates and are defined via a (possibly recursive) equation $\vec{x} \mathbin{@} \mathrm{P} \equiv_{\mathrm{ty}} A$. The type $(n_\mathsf{s}, n_\mathsf{e}) \mathbin{@} \mathrm{range}$ ensures that its values are owned pointers ℓ (via "$\mathsf{own}\, \ell\, A$") to a `ran`-struct (via "$\mathsf{struct}\texttt{[}s\texttt{]}\, \vec{A}$") with two fields: `s` containing `int`-integer n_s (via "$\mathsf{num}\texttt{[}it\texttt{]}\, n$"), and `e` containing the `int`-integer n_e. To hide the location ℓ, we use type-level existential quantification "$\exists x.\, A\, x$" and, to impose the bounds constraint $0 \leq n_\mathsf{s} \leq n_\mathsf{e}$, we use type-level separating conjunction "$A * P$". Besides the bounds constraint, the type carries an additional constraint: the resource "$\mathsf{block}\, \ell\, n$". It tracks the length of dynamically allocated blocks (*e.g.*, with `malloc`) to ensure that ownership of the entire block is given up when freeing the location ℓ (see §20.3).

Typing rules. In Thorium, instead of the standard weakest precondition $\mathbf{wp}\ e\ \{v.\, \Phi\, v\}$, we use *typed weakest preconditions*: their postcondition Φ is about the resulting value v *and, additionally,* its type A. We have *three* such weakest preconditions, one for each syntactic category of Caesium: $\mathbf{rwp}(e)\{v, A.\, \Phi\, v\, A\}$, $\mathbf{lwp}(p)\{\ell, A.\, \Phi\, \ell\, A\}$, and $\mathbf{swp}(s)\{v, A.\, \Phi\, v\, A\}$ (where v is the **`return`**-value). As we will see shortly, these typed weakest preconditions have a straightforward encoding in terms of regular weakest preconditions. The main reason why we add the types A to the postcondition Φ is to give more structure to the proof search: they ensure that the result values must always have a type, and that the nested goals in the postcondition Φ can easily make use of this type.[2]

The proof search with typed weakest preconditions works as follows: Instead of vanilla weakest precondition rules, we use *typing rules* in Thorium. A

[2] While the weakest preconditions and the intermediate goals have postconditions Φ with type arguments, for the final specification, we apply (on paper) a transformation that makes them easier to read. The transformation brings the predicate transformers into the form shown in §18, where the postcondition is only over a value. For a predicate transformer T over a value-and-type postcondition, we obtain a predicate transformer over a value postcondition as $\hat{T}(\Phi) \triangleq T(\lambda w, A.\ w \triangleleft_v A \mathrel{-\!\!*} \Phi(w))$. For example, if the resulting value w is of type $\mathsf{num}\texttt{[int]}\, n$, then we write "$w \triangleleft_v \mathsf{num}\texttt{[int]}\, n \mathrel{-\!\!*} \Phi\, w$" in the final predicate transformer.

$$\mathbf{rwp}((it_2)_{it_1}\, e)\{\Phi\} \dashv \mathbf{rwp}(e)\{v, A.\, \mathbf{cast}\ (it_2)_{it_1}(v : A)\{\Phi\}\} \tag{ty-cast-wp}$$

$$\mathbf{cast}\ (\mathtt{size\_t})_{\mathtt{int}}(v : \mathsf{num}[\mathtt{int}]\ n)\{\Phi\} \dashv n \geq 0 * \forall w.\, \Phi\, w\, (\mathsf{num}[\mathtt{size\_t}]\ n) \tag{ty-cast-sizet}$$

$$\mathbf{cast}\ (it_2)_{it_1}(v : \overline{A})\{\Phi\} \dashv \exists n.\, (\overline{A} \sqsubseteq \mathsf{num}[it_1]\ n) * \mathbf{cast}\ (it_2)_{it_1}(v : \mathsf{num}[it_1]\ n)\{\Phi\} \tag{ty-cast-def}$$

$$\mathbf{rwp}(v)\{\Phi\} \dashv v \lhd_v A * \Phi\, v\, A \ \textit{when}\ v \lhd_v A \tag{ty-val}$$

$$\mathbf{rwp}(v)\{\Phi\} \dashv \exists \overline{A}.\, v \lhd_v \overline{A} * \Phi\, v\, \overline{A} \tag{ty-val-missing}$$

$$\mathbf{conv}(v : \mathsf{num}[it]\ n)(x, y, \_.\, \mathsf{num}[x]\ y)\{\Phi\} \dashv \Phi(\lambda x, y, \_.\, x = it * y = n) \tag{ty-conv-int}$$

$$\mathbf{conv}(v : \mathsf{own}\ \ell\ A)(x, y.\, \mathsf{own}\ x\ (B\, x\, y))\{\Phi\} \dashv \ell \lhd_l A \mathrel{-\!\!*} \Phi(\lambda x.\, x = \ell * \ell \lhd_l B\, x\, y) \tag{ty-conv-own}$$

$$\mathbf{call}(v : \mathsf{fn}\, T)(w : B)\{\Phi\} \dashv w \lhd_v B \mathrel{-\!\!*} \mathbf{apply}(T\, w)\{\Phi\} \tag{ty-call}$$

$$\mathbf{ex}(x.\, v \lhd_v B\, x * G\, x \mid S\, x) \dashv \mathbf{rwp}(v)\{w, A.\, \mathbf{conv}(w : A)(x.\, B\, x)\{S'.\, \mathbf{ex}(x.\, S'\, x * G\, x \mid S x)\}\} \tag{ex-conv}$$

$$\mathbf{ex}(x.\, v\, x \lhd_v B\, x * G\, x \mid S\, x) \dashv \mathbf{ex}(x.\, G\, x \mid v\, x \lhd_v B\, x * S\, x) \tag{ex-conv-blocked}$$

Figure 20.2: A selection of Thorium typing rules (ty-) and existential instantiation rules (ex-). Overlapping rules are applied top-to-bottom.

selection of the Thorium typing rules is depicted in Fig. 20.2. There are two kinds of typing rules: (a) structural rules that descend into terms (akin to wp-let in Fig. 18.3) and (b) type-directed rules that match on types to steer the proof search (akin to wp-assign in Fig. 18.3).

To explain how they interact, we consider an example, which dereferences a location ℓ_{k} and casts the resulting integer from `int` to `size_t`:

$$\ell_{\mathrm{k}} \lhd_l \mathsf{num}[\mathtt{int}]\ n_{\mathrm{k}} * [\_] \vdash \mathbf{rwp}((\mathtt{size\_t})_{\mathtt{int}}(*\ell_{\mathrm{k}}))\{\Phi\}$$

First, using a *structural rule* (ty-cast-wp), we descend into the type cast, leaving

$$\ell_{\mathrm{k}} \lhd_l \mathsf{num}[\mathtt{int}]\ n_{\mathrm{k}} * [\_] \vdash \mathbf{rwp}(*\ell_{\mathrm{k}})\{v, A.\, \mathbf{cast}\ (\mathtt{size\_t})_{\mathtt{int}}(v : A)\{\Phi\}\}$$

Once it has been determined that the result of $*\ell_{\mathrm{k}}$ is some value v of type $\mathsf{num}[\mathtt{int}]\ n_{\mathrm{k}}$, we then encounter the following goal in the postcondition:

$$\ell_{\mathrm{k}} \lhd_l \mathsf{num}[\mathtt{int}]\ n_{\mathrm{k}} * [\_] \vdash \mathbf{cast}\ (\mathtt{size\_t})_{\mathtt{int}}(v : \mathsf{num}[\mathtt{int}]\ n_{\mathrm{k}})\{\Phi\}$$

It uses"$\mathbf{cast}\ (it_2)_{it_1}(v : A)\{\Phi\}$", an auxiliary embedded goal for C-level type casts, which is overloaded based on the type A. We use a *type-directed rule* (ty-cast-sizet) to proceed: The rule handles the cast from `int` to `size_t` by checking that the `int`-integer n (here n_{k}) is non-negative and then calling the postcondition with n as a `size_t`-integer.

Modeling Thorium in Iris. To express the types and typing judgments in Iris, we use the same approach as RefinedC. Types A, inspired by RustBelt,[3] are modeled as a triple of two predicates $v \lhd_v A$ and $\ell \lhd_l A$ together with an operator for the size of a type $|A|$ (technically a predicate). The typed weakest preconditions are defined in terms of the standard weakest precondition $\mathbf{wp}\ e\ \{\Phi\}$ of Iris (§3.1), instantiated with the Caesium language. For example, the model of the expression weakest precondition is given by

$$\mathbf{rwp}(e)\{v, A.\, \Phi\, v\, A\} \triangleq \mathbf{wp}\ e\ \{v.\, \exists A.\, v \lhd_v A * \Phi\, v\, A\}.$$

The auxiliary judgments are then defined using the typed weakest preconditions. For example, we define

$$\mathbf{cast}\ (it_2)_{it_1}(v : A)\{w, B.\, \Phi\, w\, B\} \triangleq v \lhd_v A \mathrel{-\!\!*} \mathbf{rwp}((it_2)_{it_1}\, v)\{\Phi\}.$$

[3] Jung et al., "RustBelt: Securing the foundations of the Rust programming language", 2018 [Jun+18a].

As we have seen above, their main purpose is guiding the proof search by providing type-directed rules for how to proceed.

20.2 Abductive Deductive Verification with Types

We discuss how *abductive deductive verification* $\Delta * [R] \vdash \mathbf{wp}\ e\ \{\Phi\}$ interacts with the type system Thorium, using the remaining rules from Fig. 20.2. We focus on the three most important aspects, including how we deal with *incomplete information* about the context Δ. That is—like RefinedC—we use types to guide the proof search, but—unlike RefinedC—the context Δ is incomplete (see §18.1) and, therefore, we may have to infer type assignments as part of the precondition R.

Incomplete information and abstract types. The first aspect is *abstract types* $\overline{A}$, which we introduce to address incomplete information. That is, as discussed in §20.1, the structural rules of Thorium descend into terms. At the leaves, when we encounter a value,[4] $\Delta * [R] \vdash \mathbf{rwp}(v)\{v, A.\ \Phi\, v\, A\}$, we have two options:[5] (a) the value is contained in Δ (TY-VAL) or (b) the type assignment of v should be part of the precondition R. In the latter case, there is an issue: the postcondition Φ demands a type A, but *locally* we do not know yet which type v should have. To solve this issue, we introduce an abstract type $\overline{A}$ (TY-VAL-MISSING), which serves as a placeholder for the type of v. Then, as we continue with the postcondition Φ, we collect constraints on $\overline{A}$. To be precise, we provide "default rules" that impose constraints on $\overline{A}$ when v is *used*—*e.g.*, via the rule TY-CAST-DEF, which requires $\overline{A}$ to be a $\mathtt{num}[it_1]$-type. During simplification $P \Rightarrow Q$ (SIMPLIFY in Fig. 19.2), we use these constraints to instantiate existentials analogous to equalities (in $P \Rightarrow_{\text{exist}} Q$). Thorium has default rules for unary operators, binary operators, pointer dereference, conditionals, struct field access, pointer arithmetic, *etc.*

[4] We treat locations ℓ in left expressions analogously if there is no known type assignment for them.

[5] Technically, there is a third option: extending the precondition R with the assumption $v = w$ for some $w \triangleleft_v A$ in Δ and proceeding with $\Phi\, w\, A$. We exclude this option to limit the search space. In doing so, we follow the footsteps of bi-abduction [Cal+09; Cal+11], which does not infer "$R \triangleq (\ell = r \wedge a = b)$" and "$F \triangleq \text{True}$" as a solution for its bi-abduction judgment $\ell \mapsto a * [R] \vdash r \mapsto b * [F]$ for the same reason.

Type conversion. The second aspect is type conversion, which integrates types into existential instantiation. That is, by extending the rules of $\mathbf{ex}(x.\ G\, x)$, the *type conversion judgment* $\mathbf{conv}(v : A)(x.\ B\, x)\{\Phi\}$ turns the type assignment $v \triangleleft_v A$ into $v \triangleleft_v B\, x$ where "x" is existentially quantified. When we encounter a type assignment in $\mathbf{ex}$, we either (a) trigger type conversion (EX-CONV) or (b) put it on the "blocked-stack" if the value v still depends on x (EX-CONV-BLOCKED). In the first case (EX-CONV), we (1) determine the type of v (using $\mathbf{rwp}$), (2) determine a precondition S for the type conversion to succeed (using $\mathbf{conv}$), and (3) return to the existential instantiation (using $\mathbf{ex}$). For example, TY-CONV-INT constrains the C-level integer type x to it (*e.g.*, `int`, `size_t`, …) and the mathematical integer y to n; TY-CONV-OWN constrains the location x to ℓ and, eventually, will lead to type conversion for $\ell \triangleleft_l B\, x\, y$.

Predicate transformers and joining. The third aspect are predicate transformer specifications. As explained in §18.2, our abductive deductive verification judgment infers predicate transformers T. To integrate them into Thorium, we use a type $\mathsf{fn}\, T$ for function pointers, and we use the rule TY-CALL to call them. The rule TY-CALL turns the goal into $\mathbf{apply}(T\, w)\{\Phi\}$, which—by using

ex internally (see §19.2)—inherits Argon's support for existential instantiation and thus also Thorium's support for type conversion (see ex-conv).

Besides using predicate transformers as function types, there is a second interaction of Thorium and predicate transformers: Thorium provides a heuristic for joining them. That is, when we infer the precondition of a conditional **if** ϕ **then** G_1 **else** G_2, we first obtain two separate preconditions $R_1(\Phi)$ and $R_2(\Phi)$. Usually, Argon lifts them into the precondition by picking $R(\Phi) \triangleq$ **if** ϕ **then** $R_1(\Phi)$ **else** $R_2(\Phi)$ (see abd-if in Fig. 19.2). As an alternative, Thorium provides a heuristic to join them into a single precondition, $(T \sqcup_\phi T')(\Phi)$, which can be activated on demand.[6] For example, the heuristic would join $(v \triangleleft_v \texttt{num[}it\texttt{]}\, n_1) \sqcup_\phi (v \triangleleft_v \texttt{num[}it\texttt{]}\, n_2)$ into $v \triangleleft_v \texttt{num[}it\texttt{]}\,(\text{if}\ \phi\ \text{then}\ n_1\ \text{else}\ n_2)$, pushing the conditional further into the precondition. Unfortunately, the heuristic is not always successful due to the considerable expressiveness of Thorium types: As in RefinedC, types can contain nesting (via own $\ell\ A$), existential quantification (via $\exists x.\, A\, x$), separating conjunction (via $A * P$), conditionals (via optional $\phi\ A$), and even recursion (via $x\,@\,\mathrm{P}$), making joining hard. If the heuristic fails, we default back to the precondition that Argon would otherwise pick, using $(T \sqcup_\phi T')(\Phi) =$ **if** ϕ **then** $R_1(\Phi)$ **else** $R_2(\Phi)$.

[6] In the implementation, to activate it, one uses the annotation "`[[q::join_if]]`".

20.3 Compositional Specification Inference with Thorium

To illustrate concretely how Thorium enables Quiver to infer specifications of C functions, let us return to the range example from §18. The implementation of the range data type in C is depicted in Fig. 20.3 (alongside annotations for Quiver in blue). One aspect that is interesting about this implementation is that the inference will be *compositional*: the function `mkrange` uses the specification that we will infer for `init`, and `mkrange` also uses the specification of the derived memory allocator `xmalloc` (from §17; discussed below).

The standard library. Before we dive into the details of the range example, let us briefly focus on the auxiliary function `xmalloc` and the standard library functions used to define it. Recall its implementation (from §17):

```c
void *xmalloc(size_t size) {
  void *ptr = malloc(size);
  if (ptr == NULL) abort();
  return ptr;
}
```

To infer specifications for functions like `xmalloc` and `mkrange`, Quiver assumes specifications for various functions from the C standard library. A selection of them is depicted in Fig. 20.4. In contrast to the specifications in Fig. 17.2 from §17, these specifications are stated as predicate transformers using Argon types. For example, when we call `malloc`, we must provide a `size_t`-integer n, and we get back either `NULL` (if the program is out of memory) or a freshly allocated block (pointed to by location ℓ) of size n. The memory in the allocated block is initially arbitrary (any n), and the separation logic resource block $\ell\ n$ keeps track of the size of an allocated block. To express the two cases in a predicate transformer, we use a conjunction, which ensures that clients of `malloc` have to consider *both* cases (*i.e.*, they have to prove the `NULL` case *and* the case where a valid pointer is returned). Conversely, when we call `free`,

```
 1 typedef struct
 2 [[q::refined_by(s : ℤ, e : ℤ)]]
 3 [[q::typedef(range := ∃q. own q ... ∗ block q (sizeof(struct ran)))]]
 4 [[q::constraints(0 ≤ s ≤ e)]]
 5 ran {
 6   [[q::field(num[int] s)]] int low;
 7   [[q::field(num[int] e)]] int high;
 8 } *range;
 9
10 void init(range r, int a, int b) {
11   r->low = a;
12   r->high = b;
13   [[q::assert(^r ◁_v ? @ range)]];
14 }
15
16 range mkrange(int a, int b) {
17   range r = xmalloc(sizeof(struct range));
18   init(r, a, b);
19   return r;
20 }
21
22 [[q::requires(^r ◁_v ? @ range)]]
23 [[q::ensures(^r ◁_v ? @ range)]]
24 int size(range r) {
25   return r->high - r->low;
26 }
```

Figure 20.3: The range data type implemented in C.

we must provide an owned pointer ℓ to an arbitrary memory block of size n.[7] To ensure that the entire ownership is given up, we also have to supply the block $\ell\ n$ predicate to ensure that n is indeed the size of the allocation block at location ℓ. Moreover, when we call `abort`, we terminate the execution. Early termination is considered safe behavior (*i.e.,* it is not a safety violation) and, hence, the postcondition is False (and the return value of type void).

[7] Here, the type value $n\ v$ constrains the value to be v and to be of size n.

From these, Quiver can then derive allocator specifications such as the following specification of `xmalloc`:

$$T_{\texttt{xmalloc}}(v_{\texttt{size}})(\Phi) \triangleq$$
$$\exists n.\ (v_{\texttt{size}} \lhd_v \mathsf{num[size\_t]}\ n) * (\forall w, \ell.\ w \lhd_v \mathsf{own}\ \ell\ (\mathsf{any}\ n) * \mathsf{block}\ \ell\ n \mathrel{-\!\!*} \Phi\ w)$$

Compared to the specification of `malloc`, $T_{\texttt{malloc}}$, it prunes the case where `malloc` returns `NULL`, because in that case `xmalloc` aborts. The inference of this specification needs no sketches, since the resulting specifications follows directly from puzzling together the specifications of `malloc` and `abort`.

The sketches. Let us now turn to the range implementation in Fig. 20.3. We start by discussing the sketches that we provide for this example with "`[[q::...]]`" (in blue). First, in Lines 2-8 we declare the Thorium type (n_s, n_e) @ range (from §20.1). (The annotations follow the structure of RefinedC's type declarations.)

Next, we add Thorium versions of the sketches in §18.3 (in Fig. 18.2). Concretely, for `init`, we assert (with `[[q::assert(^r ◁_v ? @ range)]]`) in Line 13 that the argument value corresponding to `r` (indicated by `^r`) should contain

$$T_{\texttt{malloc}}(v_{\mathsf{n}})(\Phi) \triangleq$$
$$\exists n.\ (v_{\mathsf{n}} \triangleleft_v \texttt{num[size\_t]}\ n) * \begin{pmatrix} \forall w.\ w \triangleleft_v \mathsf{null} \mathrel{-\!\!*} \Phi\, w \\ \land \\ \forall w, \ell.\ w \triangleleft_v \mathsf{own}\ \ell\ (\mathsf{any}\ n) * \mathsf{block}\ \ell\ n \mathrel{-\!\!*} \Phi\, w \end{pmatrix}$$
$$T_{\texttt{free}}(v_{\mathsf{x}})(\Phi) \triangleq$$
$$\exists \ell, v, n.\ (v_{\mathsf{x}} \triangleleft_v \mathsf{own}\ \ell\ (\mathsf{value}\ n\ v) * \mathsf{block}\ \ell\ n) * \forall w.\ w \triangleleft_v \mathsf{void} \mathrel{-\!\!*} \Phi\, w$$
$$T_{\texttt{abort}}()(\Phi) \triangleq \forall w.\ (\mathsf{False} * w \triangleleft_v \mathsf{void}) \mathrel{-\!\!*} \Phi\, w$$

Figure 20.4: A selection of assumed standard library specifications.

some range by the end of the function `init`. We do not specify which bounds the range should have and, instead, leave a hole using a question mark `?`. Moreover, for `size`, we assert that the argument value for `r` should be some range at the beginning and at the end of `size`. We do so in Line 22 and Line 23 with the annotation `[[q::requires(^r` $\triangleleft_v$ `? @range)]]` (for preconditions) and the annotation `[[q::ensures(^r` $\triangleleft_v$ `? @range)]]` (for postconditions).

The inferred specifications. If we now run the implementation of Quiver (see §21) on this combination of sketches and C code, we obtain the following inferred specifications:[8]

$$T_{\texttt{init}}(v_{\mathsf{r}}, v_{\mathsf{a}}, v_{\mathsf{b}})(\Phi) \triangleq$$
$$\exists \ell, n_{\mathsf{a}}, n_{\mathsf{b}}. \begin{pmatrix} v_{\mathsf{r}} \triangleleft_v \mathsf{own}\ \ell\ (\mathsf{struct}[\texttt{struct ran}]\ [\mathsf{any}\ sz_{\texttt{int}}; \mathsf{any}\ sz_{\texttt{int}}]) * \\ v_{\mathsf{a}} \triangleleft_v \texttt{num[int]}\ n_{\mathsf{a}} * v_{\mathsf{b}} \triangleleft_v \texttt{num[int]}\ n_{\mathsf{b}} * 0 \le n_{\mathsf{a}} \le n_{\mathsf{b}} * \mathsf{block}\ \ell\ sz_{\texttt{ran}} \end{pmatrix}$$
$$* \ (\forall w.\ w \triangleleft_v \mathsf{void} * v_{\mathsf{r}} \triangleleft_v (n_{\mathsf{a}}, n_{\mathsf{b}})\ @\, \mathsf{range} \mathrel{-\!\!*} \Phi\, w)$$
$$T_{\texttt{mkrange}}(v_{\mathsf{r}}, v_{\mathsf{a}}, v_{\mathsf{b}})(\Phi) \triangleq$$
$$\exists n_{\mathsf{a}}, n_{\mathsf{b}}.\ (v_{\mathsf{a}} \triangleleft_v \texttt{num[int]}\ n_{\mathsf{a}} * v_{\mathsf{b}} \triangleleft_v \texttt{num[int]}\ n_{\mathsf{b}} * 0 \le n_{\mathsf{a}} \le n_{\mathsf{b}})$$
$$* \ (\forall w.\ w \triangleleft_v (n_{\mathsf{a}}, n_{\mathsf{b}})\ @\, \mathsf{range} \mathrel{-\!\!*} \Phi\, w)$$
$$T_{\texttt{size}}(v_{\mathsf{r}})(\Phi) \triangleq$$
$$\exists n_{\mathsf{e}}, n_{\mathsf{s}}.\ (v_{\mathsf{r}} \triangleleft_v (n_{\mathsf{s}}, n_{\mathsf{e}})\ @\, \mathsf{range})$$
$$* \ (\forall w.\ w \triangleleft_v \texttt{num[int]}\ (n_{\mathsf{e}} - n_{\mathsf{s}}) * v_{\mathsf{r}} \triangleleft_v (n_{\mathsf{s}}, n_{\mathsf{e}})\ @\, \mathsf{range} \mathrel{-\!\!*} \Phi\, w)$$

For `init`, we obtain as the precondition two integers n_{a} and n_{b} such that $0 \le n_{\mathsf{a}} \le n_{\mathsf{b}}$ and a type assignment for the pointer in `r` (which can initially be storing arbitrary contents). In the postcondition, we obtain that `r` now stores a correctly initialized range with bounds n_{a} and n_{b}. For `mkrange`, we obtain a new range as the return value if we provide it with two integers n_{a} and n_{b}. Finally, for `size`, we obtain the difference between the two bounds as the return value if we call it with a range.

One thing to note about these specifications is that—as remarked above—their inference is *compositional*. The specification of `mkrange` is derived from the specification of `init` and, thus, even without sketches, it uses the `range` type. Moreover, since it calls `xmalloc` internally, the inference also uses the derived specification $T_{\texttt{xmalloc}}$ above. (Since the input type of `init` and the output type of `xmalloc` do not match up exactly, Thorium uses the type conversion from §20.2 to match them up.)

[8] For `init`, the implementation infers an equivalent specification that decomposes the struct stored in ℓ.

Chapter 21

Implementation

We have developed a prototype implementation of Quiver in Rocq.[1] More specifically, we have implemented the goal-directed abduction engine Argon $\Delta * [R] \vdash G$ (which embeds the typing rules of Thorium) as an *automated abduction procedure* in Rocq. For a given C function (and possibly a sketch), it (1) infers a specification and, at the same time, (2) proves its correctness.

[1] Spies et al., *Quiver: Guided abductive inference of separation logic specifications in Coq (Rocq development and appendix)*, 2024 [Spi+24b].

We use the Rocq proof assistant as a foundation for Quiver for two main reasons: First, Quiver inherits Rocq's rich logic for expressing complex correctness properties (as evaluated in §22). Second, it allows us to ensure the correctness of the inferred specifications. Concretely, we have proven Quiver's inference foundationally sound against RefinedC's C semantics, Caesium. Caesium provides a detailed formalization of C, modeling many challenging features ranging from bounded integers and pointer arithmetic, over uninitialized memory with poison semantics and address-of operator (also on local variables), to manipulation of the underlying byte-level representation of values.[2] To prove Quiver sound against Caesium, we have used Iris to model Argon (see §19) and Thorium (see §20.1). We have proven all rules sound against this model:

[2] Quiver's version of Caesium forgoes checking alignment of accesses as the resulting constraints would clutter the inferred specifications, and we do not use the integer-pointer-casting semantics introduced by [Lep+22].

Theorem 77.
All Argon and Thorium rules are sound wrt. the Caesium C semantics.

The automated abduction procedure combines the soundness of the individual rules into a foundational proof that the inferred specifications are sound. In our examples, we assume specifications for common operations from the C standard library (*e.g.*, `malloc`, `free`, and `abort` in Fig. 20.4). Thus,

Corollary 78. *Assuming the standard library function satisfy their specifications, the specifications inferred by Quiver are sound wrt. the Caesium C semantics.*

Finally, Quiver comes with a frontend that automatically translates annotated C code into (1) corresponding Caesium code, (2) type declarations in Thorium, and (3) calls to the abduction procedure for Argon. The abduction procedure is implemented using Rocq's Ltac tactic language[3] and typeclass mechanism.[4]

[3] Delahaye, "A tactic language for the system Coq", 2000 [Del00].

[4] Sozeau and Oury, "First-class type classes", 2008 [SO08].

Chapter 22

Evaluation

To evaluate Quiver, we have applied it to several interesting case studies, listed in Fig. 22.3. We split our evaluation into two parts: First, we take a closer look at a specific case study, *a vector implementation*, to give an impression of the kind of specifications that Quiver can infer (in §22.1). Then, we discuss the aggregate results of evaluating Quiver on these case studies (in §22.2).

22.1 The Vector Case Study

Inspired by C++ and Rust, a *vector* is a dynamically-sized array that tracks its length. An excerpt of the vector implementation is depicted in Fig. 22.2. In this implementation, vectors are pointers to a struct with two fields: the `data`-field storing the contents of the vector in a dynamically allocated array of integers and the `len`-field tracking the length of the vector. For vectors in Quiver, we define the following Thorium-data type (in Lines 1-6):

$$\begin{aligned} xs \mathbin{@} \mathsf{vec\_t} &\equiv_{\mathsf{ty}} \exists \ell.\, \mathsf{own}\, \ell\, (\mathsf{struct}[\mathtt{vector}]\, [A_{\mathtt{data}}; A_{\mathtt{len}}]) * \mathsf{block}\, \ell\, sz_{\mathtt{vec}} \\ \textit{where}\ A_{\mathtt{data}} &\triangleq \exists r.\, \mathsf{own}\, r\, (\mathsf{array}[\mathsf{num}[\mathtt{int}]]\, xs) * \mathsf{block}\, r\, (sz_{\mathtt{int}} \cdot |xs|), \\ A_{\mathtt{len}} &\triangleq \mathsf{num}[\mathtt{int}]\, (|xs|) \end{aligned}$$

That is, for a mathematical list of integers xs, a value of type $xs \mathbin{@} \mathsf{vec\_t}$ is an owned pointer ℓ to a `vector`-struct. It stores in its `data`-field an owned pointer r to an array of integers xs and in its `len`-field the length of xs as an integer. It tracks the memory block resources of ℓ of size $sz_{\mathtt{vec}}$ and r of size $sz_{\mathtt{int}} \cdot |xs|$ where $sz_{\mathtt{vec}} \triangleq$ **`sizeof`**`(`**`struct`**` vector)` and $sz_{\mathtt{int}} \triangleq$ **`sizeof`**`(int)`.

We focus on five vector operations: The operation `mkvec` creates a new vector of length n initialized with zeros. The operation `get_unsafe` retrieves an element from the vector, and `set_unsafe` updates an element in the vector. The operation `get_checked`, unlike `get_unsafe`, additionally performs a check that the index is in bounds. Lastly, the operation `grow` extends a vector by allocating a new underlying buffer. Concretely, `grow` allocates a new content array `buf` of larger size (Line 48), copies the contents of the old array over (Line 49), frees the old array (Line 51), sets all uninitialized memory to zero (Line 52), and returns the new length (Line 54).

Sketches and inferred specifications. For each operation, the *specification sketches* are annotated with "`[[q::...]]`" in Fig. 22.2 (as in §20.3). The *inferred specifications* are depicted in Fig. 22.1. For `mkvec`, Quiver infers that the size n must be a non-negative `int`-integer and that the return value is a vec_t-vector

$$
\begin{aligned}
&T_{\mathtt{mkvec}}(v_{\mathtt{n}})(\Phi) \triangleq \exists n.\ (v_{\mathtt{n}} \lhd_v \mathtt{num[int]}\ n * n \geq 0) * (\forall w.\ w \lhd_v 0^n @ \mathrm{vec\_t} \mathrel{-\!\!*} \Phi\, w)\\
&T_{\mathtt{get\_unsafe}}(v_{\mathtt{vec}}, v_{\mathtt{i}}, v_{\mathtt{x}})(\Phi) \triangleq\\
&\quad \exists xs, i, \ell.\ (0 \leq i < |xs| * v_{\mathtt{vec}} \lhd_v xs @ \mathrm{vec\_t} * v_{\mathtt{i}} \lhd_v \mathtt{num[int]}\ i * v_{\mathtt{x}} \lhd_v \mathsf{own}\ \ell\ (\mathsf{any}\ sz_{\mathtt{int}})) *\\
&\quad \forall w.\ \ell \lhd_l \mathtt{num[int]}\ (xs[i]) * v_{\mathtt{vec}} \lhd_v xs @ \mathrm{vec\_t} * w \lhd_v \mathsf{void} \mathrel{-\!\!*} \Phi\, w\\
&T_{\mathtt{set\_unsafe}}(v_{\mathtt{vec}}, v_{\mathtt{i}}, v_{\mathtt{x}})(\Phi) \triangleq\\
&\quad \exists xs, \ell, i, n.\ (0 \leq i < |xs| * v_{\mathtt{vec}} \lhd_v xs @ \mathrm{vec\_t} * v_{\mathtt{i}} \lhd_v \mathtt{num[int]}\ i * v_{\mathtt{x}} \lhd_v \mathsf{own}\ \ell\ (\mathtt{num[int]}\ n)) *\\
&\quad \forall w.\ \ell \lhd_l \mathtt{num[int]}\ n * v_{\mathtt{vec}} \lhd_v (xs[i \mapsto n]) @ \mathrm{vec\_t} * w \lhd_v \mathsf{void} \mathrel{-\!\!*} \Phi\, w\\
&T_{\mathtt{get\_checked}}(v_{\mathtt{vec}}, v_{\mathtt{i}})(\Phi) \triangleq\\
&\quad \exists xs, i.\ (v_{\mathtt{vec}} \lhd_v xs @ \mathrm{vec\_t} * v_{\mathtt{i}} \lhd_v \mathtt{num[int]}\ i) *\\
&\quad \mathbf{if}\ (i < 0 \lor i \geq |xs|)\ \mathbf{then}\ \forall w.\ v_{\mathtt{vec}} \lhd_v xs @ \mathrm{vec\_t} * w \lhd_v \mathtt{num[int]}\ (-1) \mathrel{-\!\!*} \Phi\, w\\
&\quad \mathbf{else}\ \forall w.\ v_{\mathtt{vec}} \lhd_v xs @ \mathrm{vec\_t} * w \lhd_v \mathtt{num[int]}\ (xs[i]) \mathrel{-\!\!*} \Phi\, w\\
&T_{\mathtt{grow}}(v_{\mathtt{vec}}, v_{\mathtt{new}})(\Phi) \triangleq \exists xs, n.\ (v_{\mathtt{vec}} \lhd_v xs @ \mathrm{vec\_t} * v_{\mathtt{new}} \lhd_v \mathtt{num[int]}\ n) *\\
&\qquad \mathbf{if}\ n \leq |xs|\ \mathbf{then}\ \forall w.\ v_{\mathtt{vec}} \lhd_v xs @ \mathrm{vec\_t} * w \lhd_v \mathtt{num[int]}\ |xs| \mathrel{-\!\!*} \Phi\, w\\
&\qquad \mathbf{else}\ \forall w.\ v_{\mathtt{vec}} \lhd_v (xs \mathbin{+\!\!+} 0^{n-|xs|}) @ \mathrm{vec\_t} * w \lhd_v \mathtt{num[int]}\ n \mathrel{-\!\!*} \Phi\, w
\end{aligned}
$$

Figure 22.1: Inferred vector specifications, preconditions in orange and postconditions in violet.

filled with 0^n, a list of n zeros. For `get_unsafe` and `set_unsafe`, Quiver infers constraints on the index `i`, the additional pointer argument `x`, and how the vector changes in `set_unsafe`/the pointer changes in `get_unsafe`. For `get_checked`, Quiver infers a specification that distinguishes the success and error case, using the ability of predicate transformer specifications to contain conditionals.

For `grow`, Quiver infers a conditional specification: if $n \leq |xs|$, the vector is unchanged and $|xs|$ is returned; otherwise, the vector grows by $n - |xs|$ zeros and n is returned as the new length. To arrive at this specification, Quiver (1) infers the type of the unspecified argument $v_{\mathtt{new}}$, (2) resolves the quantifier alternations that arise from each memory operation (a $\exists\forall$ for each operation), (3) instantiates the sketches (including $xs \mathbin{+\!\!+} 0^{n-|xs|} @ \mathrm{vec\_t}$ for the second case), (4) proves that $|xs \mathbin{+\!\!+} 0^{n-|xs|}| = n$ when $n > |xs|$, and (5) prunes the branch returning `0` using the fact that $xs @ \mathrm{vec\_t}$ is never `NULL`.

Abductive deductive verification. The vector case study illustrates concisely the benefits of abductive deductive verification. On the one hand, we are doing expressive separation logic verification. For example, (a) vectors track their contents as a mathematical list of integers, (b) vectors maintain the invariant that the length of the list is stored in the field `len`, (c) dynamically allocated memory can be of variable length, which is tracked via a predicate block, (d) pointer arithmetic is used to compute fields of structs and members of arrays, and (e) pointer-level operations (*e.g.*, `memset` and `memcpy`) are used to manipulate high-level data types (*e.g.*, arrays). On the other hand, we can significantly benefit from inference for the verification. In particular, we only need to provide the key bit of information—that a certain value is a vector—and can use inference to complete the rest. In fact, for `get_unsafe`, `set_unsafe`, `get_checked`, and `grow`, we provide *exactly the same sketches* and, yet, the resulting specifications are quite different.

```
[[q::refined_by(xs : list Z)]]
[[q::typedef(vec_t := ∃p. own p ... ∗ block p (sizeof(struct vector)))]]
typedef struct vector {
  [[q::field(∃q. own q (array[num[int]] xs) ∗ block q (sizeof(int) · |xs|))]] int *data;
  [[q::field(num[int] (|xs|))]] int len;
} *vec_t;

vec_t mkvec(int n) {
  size_t s = sizeof(int)*(size_t)n;
  vec_t vec = xmalloc(sizeof(*vec));
  vec->data = xzalloc(s);
  vec->len = n;
  [[q::type(? @ vec_t)]] return vec;
}


[[q::requires(^vec ◁ᵥ ? @ vec_t)]]
[[q::ensures(^vec ◁ᵥ ? @ vec_t)]]
void get_unsafe(vec_t vec, int i, int *x) {
  *x = vec->data[i];
}

[[q::requires(^vec ◁ᵥ ? @ vec_t)]]
[[q::ensures(^vec ◁ᵥ ? @ vec_t)]]
void set_unsafe(vec_t vec, int i, int *x) {
  vec->data[i] = *x;
}

[[q::requires(^vec ◁ᵥ ? @ vec_t)]]
[[q::ensures(^vec ◁ᵥ ? @ vec_t)]]
int get_checked(vec_t vec, int i){
  assert (vec->len >= 0);
  if (i < 0 || i >= vec->len) {
    return -1;
  }
  return vec->data[i];
}

[[q::requires(^vec ◁ᵥ ? @ vec_t)]]
[[q::ensures(^vec ◁ᵥ ? @ vec_t)]]
int grow(vec_t vec, int new_size) {
  if (vec == NULL) {
    return 0;
  }
  if (new_size <= vec->len) {
    return vec->len;
  }
  int *buf = xmalloc(sizeof(int) * new_size);
  memcpy(buf, vec->data, sizeof(int) * vec->len);
  free(vec->data);
  vec->data = buf;
  memset(&(vec->data[vec->len]), 0, sizeof(int)*(new_size-vec->len));
  vec->len = new_size;
  return vec->len;
}
```

Figure 22.2: The implementation of the vector. Quiver annotations in blue.

Implementation			Specification					Execution		
Name	Functions	Code	Type	Specs	Sketch	Annot	Rocq	∃	ϕ	Time
Allocators	`xmalloc`, `xzalloc`, `xrealloc`, …(+3)	41	mem	55	0	0	0	77	44/31	0:58
Linked list	`init`, `is_empty`, `push`, `pop`,	37	mem	27	0	0	0	16	11/2	0:27
	`reverse` (only functional)		func	25	10	11/5/0	0	39	21/8	0:46
Vector	`mkvec`, `get_unsafe`, `grow`,	59	mem	147	0	0	0	106	62/27	2:48
	`get_checked`, `vec_free`, …(+3)		func	75	14	11/0/0	0	117	164/43	2:40
Bipbuffer	`new`, `free`, `request`, `push`, …(+11)	105	len	210	21	10/0/2	0	378	476/160	8:51
OpenSSL Buffer	`BUF_MEM_new`, `BUF_MEM_free`,	107	mem	249	0	0	0	285	302/113	14:14
	`BUF_MEM_grow`, …(+3)		len	94	9	14/0/4	0	310	431/90	9:50
Binary search	`bin_search`	14	func	11	5	7/8/0	49	18	49/3	0:41
Hashmap	`init`, `probe`, `realloc`, …(+5)	101	func	79	72	19/18/7	506	221	375/123	7:56

Figure 22.3: Evaluation of Quiver. Code: lines of C code as formatted by clang-format; Type: type of inferred specification (*i.e.,* mem: memory safety, len: length and type invariants, func: functional); Specs: size of the inferred specification; Sketch: size of the function sketches; Annot: size of type definitions/size of loop invariants/additional inference instructions; Rocq: lines of pure Rocq definitions and lemmas; "∃": number of instantiated existential quantifiers; "ϕ": number of proven/simplified side conditions; Time: execution time in minutes:seconds.

22.2 Aggregate Evaluation

Let us now turn to the aggregate evaluation of Quiver on several case studies. We evaluate the prototype implementation of Quiver on three axes: (1) the expressivity (compared to bi-abduction), (2) the specification overhead (compared to RefinedC), and (3) the merit of the inferred specifications. We do so using the case studies in Fig. 22.3. For each case study, a more detailed discussion can be found in the appendix of the Quiver paper and all implementations and inferred specifications can be found in the Rocq development [Spi+24b]. The Allocators case study considers common wrappers around standard library functions for memory allocation (*e.g.,* `xmalloc` and `xzalloc`). The Linked List case study considers a singly linked-list implementation with pointer elements, and the Vector case study extends the vector from §22.1. The OpenSSL Buffer and Bipbuffer case studies consider open-source buffer implementations from OpenSSL [Ope25] and memcached [mem25]. The Binary Search case study considers binary search on sorted integer lists, and the Hashmap case study considers a hashmap with linear probing. For each case study, we measure the execution time on a single core of an Apple M1 Pro processor (Time).

Expressivity (*vs.* Bi-abduction). To understand the degree of expressivity that Quiver supports, we consider several types of specifications (Type in Fig. 22.3), increasing in complexity: We infer *memory safety specifications* (mem) for several examples—including the Allocators, whose inferred specifications (*e.g.,* `xmalloc` and `xzalloc`) we use in other case studies. We infer *length specifications* (len) for the open-source buffers, which track the length of the buffer and data type invariants about its fields. We infer *functional specifications* (func) for the Linked List and the Vector, which track their contents as mathematical lists. And, to test the boundaries of Quiver, we consider a binary search implementation and a Hashmap, a version of the most complex functional correctness case study of Sammler et al.[1] specialized to integer values.[2]

The case studies demonstrate that Quiver, embedded into Rocq, supports expressive separation logic reasoning over a variety of mathematical domains (*e.g.,* integers, lists, maps, and custom inductive types). For example, Quiver figures out that (a) if $n <$ `0x5ffffffc`, then $(n+3)/3 \cdot 4$ will not overflow the `size_t` type (OpenSSL Buffer) and (b) `grow` results in the vector $xs \mathbin{+\!\!+} 0^{n-|xs|}$,

[1] Sammler et al., "RefinedC: Automating the foundational verification of C code with refined ownership types", 2021 [Sam+21].

[2] Differences to the implementation of Sammler et al. [Sam+21] are discussed in the appendix of the Quiver paper [Spi+24b].

which extends the original list xs with $n - |xs|$ zeros (Vector). Moreover, provided with loop invariants and additional Rocq lemmas and definitions, Quiver does significant functional correctness reasoning for the Binary Search and Hashmap. The expressivity of Quiver goes considerably beyond the original bi-abduction inference[3] and also the bi-abduction implemented in Infer [Inf24].[4] In exchange, it requires more input from the user, in particular for more expressive specifications.

[3] Calcagno et al., "Compositional shape analysis by means of bi-abduction", 2009 [Cal+09]; Calcagno et al., "Compositional shape analysis by means of bi-abduction", 2011 [Cal+11].

[4] For example, the bi-abduction in Infer does not do integer reasoning such as (a) if n < `0x5ffffffc`, then $(n+3)/3 \cdot 4 \leq$ `0xffffffffffffffff` from the OpenSSL buffer or (b) after the loop `int k = 0; while (k < 10) k++` the counter k is 10. Quiver automatically proves the former without any guidance, and infers the latter when guided with the loop invariant $k \leq 10$.

Specification overhead (*vs.* RefinedC). To understand how much specification Quiver infers, we compare the size of the inferred specifications (Specs) with the size of our sketches (Sketch) and other annotations (Annot). We measure the size of specifications and sketches by counting the number of quantifiers, conditionals, conjunctions, type assignments, and other individual pre- and postconditions (*e.g.,* the size of $T_{\texttt{mkvec}}$ would be 5). We separately count other annotations such as type definitions, loop invariants, and inference instructions. A handcrafted specification—as it would be provided in RefinedC—could in some cases reduce the size (*e.g.,* by joining the branches in $T_{\texttt{grow}}$), but nevertheless comparing sketches and specs gives an idea how much Quiver infers. Concretely, for the "memory" case studies, we provide no sketches—the specifications are completely inferred. By design, they are low-level (*e.g.,* see $T_{\texttt{xmalloc}}$ in §20.3) and can be verbose. For all other case studies, we provide sketches. They are typically significantly smaller than the resulting specification and often contain ?-holes (*e.g.,* all 14 Vector sketches boil down to ? @ vec_t). In RefinedC, by contrast, specifications must be provided in full.

Among our case studies, there are two outliers: the Binary Search and the Hashmap. This is no surprise, since both require nuanced, ad-hoc functional correctness reasoning with additional pure Rocq definitions and lemmas (Rocq). For them, the specification overhead is overshadowed by the additional proof overhead. Nevertheless, even for those two, Quiver does interesting inference: it completes the return type of Hashmap `init`, and it derives the postcondition of the Binary Search from a loop invariant.

Merit of the specifications. The specifications that Quiver infers provide four key benefits: First, they are an additional form of *documentation.* Quiver outputs a pretty-printed version of the inferred predicate transformer, which can be read by humans. For example, in the Vector, Quiver adds the constraints on the vector size in the specification of `mkvec`.

Second, the inferred specifications provide *assurances* about the code. That is, due to soundness (Corollary 78), the inferred specifications cannot "hide" any preconditions that are undocumented in the code. For example, in the Bipbuffer, Quiver discovers a fact about the implementation that is easy to miss in the code: the implementation uses mismatched integer types (*e.g.,* the size field of the buffer uses `unsigned long int`, but the corresponding accessor function returns `int`), resulting in an additional precondition in the generated specifications.

Third, the inferred specifications are *compositional* (see also §20.3). We inherit compositionality from working in separation logic. In particular, in many of the case studies, we infer specifications of auxiliary functions, which are then reused in the inference of others (*e.g.,* BipBuffer, OpenSSL Buffer, and

Hashmap); and we use the inferred Allocator specifications in other case studies (*e.g.,* in the Vector, List, Hashmap).

Fourth, the inferred specifications *abstract over the implementation.* By insisting on a single ∃∀-alternation, Quiver ensures that the inferred specification condenses the implementation into preconditions and postconditions. In doing so, it takes care of the intricacies of the C implementation and intermediate proof obligations. To gain some insight into how much work goes into this summarization, we count the number of instantiated existentials (∃) and proven/simplified side conditions (ϕ).

Real-world code. Finally, our case studies test whether Quiver can handle the complexities of real-world code. We have applied Quiver to two buffer implementations taken from popular open source libraries, OpenSSL[5] and memcached.[6] For the OpenSSL buffer, we track the length and capacity of the buffer and enforce an invariant that the buffer capacity is always larger than the contents. For the memcached buffer, a *bipartite buffer*, we track the length and the relationship between the fields that track the segments of the buffer.

[5] OpenSSL, *OpenSSL*, 2025 [Ope25].

[6] memcached, *memcached*, 2025 [mem25].

Chapter 23
Related Work

In the literature on separation logic verification, there is a wide gap between approaches for (a) automatically inferring specifications *vs.* (b) verifying functional correctness in rich separation logics. In the first camp, there are approaches such as bi-abduction,[1] which fix a particular fragment of separation logic and then carefully design automation to infer specifications in it. This line of work started out with shape specifications (*i.e.,* linked list segments and points-to assertions) and, over the years, edged closer toward functional properties by extending the base domain to include constraints on integers, arrays, or bags. In the second camp, there are approaches such as RefinedC,[2] which are designed for proving full functional correctness in rich separation logics, as supported by the verification frameworks in which they are embedded (*e.g.,* Rocq and Iris). This line of work, over the years, developed increasingly strong proof automation but left specification inference largely untouched.

[1] Calcagno et al., "Compositional shape analysis by means of bi-abduction", 2009 [Cal+09]; Calcagno et al., "Compositional shape analysis by means of bi-abduction", 2011 [Cal+11].

[2] Sammler et al., "RefinedC: Automating the foundational verification of C code with refined ownership types", 2021 [Sam+21].

Quiver sits right in between these two camps, supporting a wide range in between automated and expressive specifications (see §22.2). Typically, Quiver requires more specification guidance from users than a fully automatic inference (increasing with expressiveness of the specification), but significantly less than traditional, deductive approaches for rich separation logics. In exchange, it does not fix any particular mathematical domain and, instead, is implemented in a general-purpose proof assistant—producing certifiably correct specifications. We first compare closely with work in both camps, and then branch out to other related work.

Inferring ownership specifications in separation logic. In their seminal work, Calcagno et al.[3] introduced *bi-abduction* as a technique for compositional shape analysis in separation logic. Over the years, several extensions to its original domain (*i.e.,* points-tos and list segments) have been proposed, including pure constraints over booleans, integers, and bags;[4] ordering constraints;[5] low-level data representations;[6] second-order predicates;[7] and arrays.[8] The key contribution of each of these extensions is to automate the inference over their respective domain.

[3] Calcagno et al., "Compositional shape analysis by means of bi-abduction", 2009 [Cal+09]; Calcagno et al., "Compositional shape analysis by means of bi-abduction", 2011 [Cal+11].

[4] Trinh et al., "Bi-abduction with pure properties for specification inference", 2013 [Tri+13]; Qin et al., "Automatically refining partial specifications for program verification", 2011 [Qin+11]; He et al., "Automated specification discovery via user-defined predicates", 2013 [He+13].

[5] Curry, Le, and Qin, "Bi-abductive inference for shape and ordering properties", 2019 [CLQ19].

[6] Holík et al., "Low-level bi-abduction", 2022 [Hol+22].

[7] Le et al., "Shape analysis via second-order bi-abduction", 2014 [Le+14].

[8] Brotherston, Gorogiannis, and Kanovich, "Biabduction (and related problems) in array separation logic", 2017 [BGK17].

In contrast, Quiver's specification inference is fundamentally different. By using abductive deductive verification, Quiver is less automated but, in exchange, handles a much richer separation logic by building on existing approaches for deductive proof automation. For example, the vector example (§22.1)—combining low-level pointer operations, arrays, and integer arithmetic—goes beyond all of the above extensions, especially considering the detailed C semantics it is verified against.

Outside of the context of bi-abduction, Dohrau et al.[9] use a static analysis to infer access permissions for array-manipulating programs, and Ferrara and Müller[10] show how to automatically infer access permissions using abstract interpretation. They handle different permission models and loop invariant inference but do not consider functional correctness properties.

Functional correctness verification using separation logic. There is a wide range of approaches for verifying functional correctness based on separation logic,[11] most of which do not infer specifications. We compare to the most closely related work and approaches with some form of specification inference.

A key inspiration for Quiver is RefinedC,[12] which provides automated and foundational verification of C code. Its approach of using a type system embedded in separation logic served as a direct inspiration for Quiver. However, RefinedC does not infer specifications and, hence, relies on *user-provided, complete specifications.* To tackle specification inference, we introduced the abductive deductive verification approach (§18), implemented a proof engine for abduction—Argon—from scratch (§19), and designed a type system—Thorium—that integrates seamlessly with abduction (§20).

For VeriFast,[13] a separation logic-based functional correctness verifier for C and Java, Vogels et al.[14] implement a bi-abduction-based shape analysis. Unlike Quiver, it does not infer functional correctness specifications and only infers a postcondition from a user-provided precondition. Separately, Automated VeriFast[15] leverages errors reported by VeriFast to extend user-written specs with additional pre- and postconditions. Automated VeriFast has only been demonstrated on predicates tracking the length of singly-linked lists.

Dohrau[16] presents a learning-based permission inference for the Viper automated verifier [MSS17]. Their approach can automatically infer loop invariants and predicate definitions, but only considers permissions, not functional correctness properties.

Diaframe. Diaframe[17] provides proof automation for Iris based, in part, on bi-abduction. Unlike Quiver, Diaframe is not concerned with specification inference and instead focuses on proof automation for Iris (*e.g.,* for verifying fine-grained concurrent data structures with Iris invariants). During its automated proof search, Diaframe uses bi-abduction locally at certain points in the search to determine which hint (*i.e.,* which proof rule) to apply next.

Liquid types. Liquid types[18] provide a refinement type-based approach for lightweight verification. Liquid types focus on the inference of pure refinements, not separation logic ownership, and often consider more shape-like properties than Quiver. For example, Lehmann et al.[19] describe a vector similar to vec_t from §22.1, but only track the length in the refinements, not its precise contents. In exchange, liquid types are more automated: they infer refinements and, additionally, loop invariants automatically.

Specification inference for other logics. Outside of the context of separation logic, a separate body of research[20] considers inferring specifications for programs that do not involve pointer manipulation or a heap. This restriction sidesteps the main challenges this part of the dissertation focuses on (see, *e.g.,*

[9] Dohrau et al., "Permission inference for array programs", 2018 [Doh+18].

[10] Ferrara and Müller, "Automatic inference of access permissions", 2012 [FM12].

[11] Jacobs et al., "VeriFast: A powerful, sound, predictable, fast verifier for C and Java", 2011 [Jac+11]; Appel, "Verified Software Toolchain", 2012 [App12]; Cao et al., "VST-Floyd: A separation logic tool to verify correctness of C programs", 2018 [Cao+18]; Müller, Schwerhoff, and Summers, "Viper: A verification infrastructure for permission-based reasoning", 2017 [MSS17]; Sammler et al., "RefinedC: Automating the foundational verification of C code with refined ownership types", 2021 [Sam+21]; Pulte et al., "CN: Verifying systems C code with separation-logic refinement types", 2023 [Pul+23].

[12] Sammler et al., "RefinedC: Automating the foundational verification of C code with refined ownership types", 2021 [Sam+21].

[13] Jacobs et al., "VeriFast: A powerful, sound, predictable, fast verifier for C and Java", 2011 [Jac+11].

[14] Vogels et al., "Annotation inference for separation logic based verifiers", 2011 [Vog+11].

[15] Mohsen and Jacobs, "One step towards automatic inference of formal specifications using automated VeriFast", 2016 [MJ16].

[16] Dohrau, "Automatic inference of permission specifications", 2022 [Doh22].

[17] Mulder, Krebbers, and Geuvers, "Diaframe: Automated verification of fine-grained concurrent programs in Iris", 2022 [MKG22].

[18] Rondon, Kawaguchi, and Jhala, "Liquid types", 2008 [RKJ08]; Rondon, Kawaguchi, and Jhala, "Low-level liquid types", 2010 [RKJ10]; Vazou et al., "Refinement types for Haskell", 2014 [Vaz+14]; Lehmann et al., "Flux: Liquid types for Rust", 2023 [Leh+23].

[19] Lehmann et al., "Flux: Liquid types for Rust", 2023 [Leh+23].

[20] Seghir and Kroening, "Counterexample-guided precondition inference", 2013 [SK13]; Albarghouthi, Dillig, and Gurfinkel, "Maximal specification synthesis", 2016 [ADG16]; S et al., "Specification synthesis with constrained Horn clauses", 2021 [S+21].

the vector in §22.1). In exchange, they typically obtain *exact* (*i.e.,* sufficient and necessary) preconditions, whereas Quiver infers sufficient preconditions.

Characteristic formulae. A characteristic formula[21] is a direct translation of a program into a separation logic formula. Characteristic formulae are not intended as specifications, but as an intermediate representation used during verification. In particular, they still contain all intermediate proof obligations required to verify a function. In contrast, Quiver infers specifications that *summarize* the behavior of a function in terms of pre- and postconditions (*i.e.,* in ∃∀-form; see §18.2) by resolving quantifier dependencies and solving side conditions.

[21] Charguéraud, "Program verification through characteristic formulae", 2010 [Cha10]; Charguéraud, "Characteristic formulae for the verification of imperative programs", 2011 [Cha11].

Part V

Daenerys

Chapter 24

Introduction

As we have seen in the previous chapters, the forte of *separation logic* (SL)—and the reason why frameworks like Iris build on it—is that it enables a modular style of reasoning for complex imperative programs via the concept of *ownership*. The canonical example is of course the *points-to assertion*, $\ell \mapsto v$, which asserts not only that the pointer ℓ currently points to the value v but also that the function being verified "owns" the memory location ℓ.

However, separation logic is not the only formal foundation for reasoning about ownership. Another closely related, yet decidedly different, foundation is that of *implicit dynamic frames (IDF)*.[1] Deployed most extensively as the foundation of the Viper verification framework,[2] IDF is similar to SL in that assertions can talk both about the state of the program and ownership of that state. But unlike in SL, IDF assertions do not have to talk about these two things simultaneously—rather, they are disentangled. For example, instead of the single SL assertion $\ell \mapsto v$, which combines ownership of ℓ with the fact that ℓ points to v, IDF has two kinds of assertions: (1) an *access assertion* $\mathrm{acc}(\ell)$, which conveys ownership of the location ℓ (entailing the right to access and update ℓ) but does not say what ℓ points to, and (2) *heap-dependent expression assertions (HDEAs)* such as $!\,\ell =_{\mathrm{IDF}} v$, which says that ℓ points to v but does not assert ownership of ℓ. (HDEAs, as we will see below, are not limited to assertions about a single memory location ℓ; they may also, for example, include assertions about the results of function calls.)

Separation logic's $\ell \mapsto v$ can be expressed in IDF as $\mathrm{acc}(\ell) * !\,\ell =_{\mathrm{IDF}} v$. Conversely, IDF's access assertion $\mathrm{acc}(\ell)$ can be expressed in SL as $\mathrm{acc}(\ell) \triangleq \exists v.\, \ell \mapsto v$. But how can one encode HDEAs in SL? It is not so simple. For example, take Iris: suppose that $\mathrm{acc}(\ell)$ and $!\,\ell =_{\mathrm{IDF}} v$ could be expressed as separate Iris assertions (conjoined by the separating conjunction $P * Q$). Using $\mathrm{acc}(\ell)$, we could update ℓ to w, thereby obtaining $!\,\ell =_{\mathrm{IDF}} w$. But by the central *frame rule* of SL (see hoare-frame in §2.2), we would be able to frame the assertion $!\,\ell =_{\mathrm{IDF}} v$ around the update, thus leading to a contradiction!

Nevertheless, in this part of the dissertation, we will show that in fact HDEAs (and more generally IDF-style reasoning) *can* be soundly incorporated into SL[3]—and Iris in particular—and that they constitute a demonstrably useful extension to the Iris toolbox. To do so, we will need to revisit the foundations of Iris once more in order to support *unstable resources*, a new type of resources that do not unconditionally enjoy the frame rule. But before we get into more details about our contributions (§24.2), let us begin by reviewing why we want to bring HDEAs to Iris in the first place (§24.1).

[1] Smans, Jacobs, and Piessens, "Implicit Dynamic Frames: Combining dynamic frames and separation logic", 2009 [SJP09].

[2] Blom et al., "The VerCors tool set: Verification of parallel and concurrent software", 2017 [Blo+17]; Müller, Schwerhoff, and Summers, "Viper: A verification infrastructure for permission-based reasoning", 2017 [MSS17]; Astrauskas et al., "Leveraging Rust types for modular specification and verification", 2019 [Ast+19]; Eilers and Müller, "Nagini: A static verifier for Python", 2018 [EM18]; Wolf et al., "Gobra: Modular specification and verification of Go programs", 2021 [Wol+21].

[3] Parkinson and Summers [PS11] show how to encode SL in IDF with $\ell \mapsto v \triangleq \mathrm{acc}(\ell) \wedge !\,\ell =_{\mathrm{IDF}} v$. In the following, we consider the *reverse direction*: we want to bring IDF to SL, and we do so for one of the most expressive separation logics out there, Iris.

$$\{\mathsf{True}\}\ \mathtt{produce\_buffer}()\ \{v.\, \exists b, \vec{u}.\, v = b * b \mapsto \vec{u}\}$$

$$\{b \mapsto \vec{u}\}\ \mathtt{read\_only\_client}(b)\ \{\_.\, b \mapsto \vec{u}\}$$

$$\{b \mapsto \vec{u}\}\ \mathtt{checksum}(b)\ \{v.\, v = cs(\vec{u}) * b \mapsto \vec{u}\}$$

Figure 24.1: Separation logic specifications of the operations in the motivating example.

24.1 Heap-Dependent Expression Assertions

To illustrate the utility of HDEAs, let us consider a concrete example:

```
1 let buf = produce_buffer() in     ■ {buf ↦ u⃗}
2 let chk1 = checksum(buf)   in     ■ {buf ↦ u⃗ ∗ chk1 = cs(u⃗)}
3 read_only_client(buf);            ■ {buf ↦ u⃗ ∗ chk1 = cs(u⃗)}
4 let chk2 = checksum(buf)   in     ■ {buf ↦ u⃗ ∗ chk1 = cs(u⃗) ∗ chk2 = cs(u⃗)}
5 assert(chk1 == chk2)
```

In this example, we allocate a buffer of 64-bit unsigned integers on the heap using the function `produce_buffer` (Line 1) and then pass it to a client (Line 3). The client is only allowed to read from the buffer. Thus, we can compute a checksum of the buffer before (Line 2) and after the read-only client (Line 4) and assert that they are the same (Line 5). (The exact algorithm by which the checksum is computed does not matter, as long as it is deterministic.)

Let us first sketch how one could show that the assert always succeeds *in separation logic*. We have annotated intermediate proof states (in ■ orange), and we use the specifications depicted in Fig. 24.1. First, we allocate the buffer and obtain the ownership of `buf` storing a sequence of 64-bit unsigned integers $\vec{u}$ (Line 1).[4] Then, we use it to compute `checksum(buf)`. The result is $cs(\vec{u})$, where cs is a mathematical version of `checksum` operating on the contents $\vec{u}$ (in Line 2). Next, we pass the buffer `buf` to the read-only client, which does not change its contents $\vec{u}$. Thus, when we recompute the checksum (Line 4), it is still $cs(\vec{u})$ and the assert—comparing $cs(\vec{u})$ and $cs(\vec{u})$—succeeds (Line 5).

[4] We gloss over the difference between a points-to assertion for a buffer b storing a sequence of values $\vec{v}$ (written $b \mapsto \vec{v}$) and a points-to assertion for an individual location ℓ storing a single value v (written $\ell \mapsto v$) in the following.

This verification works. But it is more laborious than it appears at first glance, because to complete it, we must additionally verify the Hoare triples in Fig. 24.1. This involves a non-trivial amount of work: (1) *reformulating the implementation* of `checksum` as a mathematical function cs and (2) *proving full functional correctness* of `checksum` by showing that it implements cs. If `checksum` is small and simple, proving its functional correctness is not a big burden—but if it is a nontrivial recursive function, verifying it becomes tedious quickly. Moreover, functional correctness of `checksum` is a much stronger property than we actually need! The assert succeeds so long as (1) the result of `checksum` depends only on the buffer, (2) `checksum` does not modify the buffer, and (3) the read-only client does not modify the buffer. (Yet, we cannot weaken the specification to say that `checksum` returns just *some integer*, because it must be the same one in Line 2 and Line 4.)

HDEAs offer a simpler way to handle such examples. They enrich the assertion language with the ability to describe the current result of deterministic, read-only program expressions. In particular, they permit the logic-level assertion `checksum(buf)` $=_{\mathrm{IDF}}$ `chk1` (note the use of `checksum` instead of cs), which says that the current value of `checksum(buf)` is `chk1`. Since the client does not modify `buf`, one can then frame `checksum(buf)` $=_{\mathrm{IDF}}$ `chk1` around it using an ownership argument, which we will spell out in §25.1. Thus, when we reach Line 5, we know at the same time `checksum(buf)` $=_{\mathrm{IDF}}$ `chk1` and also `checksum(buf)` $=_{\mathrm{IDF}}$ `chk2`, from which we can deduce `chk1` = `chk2`. In short, HDEAs let us validate our assertion while avoiding a needless detour through functional correctness of `checksum`.

Beyond this example. The above example is an instance of a more general pattern that arises in SL verification: *redundancy between specifications and implementations.* A typical SL specification abstracts data structures to some mathematical representation (*e.g.,* abstracting a buffer to a sequence of values) and implementation functions to mathematical functions (*e.g.,* abstracting `checksum` to *cs* above). These are then used to specify the concrete implementation. While data abstraction is undoubtedly useful in program verification, requiring a mathematical counterpart for each implementation function is especially tedious and redundant for cases where the specification ends up more or less just mirroring the implementation[5] (prominent examples include getter functions, comparison operations, and mathematical computations). In those cases, HDEAs shine: they enable one to simply talk about the result of running some code at the assertion level (so long as that result is well-defined) *without* having to develop a mathematical abstraction of it first.

[5] As we have seen in Part IV, in some cases, we can use inference to avoid the overhead of providing specifications. However, inference also has its limits (*e.g.,* when recursion or loops are involved). HDEAs explore a very different point in the design space: one still needs to provide specifications manually, but code can be used inside of the specifications.

Reducing redundancy is not the only strength of HDEAs. For example, as we will see later, HDEAs also facilitate the iterative development of proofs, whereby specifications are strengthened step by step in order to incrementally model more complex aspects of a program's behavior (see §29). Moreover, they support a new kind of automation for Iris, namely SMT solvers. That is, as we have discussed in the previous chapters, Iris is a foundational separation logic framework embedded into Rocq. A downside of its embedding within Rocq is lackluster automation for, *e.g.,* theories such as bitvectors and uninterpreted functions. In the following, we will develop the foundations for connecting *HDEAs in Iris* to *formulas in first-order logic.* This connection then allows one to benefit from the automation of an SMT solver for theories like integers, bitvectors, and uninterpreted functions when reasoning about HDEAs in Iris (see §25.3).

24.2 Daenerys

In this part of the dissertation, we introduce **Daenerys**. The main theoretical contribution of Daenerys is bringing HDEAs to Iris by extending its resource model with unstable resources. In doing so, we combine the benefits of HDEAs (described above) with the expressivity of Iris, including step-indexing (§3.2), persistent propositions (§3.3), impredicative invariants (§3.4), fine-grained concurrency (§3.5), user-defined ghost state (§3.6), and a Rocq implementation. The essence of this contribution is a new Iris assertion for HDEAs (§25):

$e \Downarrow v$, *meaning "if e is executed in the current heap, then it terminates in v".*

The assertion $e \Downarrow v$ (read "e evaluates to v") asserts the result of evaluating any deterministic, terminating, read-only expression e. It allows us to express HDEAs of traditional formulations of IDF such as "`checksum(buf)` $=_{\mathrm{IDF}}$ `chk1`" (from §24.1) as `checksum(buf)` $\Downarrow$ `chk1`.

To introduce $e \Downarrow v$ to Iris and make effective use of it, we make several technical contributions in this part of the dissertation:

Unstable resources in Iris (§26). In order to define $e \Downarrow v$, we first have to generalize the underlying model of Iris. We extend the notion of resource algebras of Iris (from §4.2) to include *unstable resources*, and we revisit the definition of the central *frame-preserving updates* (from Definition 32).[6] The key new unstable resource that we define is the *unstable points-to* $\ell \mapsto_{\mathsf{u}} v$. We use it in the definition of evaluation $e \Downarrow v$ to temporarily capture information about the memory that e accesses (see §26). Like a regular, stable points-to $\ell \mapsto v$, the unstable points-to $\ell \mapsto_{\mathsf{u}} v$ allows reading from location ℓ. However, unlike $\ell \mapsto v$, it can be freely duplicated (*i.e.,* $\ell \mapsto_{\mathsf{u}} v \vdash \ell \mapsto_{\mathsf{u}} v * \ell \mapsto_{\mathsf{u}} v$), and it can co-exist with the regular points-to at the same time (*i.e.,* $\ell \mapsto v \dashv\vdash \ell \mapsto v * \ell \mapsto_{\mathsf{u}} v$). In fact, it satisfies the equivalence $\ell \mapsto v \dashv\vdash \mathrm{acc}(\ell) * \ell \mapsto_{\mathsf{u}} v$ where $\mathrm{acc}(\ell) \triangleq \ell \mapsto \_$, illustrating that it constrains the contents of ℓ without asserting ownership.

[6] Daenerys is not the first SL with unstable resources. In other logics [Din+13; RDG14; DOs+21], typically the update is taken as the primitive and stability is derived from it. We do the opposite: stability is the primitive and updates are derived. See §31 for a comparison.

Program logic with HDEAs (§27). On top of the adapted resource model, we then build a program logic $\{P\}\ e\ \{v.\, Q(v)\}$. In typical Iris fashion (see §3), it is an expressive, higher-order program logic with impredicative invariants, step-indexing, *etc.* However, we have to be careful! The catch of unstable resources like $\ell \mapsto_{\mathsf{u}} v$ is that not all assertions can be framed anymore (see §25). That is, as we have seen in the previous parts, the central frame rule of Iris is FRAME-IRIS:

$$\textsc{frame-iris}\quad \frac{\{P\}\ e\ \{v.\, Q(v)\}}{\{P * R\}\ e\ \{v.\, Q(v) * R\}} \qquad\qquad \textsc{frame-daenerys}\quad \frac{\{P\}\ e\ \{v.\, Q(v)\}}{\{\boxplus P * \boxplus R\}\ e\ \{v.\ \boxplus\, Q(v) * \boxplus R\}}$$

and the frame rule is baked into Iris at its very core (see §26.1). However, in the presence of unstable resources like $\ell \mapsto_{\mathsf{u}} v$, this rule is no longer sound. For example, it would be unsound to frame $\ell \mapsto_{\mathsf{u}} 5$ around $\{\ell \mapsto 5\}\ \ell := 42\ \{\_.\, \ell \mapsto 42\}$, since $\ell \mapsto 42 * \ell \mapsto_{\mathsf{u}} 5$ is absurd. To recover framing, we introduce a new modality, the *frame modality* $\boxplus P$, and replace the frame rule with FRAME-DAENERYS above. The frame modality $\boxplus P$ acts as a "gate keeper": it makes sure that we only frame assertions that do not depend on ownership of unstable resources.[7]

[7] In IDF, soundness of the frame rule is typically ensured by requiring the assertions to be "self-framing". The frame modality $\boxplus P$ internalizes this notion of "self-framingness" into the logic in the form of a modality.

Automation via almost-pure assertions (§28). Equipped with evaluation $e \Downarrow v$, we then develop a very useful fragment of *almost-pure assertions*:

$$F, G : hProp ::= \phi \mid e \Downarrow v \mid \ell \mapsto_{\mathsf{u}} v \mid F \land G \mid F \lor G \mid F \Rightarrow G \mid \exists x.\, F\, x \mid \forall x.\, F\, x \mid \cdots$$

It contains actually pure assertions ϕ and is closed under standard logical connectives. It also contains connectives such as $e \Downarrow v$ and $\ell \mapsto_{\mathsf{u}} v$ that can implicitly refer to the current heap.

We use this fragment to lay the groundwork for new automation for Iris: we show a correspondence between *hProp*-assertions (using HDEAs like $e \Downarrow v$) and standard first-order logic (agnostic about heaps and state). This connection allows us, for the first time, to automate parts of an Iris proof using an SMT solver. For example, we consider a polymorphic hashmap with an equality function eq and a hash-function hash (in §29). We express the key relationship between them as:

$$\forall x, y.\, \mathsf{eq}(x, y) \equiv \mathsf{true} \Rightarrow \mathsf{hash}(x) \equiv \mathsf{hash}(y) \qquad (\textsc{eq-hash})$$
$$\textit{where } e_1 \equiv e_2 \triangleq \exists v.\, e_1 \Downarrow v \wedge e_2 \Downarrow v$$

We then encode this condition for concrete instantiations of eq and hash into a first-order logic formula, (manually) query the SMT solver Z3[8] on this formula, and assume in Rocq that it holds.

[8] Moura and Bjørner, "Z3: An efficient SMT solver", 2008 [MB08].

The use of an SMT solver means verification is not completely foundational: we trust Z3 to be sound w.r.t. standard first-order logic semantics. We do, however, show foundationally (Theorem 84 in §28.2) that our connection between *almost-pure assertions in Iris* (which use HDEAs to reason about memory) and *first-order logic* (which is agnostic about the heap) is sound.

Case studies (§29). We have applied Daenerys to several interesting case studies, demonstrating the benefits of combining IDF and Iris and of the SMT-based automation enabled by HDEAs.

Daenerys is fully mechanized in Rocq, extending the implementation of Iris and the Iris Proof Mode. See the Daenerys Rocq development for the Rocq proofs.[9]

[9] Spies et al., *Destabilizing Iris (Rocq development and appendix)*, 2025 [Spi+25b].

Chapter 25

Heap-Dependent Expression Assertions in Daenerys

In this chapter, we focus on the main heap-dependent expression assertion of Daenerys, the evaluation assertion $e \Downarrow v$. We explain how it works (§25.1), how it integrates into the program logic (§25.2), and how it supports new automation for Iris by connecting to first-order logic (§25.3). Throughout this introduction, we use the rules in Fig. 25.1.

25.1 The Evaluation Assertion

Before we explain $e \Downarrow v$, let us first introduce the language λ_{dyn} that we will be working with in the following. It extends Iris's HeapLang (see §2 and §3) with vectors $\#[v_1, \ldots, v_n]$ (*i.e.,* immutable sequences of values for, *e.g.,* strings) and bitvectors u (*i.e.,* fixed-size integers such as unsigned 64-bit integers):

$$\begin{array}{rrl} \text{Values} & v, w ::= & \cdots \mid \#[v_1, \ldots, v_n] \mid u \\ \text{Expressions} & e ::= & \cdots \mid e_1[e_2] \mid e_1[e_2 \leftarrow e_3] \mid \mathtt{size}(e) \mid \cdots \end{array}$$

Notably, just like HeapLang, λ_{dyn} does not distinguish between commands and expressions: everything is an expression, including recursive functions $\mathsf{fix}\ f\ x.\ e$ and function application $e_1\ e_2$. In particular, the combinator `iter` below—applying a function f to integers in the range $[n, m)$—is a *value* (and hence an expression) in λ_{dyn}:

$$\mathtt{iter} \triangleq \mathsf{fix}\ \mathtt{it}\ (n, m, s, f).\ \mathsf{if}\ m \leq n\ \mathsf{then}\ s\ \mathsf{else}\ \mathtt{it}(n+1, m, f\ n\ s, f)$$

Evaluation $e \Downarrow v$. Let us now explain the evaluation assertion $e \Downarrow v$. It is a simple judgment for reasoning about terminating, deterministic, read-only expressions. Intuitively, $e \Downarrow v$ means "if we execute e in the current heap, then it terminates in the value v". For example, the entailment

$$\begin{array}{l} \ell \mapsto v_{\text{vec}} \vdash e_{\text{add}} \Downarrow 42 \\ \textit{where}\ \ e_{\text{add}} \triangleq \mathtt{iter}(0, \mathtt{size}(!\,\ell), 0, \lambda i, s.\ !\,\ell[i] + s) \ \ \textit{and}\ \ v_{\text{vec}} \triangleq \#[13, 11, 6, 12], \end{array}$$

means if ℓ stores the vector v_{vec} in memory, then e_{add} (which adds up the elements of the vector stored in ℓ) evaluates to 42. We can prove it using the rules in Fig. 25.1: First, we focus on the subexpression $!\,\ell$ in evaluation position with eval-ctx for $K \triangleq \mathtt{iter}(0, \mathtt{size}(\bullet), 0, \lambda i, s.\ !\ell[i] + s)$. We can then justify the load with eval-load, leaving us with $K[v_{\text{vec}}] \Downarrow 42$. (We ask the reader for now to ignore the subscript on points-to assertions $\ell \mapsto_q v$ such as in eval-load.) Next, we compute the vector size with a pure step $\mathtt{size}(v_{\text{vec}}) \rightarrow_{\text{pure}} 4$ with eval-pure, leaving $\mathtt{iter}(0, 4, 0, \lambda i, s.\ !\ell[i] + s) \Downarrow 42$ to prove. We continue with pure steps and dereferencing ℓ until we reach $42 \Downarrow 42$, which holds by eval-val.

$$\textsc{eval-val}\quad v \Downarrow v \qquad \textsc{eval-pure}\quad e \to_{\mathsf{pure}} e' * e' \Downarrow v \vdash e \Downarrow v \qquad \textsc{eval-load}\quad \ell \mapsto_q v \vdash\, !\ell \Downarrow v \qquad \textsc{eval-ctx}\quad e \Downarrow v * K[v] \Downarrow w \vdash K[e] \Downarrow w \qquad \textsc{eval-det}\quad e \Downarrow v * e \Downarrow w \vdash v = w$$

$$\textsc{hoare-eval}\quad \{P * e \Downarrow \_\}\; e\; \{v.\, P * e \Downarrow v\} \qquad \textsc{pts-frame}\quad \ell \mapsto_q v \vdash \boxplus \ell \mapsto_q v \qquad \textsc{frame-elim}\quad \boxplus P \vdash P \qquad \textsc{eval-dupl}\quad \frac{P \vdash e \Downarrow v \qquad P \vdash Q}{P \vdash e \Downarrow v * Q}$$

$$\textsc{frame-evals}\quad \frac{P \vdash e \Downarrow \_}{\boxplus P * e \Downarrow v \vdash \boxplus(P * e \Downarrow v)} \qquad \textsc{hoare-frame}\quad \frac{\{P\}\; e\; \{v.\, Q(v)\}}{\{\boxplus P * \boxplus R\}\; e\; \{v.\; \boxplus Q(v) * \boxplus R\}}$$

Figure 25.1: A selection of core proof rules of Daenerys.

What makes $e \Downarrow v$ special—particularly from an SL perspective—is that it does not consume any ownership of the locations that e accesses. Traditionally, the separating conjunction $P * Q$ enforces that P and Q access *disjoint* parts of the heap (and more generally disjoint resources; see §4.3). However, for $e \Downarrow v$, we have $\ell \mapsto v_{\mathrm{vec}} \vdash \ell \mapsto v_{\mathrm{vec}} * e_{\mathrm{add}} \Downarrow 42$, yet clearly e_{add} accesses ℓ. The key rule is eval-dupl: when we prove $P \vdash e \Downarrow v * Q$, we do not have to split up the ownership of P between $e \Downarrow v$ and Q—as would usually be the case (see sep-split in Fig. 2.2). Instead, we can use P for proving $e \Downarrow v$ *and* Q. More broadly, this means that $e \Downarrow v$ escapes the usually linear (or affine in Iris) resource management of separation logic, which makes it easier to reason about. We will see in §26 that the underlying reason why $e \Downarrow v$ enjoys this rule is that it is based on *unstable points-tos* $\ell \mapsto_{\mathrm{u}} v$.

25.2 Evaluation and the Program Logic

As in the other chapters, we verify (effectful) programs using a program logic $\{P\}\; e\; \{v.\, Q(v)\}$ (fully introduced in §27). Let us now discuss how $e \Downarrow v$ integrates into it. For this, we return to the `checksum` example (from §24.1). Recall that our goal in this example is to avoid defining a mathematical representation *cs* of `checksum`. We start with a high-level proof sketch, this time using $e \Downarrow v$:

```
6  let buf = produce_buffer() in   ■ {buf ↦ u⃗}
7  let chk1 = checksum(buf)   in   ■ {buf ↦ u⃗ ∗ checksum(buf) ⇓ chk1}
8  read_only_client(buf);          ■ {buf ↦ u⃗ ∗ checksum(buf) ⇓ chk1}
9  let chk2 = checksum(buf)   in   ■ {buf ↦ u⃗ ∗ checksum(buf) ⇓ chk1
                                      ∗ checksum(buf) ⇓ chk2}
10 assert(chk1 == chk2)
```

We obtain `checksum(buf)`$\Downarrow$`chk1` in Line 7 and `checksum(buf)`$\Downarrow$`chk2` in Line 9, and then thread them through to the assert in Line 10. We can prove that the assert succeeds, because $e \Downarrow v$ is deterministic (eval-det), so `chk1` and `chk2` are equal at this point.

Two of these steps warrant a closer look. We discuss how we obtain $e \Downarrow v$ for `checksum` (Lines 6-7) and how we frame it past the read-only client (Lines 7-8).

Connecting evaluation and the program logic. To get from Line 6 to Line 7, we prove the Hoare triple

$$\{b \mapsto \vec{u}\}\; \mathtt{checksum}(b)\; \{v.\, b \mapsto \vec{u} * \mathtt{checksum}(b) \Downarrow v\}.$$

In general, we connect evaluation to Hoare triples with the rule hoare-eval (Fig. 25.1). It allows one to prove a Hoare triple for e if e evaluates, written $e \Downarrow \_ \triangleq \exists v.\, e \Downarrow v$. To apply hoare-eval here, it suffices to prove

$$b \mapsto_q \vec{u} \vdash \texttt{checksum}(b) \Downarrow \_ \qquad \text{(checksum-eval)}$$

meaning that if the buffer b currently stores $\vec{u}$, then $\texttt{checksum}(b)$ will terminate in some value.

At first glance, this may seem like it requires us to verify `checksum` after all, even though the assertion (in Line 10) only requires that `checksum` is deterministic. Recall that providing a mathematical specification like cs for `checksum` is exactly the overhead that we are trying to avoid by using HDEAs. Fortunately, showing that a function deterministically computes *some result* is a weaker requirement than showing that it computes *a specific result*. It suffices for `checksum` to be safe, deterministic, and terminating—but a mathematical function cs on the buffer contents is not needed. We will see in §28.1 how we can exploit this relaxation by introducing a semantic type system, which will give us (in many cases) a simple way of proving $e \Downarrow \_$ via "type checking" e.

Framing. As the final piece of the proof (Lines 7-8), let us turn our attention to *framing*. Recall that, usually in Iris, once we proved a Hoare triple $\{P\}\ e\ \{v.\, Q(v)\}$, we can frame any assertion R around it (see frame-iris in §24.2). However, the assertion $e \Downarrow v$ is special in that it *cannot* be framed on its own. For example, it would be unsound to frame $e_{\text{add}} \Downarrow 42$ around the Hoare triple $\{\ell \mapsto v_{\text{vec}}\}\ \ell := \texttt{\#[]}\ \{\_.\, \ell \mapsto \texttt{\#[]}\}$, since the contents of ℓ change. Instead, to frame $e \Downarrow v$, we have to frame enough ownership alongside it to ensure that the result of e does not change. For example, the ownership of $\ell \mapsto v_{\text{vec}}$ ensures that e_{add} does not change, so we can frame $R \triangleq \ell \mapsto v_{\text{vec}} * e_{\text{add}} \Downarrow 42$.

In our example (Lines 7-8), we must be careful not to frame $\texttt{buf} \mapsto \vec{u}$ around the read-only client, since the client *also* needs ownership of the buffer `buf` to justify reading from it. To resolve this tension, we use *fractional permissions*.[1] More specifically, we use the *fractional points-to* $\ell \mapsto_q v$, which generalizes the regular points-to $\ell \mapsto v$ (from §4.2.2).[2] For $q = 1$, it is the same as $\ell \mapsto v$, and for any fraction $0 < q < 1$, it allows reading but not writing. As with fractional ghost variables (see §4.2.2), one can split and combine the points-to assertions based on their fractions (*i.e.*, $\ell \mapsto_{q+q'} v \dashv\vdash \ell \mapsto_q v * \ell \mapsto_{q'} v$), and two points-to assertions for the same location agree on the value stored there (*i.e.*, $\ell \mapsto_q v * \ell \mapsto_{q'} w \vdash v = w$). In this case, we split the ownership of $\texttt{buf} \mapsto \vec{u}$ into $\texttt{buf} \mapsto_{1/2} \vec{u} * \texttt{buf} \mapsto_{1/2} \vec{u}$. We use one half $\texttt{buf} \mapsto_{1/2} \vec{u}$ to frame $\texttt{checksum(buf)} \Downarrow \texttt{chk1}$, and we give the other to the read-only client:

$$\left\{\texttt{buf} \mapsto_{1/2} \vec{u}\right\}\ \texttt{read\_only\_client(buf)}\ \left\{\_.\, \texttt{buf} \mapsto_{1/2} \_\right\} \qquad \text{(roc-spec)}$$

[1] Boyland, "Checking interference with fractional permissions", 2003 [Boy03]; Bornat et al., "Permission accounting in separation logic", 2005 [Bor+05].

[2] To obtain a fractional points-to assertion, one can choose the resource algebra $Auth(Loc \xrightarrow{\text{fin}} ((0,1],+) \times Ag(Val)))$. We will discuss a version of fractional points-to assertions with the unstable points-to assertion $\ell \mapsto_{\text{u}} v$ in §26.3.2.

The frame modality $\boxplus P$. IDF ensures soundness of the frame rule by ensuring that the framed assertion is "self-framing" (*i.e.*, contains non-zero ownership for each memory location it depends on). We internalize this notion into Daenerys with a revised frame rule, hoare-frame, and a new modality, the *frame modality* $\boxplus P$. The latter is the "gate keeper" that ensures that we frame enough ownership such that P will not be invalidated. To explain both, let us zoom in on the proof step around the read-only client (with two intermediate proof states added in ▸ violet):

```
11 ■ {buf ↦ u⃗ ∗ checksum(buf) ⇓ chk1}
12 ▶ {⊞buf ↦_{1/2} u⃗ ∗ ⊞(buf ↦_{1/2} u⃗ ∗ checksum(buf) ⇓ chk1)}
13 read_only_client(buf);
14 ▶ {⊞buf ↦_{1/2} _ ∗ ⊞(buf ↦_{1/2} u⃗ ∗ checksum(buf) ⇓ chk1)}
15 ■ {buf ↦ u⃗ ∗ checksum(buf) ⇓ chk1}
```

We want to pass $P \triangleq \texttt{buf} \mapsto_{1/2} \vec{u}$ to the read-only client and frame $R \triangleq \texttt{buf} \mapsto_{1/2} \vec{u} * \texttt{checksum(buf)} \Downarrow \texttt{chk1}$. To do so, HOARE-FRAME asks us to split our precondition into two parts P and R and put both into a frame modality "$\boxplus$".[3] We do so in Lines 11-12: For P, we use that fractional ownership of a points-to assertion $\ell \mapsto_q v$ can always be put into a frame modality (PTS-FRAME), because it precludes others from modifying ℓ. For R, we exploit that we can frame $e \Downarrow v$ if we combine it with enough ownership to prove that e evaluates (FRAME-EVALS). Since R contains $\texttt{buf} \mapsto_{1/2} \vec{u}$, we can use CHECKSUM-EVAL to prove that checksum(buf) evaluates and obtain $\boxplus R$. (Fractional ownership is enough for CHECKSUM-EVAL, since checksum does not need to modify the buffer.)

[3] The reader may wonder whether both frame modalities in the precondition of HOARE-FRAME are needed. The frame modality around R ensures that e does not invalidate R. We discuss the one around P in §27.1.

Finally, after the read-only client, we have to establish the postcondition in Lines 14-15. But this is easy! The frame modality $\boxplus P$ tells us, in particular, that P holds, so we can just eliminate it (FRAME-ELIM). We can then re-assemble the full points-to assertion for the buffer from the two halves. This completes the last missing step in our example from the start of this subsection.

25.3 Evaluation and First-Order Logic

We will now show how to connect evaluation $e \Downarrow v$ to first-order logic to enable new automation via SMT solvers. As motivation, consider a small variation of the checksum example:

```
16 let buf = produce_buffer() in ■ {buf ↦ u⃗}
17 let chk = checksum(buf)    in ■ {buf ↦ u⃗ ∗ checksum(buf) ⇓ chk}
18 read_only_client(buf);        ■ {buf ↦ u⃗ ∗ checksum(buf) ⇓ chk}
19 assert(validate(buf, chk))    ■ {buf ↦ u⃗ ∗ checksum(buf) ⇓ chk
                                    ∗ validate(buf,chk) ⇓ true}
```

In this version, we call a function validate to validate the buffer buf against the checksum chk. We consider this version, because—unlike for checksum where it was enough to know that it computes *some* value—for validate, it actually matters *what* the function does. That is, to show that the assert in Line 19 succeeds, validate cannot be just any function. Instead, we need that *for any buffer* b*,* $\texttt{validate}(b, c)$ *returns true if* c *is the result of* $\texttt{checksum}(b)$.

Almost-pure assertions. Suppose checksum returns a 64-bit unsigned integer. Then we can make the desired relationship between validate and checksum formal in Daenerys as:

$$\forall b, c.\, \mathsf{buffer}(b) \Rightarrow \mathsf{u64}(c) \Rightarrow \texttt{checksum}(b) \Downarrow c \Rightarrow \texttt{validate}(b, c) \Downarrow \texttt{true} \tag{CHECK-VAL}$$

where buffer is defined below and u64 ensures that c is a 64-bit unsigned integer.

To state and prove properties relating HDEAs like CHECK-VAL, we use *almost-pure assertions* in Daenerys—assertions $F, G : \mathit{hProp} \subseteq \mathit{iProp}$ that largely behave pure (*i.e.,* non-linear), yet can also refer to the heap:

$$F, G : \mathit{hProp} ::= \phi \mid e \Downarrow v \mid \ell \mapsto_{\mathrm{u}} v \mid F \wedge G \mid F \vee G \mid F \Rightarrow G \mid \exists x.\, F\,x \mid \forall x.\, F\,x \mid \cdots$$

They are a fragment of (our version of) Iris's propositions *iProp*, containing actually-pure assertions ϕ, evaluation assertions $e \Downarrow v$, *unstable* points-to assertions $\ell \mapsto_{\mathsf{u}} v$ (see §26), and standard logical connectives including impredicative, higher-order quantification (which gives rise to least- and greatest fixpoints). The unstable points-to allows one (together with $e \Downarrow v$) to constrain the current memory. For example, we define $\mathrm{buffer}(b) \triangleq \exists \vec{u}.\, b \mapsto_{\mathsf{u}} \vec{u}$ to ensure that b is a buffer in memory.

The *hProp*-fragment is quite expressive: For example, we can use it to reason about evaluation $e \Downarrow v$ by restating the proof rules from Fig. 25.1 in *hProp* with $\ell \mapsto_{\mathsf{u}} v$ in place of $\ell \mapsto_q v$, conjunction in place of separating conjunction, and implication in place of entailment:

$$e \rightarrow_{\mathsf{pure}} e' \wedge e' \Downarrow v \Rightarrow e \Downarrow v \qquad \ell \mapsto_{\mathsf{u}} v \Rightarrow\ !\ell \Downarrow v \qquad e \Downarrow v \wedge K[v] \Downarrow w \Rightarrow K[e] \Downarrow w$$

Hence, one can prove, *e.g.*, $\ell \mapsto_{\mathsf{u}} v_{\mathrm{vec}} \Rightarrow e_{\mathrm{add}} \Downarrow 42$ in *hProp* analogously to §25.1. We will make the *hProp*-fragment precise in §28.

First-order logic. Suppose we want to prove check-val for a concrete implementation of `validate`, such as

```
validate(buf,chk) = (checksum(buf) xor chk == 0).
```

Does it satisfy the property check-val (and, consequently, does the assert in Line 19 succeed)? The answer is yes!

To prove it, one option would be to roll up our sleeves and use the rules for $e \Downarrow v$ above. (In Rocq, we have instantiated the Iris Proof Mode[4] with *hProp* for such cases.) However, Daenerys also provides a second option: *solve the problem automatically in first-order logic.* If one squints a little, check-val looks a lot like a formula in first-order logic: the functions correspond to first-order function symbols, the predicates to first-order sorts, and the evaluation $e \Downarrow v$ to equality. Following this analogy, check-val could be restated in first-order logic (indicated in blue) as:

$$\dot{\forall}(\mathsf{b} : \mathsf{buffer}), (\mathsf{c} : \mathsf{bv}\,64).\, \mathsf{checksum}(\mathsf{b}) \doteq_{\mathsf{bv}\,64} \mathsf{c} \mathrel{\dot{\Rightarrow}} \mathsf{validate}(\mathsf{b}, \mathsf{c}) \doteq_{\mathsf{bool}} \mathsf{true} \tag{check-val-fo}$$

This is basically how IDF-based verifiers like Viper[5] reason about HDEAs (although the details differ substantially; see §28.2). In Daenerys, we develop a *foundational* justification for this correspondence (in §28.2): we show that one can translate a first-order logic formula π such as check-val-fo (which has no concept of "memory") to an *hProp*-assertion $\langle \pi \rangle_{\mathsf{F}}$ such as check-val (which refers to the current memory via $\ell \mapsto_{\mathsf{u}} v$ and $e \Downarrow v$) such that if π holds, then we can assume $\langle \pi \rangle_{\mathsf{F}}$ in Iris.

SMT solvers. The main use-case for this connection is laying the groundwork for connecting Iris to SMT solvers such as Z3[6] or CVC5[7] to benefit from their built-in automation. Of course, SMT solvers are not foundational, and we by no means attempt to verify an SMT solver. Instead, Daenerys provides the assurance that *if* π holds in first-order logic—the language of SMT solvers—then the *hProp*-assertion $\langle \pi \rangle_{\mathsf{F}}$—indirectly referring to the heap—can be soundly used in Iris. For example, for assert, we derive the following proof rule (from

[4] Krebbers, Timany, and Birkedal, "Interactive proofs in higher-order concurrent separation logic", 2017 [KTB17].

[5] Müller, Schwerhoff, and Summers, "Viper: A verification infrastructure for permission-based reasoning", 2017 [MSS17].

[6] Moura and Bjørner, "Z3: An efficient SMT solver", 2008 [MB08].

[7] Barbosa et al., "cvc5: A versatile and industrial-strength SMT solver", 2022 [Bar+22].

our generic result, Theorem 84):

$$\frac{\text{HOARE-ASSERT}\quad \vDash \pi \qquad P \vdash e \Downarrow \_ \qquad P * \langle \pi \rangle_{\mathsf{F}} \vdash e \Downarrow \mathsf{true}}{\{P\}\ \mathsf{assert}(e)\ \{\_.\ P\}}$$

where $\vDash \pi$ means π is provable in first-order logic with knowledge of, *e.g.*, numbers and bitvectors. HOARE-ASSERT means that an assert e succeeds if (1) π holds in first-order logic, (2) P suffices to prove that e will evaluate,[8] and (3) *assuming* $\langle \pi \rangle_{\mathsf{F}}$, one can prove that e evaluates to true.

For example, given the implementation of `validate`, Z3 can prove CHECK-VAL-FO automatically, because it knows that $u\ \mathsf{xor}\ u = 0$ for any 64-bit unsigned integer u. Thus, if we trust Z3 to be sound w.r.t. to standard first-order logic semantics, we can verify the assert in Line 19—even though it uses heap-accessing functions like `validate`—without ever (1) specifying `checksum` as a mathematical function cs and (2) reasoning about bitvector arithmetic in Iris.

[8] Analogously to the rule HOARE-FRAME, we need the assumption $P \vdash e \Downarrow \_$ here to ensure that—given the ownership of P—the result of e is not affected by any other parts of the program such as concurrent threads. We discuss a proof technique for this assumption in §28.1.

Chapter 26

Destabilizing the Foundations of Iris

Having used the evaluation assertion $e \Downarrow v$ (in §25), let us now define it. To do so, we have to go down to the core of Iris and change its underlying notion of *resources*. As we have already touched upon in the previous chapters, almost everything in Iris boils down to resources, including *the heap* with $\ell \mapsto_q v$, but also *state transition systems*,[1] *invariants*,[2] *refinements*,[3] *time complexity*,[4] and even to some extent *step-indexing* (see Part III). In Daenerys, we generalize Iris's resources one step further by introducing *unstable resources*.

[1] Jung et al., "Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning", 2015 [Jun+15].

[2] Jung et al., "Higher-order ghost state", 2016 [Jun+16].

[3] Turon, Dreyer, and Birkedal, "Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency", 2013 [TDB13]; Frumin, Krebbers, and Birkedal, "ReLoC: A mechanised relational logic for fine-grained concurrency", 2018 [FKB18].

[4] Mével, Jourdan, and Pottier, "Time credits and time receipts in Iris", 2019 [MJP19].

26.1 Unstable Resources

To define $e \Downarrow v$, we need a new resource assertion, the *unstable points-to* $\ell \mapsto_{\mathrm{u}} v$. Like a normal points-to $\ell \mapsto_q v$, it asserts the value of ℓ in the current heap. But unlike $\ell \mapsto_q v$, (1) it can co-exist with the full ownership of ℓ in the sense that $\ell \mapsto_q v * v = w \dashv\vdash \ell \mapsto_q v * \ell \mapsto_{\mathrm{u}} w$ holds, including for $q = 1$ and (2) it does not prevent updates to ℓ. As we will see below, this means it goes beyond the resource model of Iris. But before we get there, let us first use it to define $e \Downarrow v$:

$$e \Downarrow v \triangleq \exists h.\ (e, h) \rightsquigarrow^*_{\mathrm{det}} (v, h) * (\ast_{\ell \mapsto w \in h}\, \ell \mapsto_{\mathrm{u}} w) \qquad \text{(EVAL-DEF)}$$

That is, e evaluates to v if (1) there is a heap fragment h in which e deterministically steps to v in the operational semantics,[5] written $(e, h) \rightsquigarrow^*_{\mathrm{det}} (v, h)$, and (2) we have unstable points-to assertions $\ell \mapsto_{\mathrm{u}} w$ for all entries in h, which ensures that e evaluates to v in (a fragment of) *the current heap*.

[5] Concretely, the relation $(e, h) \rightsquigarrow_{\mathrm{det}} (e', h)$ restricts the operational semantics of λ_{dyn} $(e, h) \rightsquigarrow (e', h', es)$ to those steps that (1) do not change the heap h and (2) do not fork any additional threads. They are deterministic in λ_{dyn} (and HeapLang).

Framing by construction. We will define $\ell \mapsto_{\mathrm{u}} v$ in §26.3.2. Let us first see why $\ell \mapsto_{\mathrm{u}} v$ requires us to modify the model of Iris. The issue is that $\ell \mapsto_{\mathrm{u}} v$ cannot soundly be framed around code that modifies ℓ, yet Iris "bakes in" framing at its very core as we have seen in §4.2. More specifically, as part of its design philosophy,[6] Iris makes framing *the defining feature* (see Definition 32 in §4.2) of which resource updates it permits via its frame preserving update $a \rightsquigarrow b$. We will now explore the essence of the problem and how we resolve it.

[6] See §7.1 in Jung, "Understanding and evolving the Rust programming language", 2020 [Jun20].

To illustrate the issue with Iris's original resources, we focus on the *exclusive resource algebra* $Ex(\mathbb{N})$.[7] Recall from §4.2.1 that it essentially has only a single resource, $\mathrm{ex}(n)$, carrying full ownership of the number n, and it adheres to the rules:

[7] Technically, we consider the exclusive resource algebra extended with a unit element ε here, so $\mathrm{option}(Ex(\mathbb{N}))$.

EX-VALID	EX-EXCL	EX-UPD
$\mathrm{ex}(n) \in \mathcal{V}$	$\mathrm{ex}(n) \cdot \mathrm{ex}(m) \notin \mathcal{V}$	$\mathrm{ex}(n) \rightsquigarrow \mathrm{ex}(m)$

The resource $\mathrm{ex}(n)$ on its own is valid (EX-VALID), and if we compose it with another copy of $\mathrm{ex}(\_)$ it becomes invalid (EX-EXCL). In other words, $\mathrm{ex}(n)$ carries full ownership: it cannot exist at the same time as another copy $\mathsf{ex}(m)$. Lastly, we can update $\mathsf{ex}(n)$ to $\mathsf{ex}(m)$ for any m, since $\mathsf{ex}(n)$ carries full ownership (ex-upd). As we have seen in §4.2.2, the reader can think of $\mathsf{ex}(n)$ as the resource underlying $\ell \mapsto_1 n$ for a particular location ℓ.[8]

[8] That is, recall that in the heap resource algebra from §4.2.2, we define $k \mapsto^{\gamma} v \triangleq \boxed{\circ[k \mapsto \mathsf{ex}(v)]}^{\gamma}$.

To illustrate why we need to update the foundations of Iris, we will now show that unstable points-to assertions are incompatible with its model. Consider an extension of the exclusive resource algebra with a resource $\mathsf{tmp}(n)$, the analog of $\ell \mapsto_{\mathrm{u}} n$. This resource co-exists with $\mathsf{ex}(n)$ yet—crucially—should still allow updating ex via ex-upd. Formally, we want that $\mathsf{ex}(n) = \mathsf{ex}(n) \cdot \mathsf{tmp}(n)$ (*i.e.,* we can always create a temporary copy) and $\mathsf{tmp}(n) \cdot \mathsf{ex}(m) \in \mathcal{V} \Rightarrow n = m$ (*i.e.,* the two agree on the current value). Unfortunately, if these two hold, then ex-upd *is no longer true.* To understand why, we have to take a closer look at the frame-preserving update $a \rightsquigarrow b$ (from Definition 32; specialized to a single successor element and unital resource algebras). Its defining characteristic is that it preserves all valid frames a_f:

$$a \rightsquigarrow b \triangleq \forall a_f.\, (a \cdot a_f) \in \mathcal{V} \Rightarrow (b \cdot a_f) \in \mathcal{V}$$

This breaks ex-upd in the presence of $\mathsf{tmp}(n)$. In ex-upd, we have $a = \mathsf{ex}(n)$. This makes the resource $a_f = \mathsf{tmp}(n)$ a valid frame of a. But for $n \neq m$, the resource a_f is not a valid frame of $b = \mathsf{ex}(m)$ (*i.e.,* $\mathsf{tmp}(n) \cdot \mathsf{ex}(m) \notin \mathcal{V}$ for $n \neq m$). Thus, there are no more updates of ex as soon as we add $\mathsf{tmp}(n)$.

Unstable resources and stable updates. This puts us in a pickle! How can we have both $\mathsf{tmp}(n)$ and ex-upd? The issue is that $a \rightsquigarrow b$ preserves *too many* frames. Intuitively, we should be allowed to update $\mathsf{ex}(n)$ to $\mathsf{ex}(m)$, and the update should invalidate temporary copies $\mathsf{tmp}(n)$ rather than preserve them.

To realize this intuition, we extend Iris's resource algebras with *unstable resources.* More specifically, we add two projections $|a|_{\mathrm{st}}$ and $|a|_{\mathrm{unst}}$ such that $a = |a|_{\mathrm{st}} \cdot |a|_{\mathrm{unst}}$. The first projection, $|a|_{\mathrm{st}}$, yields the stable part of a resource. (Usually, in Iris, all resources are stable.) The second projection, $|a|_{\mathrm{unst}}$, yields the unstable part. We then define a new *stable update* $a \rightsquigarrow_{\mathrm{st}} b$, which only preserves the stable parts of the frame resource a_f:

$$a \rightsquigarrow_{\mathrm{st}} b \triangleq \forall a_f.\, (a \cdot a_f) \in \mathcal{V} \Rightarrow (b \cdot |a_f|_{\mathrm{st}}) \in \mathcal{V}$$

Let us consider this definition in the context of our example. In the example, $\mathsf{tmp}(n)$ represents temporary information about the value of $\mathsf{ex}(n)$. Thus, we define $|\mathsf{tmp}(n)|_{\mathrm{st}} \triangleq \varepsilon$, making it an unstable resource. More specifically, for $a_f = \mathsf{tmp}(n)$, the stable update $\rightsquigarrow_{\mathrm{st}}$ now *erases* $\mathsf{tmp}(n)$ *from the frame*, such that ex-upd holds for $\rightsquigarrow_{\mathrm{st}}$ (*i.e.,* $\mathsf{ex}(n) \rightsquigarrow_{\mathrm{st}} \mathsf{ex}(m)$).

Of course, as we have seen in §4.3, the frame preserving update $a \rightsquigarrow b$ is a corner stone of resource reasoning in Iris, since it underlies Iris's *update modality* $\Rrightarrow P$. Thus, the introduction of unstable resources (and of $a \rightsquigarrow_{\mathrm{st}} b$ specifically) ripples through all layers of Iris. We will now discuss how unstable resources give rise to new modalities (§26.2) and affect resource algebras (§26.3). We will then discuss how they alter the program logic (§27).

SUPD-OWN
$$\frac{a \rightsquigarrow_{\mathsf{st}} b}{\mathsf{Own}\,(a) \vdash |\!\!\Rrightarrow_{\mathsf{st}} \mathsf{Own}\,(b)}$$

SUPD-MONO
$$\frac{P \vdash Q}{|\!\!\Rrightarrow_{\mathsf{st}} P \vdash |\!\!\Rrightarrow_{\mathsf{st}} Q}$$

SUPD-UPD
$$|\!\!\Rrightarrow P \vdash |\!\!\Rrightarrow_{\mathsf{st}} P$$

SUPD-INTRO
$$P \vdash |\!\!\Rrightarrow_{\mathsf{st}} P$$

SUPD-TRANS
$$|\!\!\Rrightarrow_{\mathsf{st}} |\!\!\Rrightarrow_{\mathsf{st}} P \vdash |\!\!\Rrightarrow_{\mathsf{st}} P$$

SUPD-FRAME
$$(\boxplus P) * (|\!\!\Rrightarrow_{\mathsf{st}} Q) \vdash |\!\!\Rrightarrow_{\mathsf{st}} (\boxplus P) * Q$$

FRAME-MONO
$$\frac{P \vdash Q}{\boxplus P \vdash \boxplus Q}$$

FRAME-ELIM
$$\boxplus P \vdash P$$

FRAME-IDEMP
$$\boxplus P \vdash \boxplus \boxplus P$$

FRAME-EXISTS
$$\boxplus(\exists x.\, P\, x) \dashv\vdash \exists x.\, (\boxplus P\, x)$$

FRAME-ALL
$$\boxplus(\forall x.\, P\, x) \dashv\vdash \forall x.\, (\boxplus P\, x)$$

FRAME-LATER
$$\boxplus(\triangleright P) \dashv\vdash \triangleright(\boxplus P)$$

FRAME-SEP
$$(\boxplus P) * (\boxplus Q) \vdash \boxplus(P * Q)$$

FRAME-PERS
$$\Box P \vdash \boxplus P$$

FRAME-OWN
$$\mathsf{Own}\,(a) \vdash \boxplus \mathsf{Own}\,(|a|_{\mathsf{st}})$$

UNSTABLE-MONO
$$\frac{P \vdash Q}{\divideontimes P \vdash \divideontimes Q}$$

UNSTABLE-ELIM
$$\divideontimes P \vdash P$$

UNSTABLE-IDEMP
$$\divideontimes P \vdash \divideontimes \divideontimes P$$

UNSTABLE-EXISTS
$$\divideontimes(\exists x.\, P\, x) \dashv\vdash \exists x.\, (\divideontimes P\, x)$$

UNSTABLE-ALL
$$\divideontimes(\forall x.\, P\, x) \dashv\vdash \forall x.\, (\divideontimes P\, x)$$

UNSTABLE-LATER
$$\divideontimes(\triangleright P) \dashv\vdash \triangleright(\divideontimes P)$$

UNSTABLE-SEP
$$(\divideontimes P) * (\divideontimes Q) \dashv\vdash \divideontimes(P * Q)$$

UNSTABLE-OWN
$$\mathsf{Own}\,(a) \vdash \divideontimes \mathsf{Own}\,(|a|_{\mathsf{unst}})$$

UNSTABLE-IMPL
$$\divideontimes P \Rightarrow \divideontimes Q \vdash \divideontimes(\divideontimes P \Rightarrow \divideontimes Q)$$

UNSTABLE-DUPL
$$(\divideontimes P) \wedge Q \dashv\vdash (\divideontimes P) * Q$$

Figure 26.1: Base logic rules for the stable update modality $|\!\!\Rrightarrow_{\mathsf{st}} P$, the frame modality $\boxplus P$, and the unstable modality $\divideontimes P$

26.2 Extending the Base Logic

We start by lifting the new resource algebra operations (*i.e.,* the stable update $a \rightsquigarrow_{\mathsf{st}} b$ and the projections $|a|_{\mathsf{st}}$ and $|a|_{\mathsf{unst}}$) to the assertion level. To this end, we define three new modalities in Daenerys:[9]

$$\boxplus P \triangleq \{(n, a) \mid (n, |a|_{\mathsf{st}}) \in P\} \qquad \divideontimes P \triangleq \{(n, a) \mid (n, |a|_{\mathsf{unst}}) \in P\}$$

$$|\!\!\Rrightarrow_{\mathsf{st}} P \triangleq \left\{(n, a) \mid \forall m \le n.\, \forall a_f.\, a \cdot a_f \in \mathcal{V} \Rightarrow \exists b.\, b \cdot |a_f|_{\mathsf{st}} \in \mathcal{V} \wedge (m, b) \in P\right\}$$

[9] Technically, in Rocq, we extend the notion of Iris's step-indexed resources, so-called "cameras". To ease the presentation, we stay here at the same level of abstraction as in §4.2.

In short, the *frame modality* $\boxplus P$ must be proven using only the stable parts of the current resource, and analogously the *unstable modality* $\divideontimes P$ only using the unstable parts. The *stable update modality* $|\!\!\Rrightarrow_{\mathsf{st}} P$ reflects $a \rightsquigarrow_{\mathsf{st}} b$ into the logic (analogously to how $|\!\!\Rrightarrow P$ reflects $a \rightsquigarrow b$ in Iris; see §4.3). We take a closer look at each one and discuss their proof rules, depicted in Fig. 26.1.

The stable update modality. The *stable update modality* $|\!\!\Rrightarrow_{\mathsf{st}}$ allows us to perform stable updates $a \rightsquigarrow_{\mathsf{st}} b$ on resources (SUPD-OWN). For $|\!\!\Rrightarrow_{\mathsf{st}}$ to be a usable update modality, it is important that the definition of $a \rightsquigarrow_{\mathsf{st}} b$ ensures that $|\!\!\Rrightarrow_{\mathsf{st}}$ retains (almost; explained below) the same compositionality as the normal update $|\!\!\Rrightarrow$. Concretely, it is still a monad (SUPD-INTRO, SUPD-MONO, and SUPD-TRANS), which is important to make updates practically usable and integrate them into the weakest precondition (see §27.2).[10] Moreover, we have $|\!\!\Rrightarrow P \vdash |\!\!\Rrightarrow_{\mathsf{st}} P$ (SUPD-UPD), which ensures backwards compatibility: existing resource algebra constructions such as user-defined ghost state (*e.g.,* from §3.6) can still be reused, since their updates can be turned into stable updates.[11]

[10] At the resource level, the important properties of $\rightsquigarrow_{\mathsf{st}}$ are *reflexivity* (for SUPD-INTRO) and *transitivity* (for SUPD-TRANS).

[11] At the resource level, the important property here is that $a \rightsquigarrow b$ implies $a \rightsquigarrow_{\mathsf{st}} b$. It is also important that $\rightsquigarrow_{\mathsf{st}}$ does not also apply $|\_|_{\mathsf{st}}$ to b.

The frame modality. The *frame modality* $\boxplus$ illustrates the key difference between the original update $|\!\!\Rrightarrow$ and the new update $|\!\!\Rrightarrow_{\mathsf{st}}$, namely we can only *frame* assertions that are guarded by a frame modality (SUPD-FRAME). This rule holds without $\boxplus$ for $\Rrightarrow$ in Iris (see UPD-FRAME in §3.6) and is the basis for Iris's frame rule (FRAME-IRIS in §24.2), since $\Rrightarrow$ is used in the definition of the weakest precondition (see §4.1). By adding frame modalities to it here (and using $\Rrightarrow_{\mathtt{st}}$), we make sure that it is sound for stable updates to remove unstable resources (*e.g.*, the temporary resource $\mathsf{tmp}(n)$ from above).

Moreover, the frame modality is a co-monad (like the persistency modality $\square$ from §3.3): it is monotone (FRAME-MONO), we can always eliminate it (FRAME-ELIM), and it is idempotent (FRAME-IDEMP). The rule FRAME-ELIM means we never have to worry about "getting rid" of a frame modality (from our assumptions). The rule FRAME-IDEMP means we can always "add another" frame modality, which can be useful for moving inside a frame modality (*e.g.*, with FRAME-MONO). In addition, the frame modality distributes over existential quantification (FRAME-EXISTS), universal quantification (FRAME-ALL), and the later modality (FRAME-LATER). It *does not*—in both directions—distribute over the separating conjunction: we can combine two frameable assertions (FRAME-SEP), but the opposite direction does not hold.[12] Furthermore, to make sure that persistent assertions $\square P$ (§3.3) keep their usual meaning (*i.e.*, once created, they persist across updates and state changes), they are always frameable (FRAME-PERS). Lastly, if we own a resource, we frameably own its stable part (FRAME-OWN).

The unstable modality. The *unstable modality* $\divideontimes$ reflects a key property of unstable resources into the logic that we have not discussed yet: *they are duplicable*. More specifically, we will require $a = a \cdot |a|_{\mathrm{unst}}$ (in §26.3). The result is that when we prove an unstable assertion $\divideontimes P$, intuitively, we do not have to give up any ownership to do so. Formally, it means that the rule UNSTABLE-DUPL holds, which says that ordinary conjunction and separating conjunction coincide for unstable assertions.[13]

Like the frame modality, the unstable modality is a co-monad: it is monotone (UNSTABLE-MONO), we can always eliminate it (UNSTABLE-ELIM), and it is idempotent (UNSTABLE-IDEMP). It also distributes over existential quantification (UNSTABLE-EXISTS), universal quantification (UNSTABLE-ALL), and the later modality (UNSTABLE-LATER). Unlike the frame modality, the unstable modality *does* distribute over separation conjunction in both directions (UNSTABLE-SEP). When we own a resource a, we always unstably own its unstable part (UNSTABLE-OWN).

Furthermore, we have the rule UNSTABLE-IMPL, which effectively means that the implication between two unstable assertions $\divideontimes P$ and $\divideontimes Q$ is itself an unstable assertion. We will use it in §28 to show that implication is an almost-pure assertion (*i.e.*, that the fragment *hProp* is closed under implication), which makes the fragment of almost-pure assertions considerably more expressive. Obtaining UNSTABLE-IMPL is non-trivial. The trick to get it are two dedicated axioms about resources in our definition of resource algebras (see RA-UNSTABLE-FLIP and RA-UNSTABLE-EXTENSION in Fig. 26.2).

26.3 Resource Algebras with Unstable Elements

Let us now turn to the resource algebras underlying the logic. In Daenerys, we introduce *partially stable resource algebras* $M = (\mathcal{A}, \cdot, \varepsilon, |\_|_{\mathrm{core}}, \mathcal{V}, |\_|_{\mathrm{st}}, |\_|_{\mathrm{unst}})$, consisting of (1) a carrier set $\mathcal{A}$; (2) a composition $a \cdot b$ for $a, b \in \mathcal{A}$; (3) a unit ε;

[12] That is, the rule $\boxplus(P * Q) \vdash \boxplus P * \boxplus Q$ is *unsound*. For example, it would be unsound to turn $\boxplus(\ell \mapsto v * \ell \mapsto_{\mathrm{u}} v)$ into $(\boxplus \ell \mapsto v) * (\boxplus \ell \mapsto_{\mathrm{u}} v)$, because $(\boxplus \ell \mapsto v)$ could be used to update ℓ invalidating $(\boxplus \ell \mapsto_{\mathrm{u}} v)$ as a frame.

[13] The reader may wonder about the relationship between persistency $\square P$ (backed by $|\_|_{\mathrm{core}}$) and unstable propositions $\divideontimes P$ (backed by $|\_|_{\mathrm{unst}}$): unstable resources can describe larger parts of a resource, since they are only temporary whereas persistent propositions are stable (FRAME-PERS). For example, we have $|\mathsf{ex}(n)|_{\mathrm{core}} = \varepsilon$ and $|\mathsf{ex}(n)|_{\mathrm{unst}} = \mathsf{tmp}(n)$.

Resource Algebra

A *partially stable resource algebra* $M = (\mathcal{A}, \cdot, \varepsilon, |\_|_{\mathrm{core}}, \mathcal{V}, |\_|_{\mathrm{st}}, |\_|_{\mathrm{unst}})$ is a *unital resource algebra* (§4.2):

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c \qquad a \cdot b = b \cdot a \qquad |a|_{\mathrm{core}} \cdot a = a \qquad ||a|_{\mathrm{core}}|_{\mathrm{core}} = |a|_{\mathrm{core}} \qquad a \preccurlyeq b \Rightarrow |a|_{\mathrm{core}} \preccurlyeq |b|_{\mathrm{core}}$$

$$(a \cdot b) \in \mathcal{V} \Rightarrow a \in \mathcal{V} \qquad \varepsilon \in \mathcal{V} \qquad \varepsilon \cdot a = a \qquad |\varepsilon|_{\mathrm{core}} = \varepsilon$$

where $a \preccurlyeq b \triangleq \exists c.\ a \cdot c = b$ and $a \rightsquigarrow B \triangleq \forall a_f.\ (a \cdot a_f) \in \mathcal{V} \Rightarrow \exists b \in B.\ (b \cdot a_f) \in \mathcal{V}$.

with Stable and Unstable Elements

$$||a|_{\mathrm{st}}|_{\mathrm{st}} = |a|_{\mathrm{st}} \quad (\textsc{ra-stable-idemp}) \qquad ||a|_{\mathrm{unst}}|_{\mathrm{unst}} = |a|_{\mathrm{unst}} \quad (\textsc{ra-unstable-idemp})$$

$$|a \cdot b|_{\mathrm{st}} = |a|_{\mathrm{st}} \cdot |b|_{\mathrm{st}} \quad (\textsc{ra-stable-distr}) \qquad a \preccurlyeq b \Rightarrow |a|_{\mathrm{unst}} \preccurlyeq |b|_{\mathrm{unst}} \quad (\textsc{ra-unstable-mono})$$

$$||a|_{\mathrm{core}}|_{\mathrm{st}} = |a|_{\mathrm{core}} \quad (\textsc{ra-core-stable}) \qquad |a \cdot |b|_{\mathrm{unst}}|_{\mathrm{unst}} = ||a|_{\mathrm{unst}} \cdot b|_{\mathrm{unst}} \quad (\textsc{ra-unstable-flip})$$

$$|a|_{\mathrm{st}} \cdot |a|_{\mathrm{unst}} = a \quad (\textsc{ra-decompose}) \qquad a \in \mathcal{V} \Rightarrow |a|_{\mathrm{unst}} \cdot b \in \mathcal{V} \Rightarrow a \cdot |b|_{\mathrm{unst}} \in \mathcal{V} \quad (\textsc{ra-unstable-extension})$$

$$|a|_{\mathrm{unst}} \cdot a = a \quad (\textsc{ra-unstable-dupl})$$

where $a \rightsquigarrow_{\mathrm{st}} B \triangleq \forall a_f.\ (a \cdot a_f) \in \mathcal{V} \Rightarrow \exists b \in B.\ (b \cdot |a_f|_{\mathrm{st}}) \in \mathcal{V}$

Figure 26.2: The partially stable resource algebra, additions in red.

(4) a core projection $|a|_{\mathrm{core}}$; (5) a validity predicate $\mathcal{V}$; (6) a stable projection $|\_|_{\mathrm{st}}$, and (7) an unstable projection $|\_|_{\mathrm{unst}}$.

As depicted in Fig. 26.2, they extend Iris's *unital resource algebras* from §4.2 $(\mathcal{A}, \cdot, \varepsilon, |\_|_{\mathrm{core}}, \mathcal{V})$ by the projections $|\_|_{\mathrm{st}}$ and $|\_|_{\mathrm{unst}}$. All existing rules still apply. We focus on the new rules governing the new projections.

The stable projection. A resource decomposes into its stable and unstable part (ra-decompose). The stable-projection $|a|_{\mathrm{st}}$ defines which part of the resource will be preserved by the corresponding updates ($\rightsquigarrow_{\mathrm{st}}$):

$$b \rightsquigarrow_{\mathrm{st}} b' \quad \textit{implies} \quad b \cdot a \rightsquigarrow_{\mathrm{st}} b' \cdot |a|_{\mathrm{st}}$$

The stable projection is idempotent (ra-stable-idemp), distributes over composition (ra-stable-distr), and preserves the core (ra-core-stable). Idempotence ensures that the frame modality is idempotent (frame-idemp in Fig. 26.1). Distributivity ensures that we can combine the separating conjunction of two frame modalities (frame-sep in Fig. 26.1). The preservation of the core ensures that persistent assertions are always frameable (frame-pers in Fig. 26.1).

For all unstable resource algebras, we have that $a \rightsquigarrow b$ *implies* $a \rightsquigarrow_{\mathrm{st}} b$, since $|a|_{\mathrm{st}} \preccurlyeq a$ (via ra-decompose). Moreover, in resource algebras from standard Iris, all elements are stable (*i.e.*, $|a|_{\mathrm{st}} = a$) such that for them ($\rightsquigarrow_{\mathrm{st}}$) and ($\rightsquigarrow$) coincide. More specifically, one can trivially turn regular Iris unital resource algebras into partially stable resource algebra by picking $|a|_{\mathrm{st}} = a$ and $|a|_{\mathrm{unst}} = \varepsilon$.

The unstable projection. The unstable projection yields a duplicable part of a resource (ra-unstable-dupl), justifying the key rule unstable-dupl. It is idempotent (ra-unstable-idemp) and monotone (ra-unstable-mono). Idempotence ensures that the unstable modality is idempotent (unstable-idemp in Fig. 26.1).

fragment rules

FRAG-OP
$\circ(a \cdot b) = \circ a \cdot \circ b$

FRAG-CORE
$|\circ a|_{\mathrm{core}} = \circ |a|_{\mathrm{core}}$

FRAG-VAL
$\circ a \in \mathcal{V} \iff a \in \mathcal{V}_M$

FRAG-UNIT
$\circ \varepsilon_M = \varepsilon$

FRAG-INCL
$\circ a \preccurlyeq \circ b \iff a \preccurlyeq_M b$

FRAG-STABLE
$|\circ a|_{\mathrm{st}} = \circ |a|_{\mathrm{st}}$

FRAG-UNSTABLE
$|\circ a|_{\mathrm{unst}} = \circ |a|_{\mathrm{unst}}$

authoritative element rules

AUTH-VAL
$\bullet a \in \mathcal{V} \iff a \in \mathcal{V}_M \land \mathit{maximal}(a)$

AUTH-EXCL
$\bullet a \cdot \bullet b \in \mathcal{V} \iff \mathsf{False}$

AUTH-STABLE
$|\bullet a|_{\mathrm{st}} = \bullet a$

AUTH-UNSTABLE
$|\bullet a|_{\mathrm{unst}} = ◒ a$

unstable authoritative element rules

AUTH-UNSTABLE-VAL
$◒ a \in \mathcal{V} \iff a \in \mathcal{V}_M \land \mathit{maximal}(a)$

AUTH-UNSTABLE-AGREE
$◒ a \cdot ◒ b \in \mathcal{V} \Rightarrow a = b$

AUTH-UNSTABLE-AUTH-OP
$\bullet a = \bullet a \cdot ◒ a$

AUTH-UNSTABLE-OP
$◒ a = ◒ a \cdot ◒ a$

AUTH-UNSTABLE-UNSTABLE
$|◒ a|_{\mathrm{unst}} = ◒ a$

AUTH-UNSTABLE-STABLE
$|◒ a|_{\mathrm{st}} = \varepsilon$

interaction rules

BOTH-VALID
$\bullet a \cdot \circ b \in \mathcal{V} \iff b \preccurlyeq_M a \land a \in \mathcal{V}_M \land \mathit{maximal}(a)$

BOTH-UNSTABLE-VALID
$◒ a \cdot \circ b \in \mathcal{V} \iff b \preccurlyeq_M a \land a \in \mathcal{V}_M \land \mathit{maximal}(a)$

BOTH-UPDATE
$(a, b) \rightsquigarrow_{\mathsf{L}} (a', b') \Rightarrow \bullet a \cdot \circ b \rightsquigarrow \bullet a' \cdot \circ b'$

BOTH-UPDATE-STABLE
$(a, b) \rightsquigarrow^{\mathsf{L}}_{\mathrm{st}} (a', b') \Rightarrow \bullet a \cdot \circ b \rightsquigarrow_{\mathrm{st}} \bullet a' \cdot \circ b'$

Figure 26.3: The partially stable authoritative resource algebra *PSAuth*.

Monotonicity ensures (together with RA-UNSTABLE-DUPL) that the unstable modality distributes over separating conjunction (UNSTABLE-SEP in Fig. 26.1).

In addition, we impose two key axioms RA-UNSTABLE-FLIP and RA-UNSTABLE-EXTENSION to obtain the implication rule for unstable propositions (UNSTABLE-IMPL in Fig. 26.1). The axiom RA-UNSTABLE-FLIP allows one to flip the unstable projection inside an unstable projection. It is a weaker form of distributivity of the unstable projection (*i.e.,* weaker than $|a \cdot b|_{\mathrm{unst}} = |a|_{\mathrm{unst}} \cdot |b|_{\mathrm{unst}}$) that still suffices for the implication rule.[14] The axiom RA-UNSTABLE-EXTENSION ensures that the unstable part $|a|_{\mathrm{unst}}$ is a "complete snapshot" of a in the sense that there cannot be an element b that is valid with $|a|_{\mathrm{unst}}$ but whose unstable part $|b|_{\mathrm{unst}}$ is not valid with a. It arises because Iris's implication $P \Rightarrow Q$ is upclosed with respect to larger resources (see §4.3) and so to prove UNSTABLE-IMPL, we end up needing to extend the validity predicate for a resource. If the unstable projection provides a "complete snapshot" as given by RA-UNSTABLE-EXTENSION, one can show that the extension does not break validity.

[14] The $\mathit{Sil}(M)$ resource algebra that we will discuss in §26.3.1 satisfies this rule but does not satisfy distributivity.

26.3.1 Resource Algebra Combinators

We define the points-to assertions $\ell \mapsto_{\mathrm{u}} v$ and $\ell \mapsto_q v$ in §26.3.2. As usual for Iris (see §4.2.2), we factor their definition through several reusable combinators—including new ones for unstable resources. We discuss the combinators.

$$\mathsf{stab}_q(x) \in \mathcal{V} \Leftrightarrow q \leq 1 \qquad \mathsf{unst}(x) \in \mathcal{V} \qquad |\mathsf{stab}_q(x)|_{\mathrm{st}} = \mathsf{stab}_q(x) \qquad |\mathsf{unst}(x)|_{\mathrm{st}} = \varepsilon \qquad |\mathsf{stab}_q(x)|_{\mathrm{unst}} = \mathsf{unst}(x)$$

$$|\mathsf{unst}(x)|_{\mathrm{unst}} = \mathsf{unst}(x) \qquad \mathsf{stab}_{q_1}(x) \cdot \mathsf{stab}_{q_2}(x) = \mathsf{stab}_{q_1+q_2}(x) \qquad \mathsf{stab}_q(x) = \mathsf{stab}_q(x) \cdot \mathsf{unst}(x)$$

$$\mathsf{unst}(x) = \mathsf{unst}(x) \cdot \mathsf{unst}(x) \qquad \mathsf{stab}_{q_1}(x) \cdot \mathsf{stab}_{q_2}(y) \in \mathcal{V} \Leftrightarrow (q_1 + q_2 \leq 1 \land x = y)$$

$$\mathsf{unst}(x) \cdot \mathsf{unst}(y) \in \mathcal{V} \Rightarrow x = y \qquad \mathsf{stab}_1(x) \rightsquigarrow_{\mathrm{st}} \mathsf{stab}_1(y) \qquad \mathit{maximal}(\mathsf{stab}_1(x))$$

Figure 26.4: The resource algebra *ExFrac*(X).

Authoritative resource algebra. We define a version of the authoritative resource algebra *Auth*(M) (see §4.2.1) for partially stable resource algebras, written *PSAuth*(M), depicted in Fig. 26.3. Compared to the standard Iris authoritative resource algebra, this version has an additional, unstable element $◒a$, which is an unstable copy of the full element $\bullet a$ (*i.e.,* $\bullet a = \bullet a \cdot ◒a$; see AUTH-UNSTABLE-AUTH-OP). Moreover, it can be applied to unstable resource algebras (*i.e.,* M can have unstable elements), where it lifts the stable and unstable projections to fragments (see FRAG-STABLE and FRAG-UNSTABLE). To fulfill the property RA-UNSTABLE-EXTENSION, we require that full elements $\bullet a$ are maximal (see AUTH-VAL) in the sense that

$$\mathit{maximal}(a) \triangleq \forall b.\, b \in \mathcal{V}_M \Rightarrow |b|_{\mathrm{unst}} \preccurlyeq_M |a|_{\mathrm{unst}} \Rightarrow b \preccurlyeq_M a$$

For this resource algebra, we then obtain a new stable update rule (BOTH-UPDATE-STABLE). It mirrors the regular update rule for the authoritative resource algebra, but it uses a different notion of the local update, a *stable local update* $(a, b) \rightsquigarrow^{\mathsf{L}}_{\mathrm{st}} (a', b')$, which we define as

$$(a, b) \rightsquigarrow^{\mathsf{L}}_{\mathrm{st}} (a', b') \triangleq \forall a_f.\, a \in \mathcal{V}_M \land \mathit{maximal}(a) \land b \cdot a_f \preccurlyeq_M a \Rightarrow$$
$$a' \in \mathcal{V}_M \land \mathit{maximal}(a') \land b' \cdot |a_f|_{\mathrm{st}} \preccurlyeq_M a'$$

Exclusive with fractions. The exclusive resource algebra with fractions *ExFrac*$(X) \ni \varepsilon \mid \mathsf{stab}_q(x) \mid \mathsf{unst}(x)$ generalizes the resource algebra *Ex*$(\mathbb{N})$ by (1) enabling fractional ownership of the exclusive element and (2) *adding unstable elements.* We have two elements $\mathsf{stab}_q(x)$ (the counter part of $\mathsf{ex}(x)$) and $\mathsf{unst}(x)$ (the counter part of $\mathsf{tmp}(x)$). Their key rules are depicted in Fig. 26.4. The element $\mathsf{stab}_q(x)$ carries fractional ownership, it is always stable, and it is maximal for fraction 1. We can stably update it for fraction 1. The element $\mathsf{unst}(x)$ is an unstable, temporary copy.

Silhouette resource algebra. We define the *Sil*$(M) \ni \varepsilon \mid \mathsf{orig}(a) \mid \mathsf{sil}(a)$ resource algebra, which completes a standard unital resource algebra M to one with unstable resources.[15] The resource algebra is depicted in Fig. 26.5. Its two key elements are $\mathsf{orig}(a)$, which simply embeds elements $a \in M$, and $\mathsf{sil}(a)$, which is an unowned copy of a. When we have two unowned copies $\mathsf{sil}(a) \cdot \mathsf{sil}(b)$, we do not add them together. Instead, there must be some c that includes a and b. (This is useful for, *e.g.,* the unstable points-to to know that $\ell \mapsto_{\mathrm{u}} v * \ell \mapsto_{\mathrm{u}} w$ ensures $v = w$.) The $\mathsf{orig}(a)$ elements are maximal, so we can use them in *PSAuth* constructions.

[15] This resource algebra supports only non-step-indexed (typically called "discrete") resource algebras as instantiations of M.

$$\mathsf{orig}(a) \in \mathcal{V} \Leftrightarrow a \in \mathcal{V} \qquad \mathsf{sil}(a) \in \mathcal{V} \Leftrightarrow a \in \mathcal{V} \qquad |\mathsf{orig}(a)|_{\mathsf{st}} = \mathsf{orig}(a) \qquad |\mathsf{sil}(a)|_{\mathsf{st}} = \varepsilon \qquad |\mathsf{orig}(a)|_{\mathsf{unst}} = \mathsf{sil}(a)$$

$$|\mathsf{sil}(a)|_{\mathsf{unst}} = \mathsf{sil}(a) \qquad \mathsf{orig}(a) = \mathsf{orig}(a) \cdot \mathsf{sil}(a) \qquad \mathsf{orig}(a \cdot b) = \mathsf{orig}(a) \cdot \mathsf{orig}(b) \qquad \mathsf{sil}(a) = \mathsf{sil}(a) \cdot \mathsf{sil}(a)$$

$$\mathsf{sil}(a) \cdot \mathsf{sil}(b) \in \mathcal{V} \Leftrightarrow (\exists c.\, c \in \mathcal{V} \wedge a \preccurlyeq c \wedge b \preccurlyeq c) \qquad a \preccurlyeq b \Leftrightarrow \mathsf{sil}(a) \preccurlyeq \mathsf{sil}(b) \qquad (a \rightsquigarrow b) \Rightarrow (\mathsf{orig}(a) \rightsquigarrow_{\mathsf{st}} \mathsf{orig}(b))$$

$$((a, b) \rightsquigarrow_{\mathsf{L}} (a', b')) \Rightarrow ((\mathsf{orig}(a), \mathsf{orig}(b)) \rightsquigarrow^{\mathsf{L}}_{\mathsf{st}} (\mathsf{orig}(a'), \mathsf{orig}(b'))) \qquad \mathit{maximal}(\mathsf{orig}(a))$$

Figure 26.5: The resource algebra *Sil*(M).

FRAME-PTS
$$\ell \mapsto_q v \vdash \boxplus \ell \mapsto_q v$$

FRAME-HEAP
$$\mathsf{heap}(h) \vdash \boxplus \mathsf{heap}(h)$$

UNSTABLE-PTS
$$\ell \mapsto_{\mathsf{u}} v \vdash \divideontimes\, \ell \mapsto_{\mathsf{u}} v$$

UNSTABLE-HEAP
$$\mathsf{heap}_{\mathsf{u}}(h) \vdash \divideontimes\, \mathsf{heap}_{\mathsf{u}}(h)$$

GET-PTS-UNSTABLE
$$\ell \mapsto_q v \vdash \ell \mapsto_{\mathsf{u}} v$$

GET-HEAP-UNSTABLE
$$\mathsf{heap}(h) \vdash \mathsf{heap}_{\mathsf{u}}(h)$$

OWNED-AGREE
$$\ell \mapsto_q v * \ell \mapsto_{q'} w \vdash q + q' \leq 1 * v = w$$

UNSTABLE-AGREE
$$\ell \mapsto_{\mathsf{u}} v * \ell \mapsto_{\mathsf{u}} w \vdash v = w$$

HEAP-AGREE
$$\mathsf{heap}_{\mathsf{u}}(h) * \mathsf{heap}_{\mathsf{u}}(h') \vdash h = h'$$

PTS-SPLIT
$$\ell \mapsto_{q+q'} v \dashv\vdash \ell \mapsto_q v * \ell \mapsto_{q'} v$$

UNSTABLE-LOOKUP
$$\ell \mapsto_{\mathsf{u}} v * \mathsf{heap}_{\mathsf{u}}(h) \vdash h(\ell) = v$$

HEAP-LOOKUP
$$\ell \mapsto_q v * \mathsf{heap}(h) \vdash h(\ell) = v$$

PTS-UPDATE
$$\ell \mapsto_1 v * \mathsf{heap}(h) \vdash \Rrightarrow_{\mathsf{st}} \ell \mapsto_1 w * \mathsf{heap}(h[\ell := w])$$

PTS-ALLOC
$$\ell \notin \mathsf{dom}\, h * \mathsf{heap}(h) \vdash \Rrightarrow_{\mathsf{st}} \ell \mapsto_1 v * \mathsf{heap}(h[\ell := v])$$

Figure 26.6: The theory of heaps obtained by choosing the resource algebra *PSHeap*

26.3.2 Modeling the Unstable Points-To

Equipped with these resource algebra combinators, we can discuss the model of the unstable points-to assertion $\ell \mapsto_{\mathsf{u}} v$. In this version, we obtain the following resource assertions:

$$P, Q ::= \cdots \mid \ell \mapsto_q v \mid \ell \mapsto_{\mathsf{u}} v \mid \mathsf{heap}(h) \mid \mathsf{heap}_{\mathsf{u}}(h)$$

They extend the heap construction from §4.2.2 with two new, unstable assertions: $\mathsf{heap}_{\mathsf{u}}(h)$, an unstable temporary copy of $\mathsf{heap}(h)$ and $\ell \mapsto_{\mathsf{u}} v$, an unstable temporary copy of the points-to. We first discuss their resulting ghost theory, depicted in Fig. 26.6, and then we return to how they are defined in terms of resource algebra combinators.

As usual, points-tos must agree on their values (OWNED-AGREE), we can split- and combine points-to assertions as needed (PTS-SPLIT), and a points-to assertion for ℓ determines the value of ℓ in h (HEAP-LOOKUP). We can update an entry in the heap with a *stable update* (PTS-UPDATE) and allocate a new entry (PTS-ALLOC). The points-to for $q > 0$ and the heap are frameable (FRAME-PTS and FRAME-HEAP), and their unstable counterparts are unstable (UNSTABLE-PTS and UNSTABLE-HEAP). We can always get an unstable copy from the owned versions (GET-PTS-UNSTABLE and GET-HEAP-UNSTABLE), and the unstable versions agree: the unstable points-tos agree on the value in the heap (UNSTABLE-AGREE), two unstable heaps agree on the entire heap (heap-agree), and we can look up a location in the unstable heap (unstable-lookup).

To obtain this resource theory, we pick the resource algebra *PSHeap* $\triangleq$ $\mathit{PSAuth}(\mathit{Loc} \xrightarrow{\text{fin}} \mathit{ExFrac}(\mathit{Val}))$.

$$\ell \mapsto_q v \triangleq \boxed{\circ[\ell \mapsto \mathsf{stab}_q(v)]}^{\gamma_{\mathrm{heap}}} \qquad \ell \mapsto_{\mathsf{u}} v \triangleq \boxed{\circ[\ell \mapsto \mathsf{unst}(v)]}^{\gamma_{\mathrm{heap}}}$$

$$\mathsf{heap}(h) \triangleq \boxed{\bullet[\ell \mapsto \mathsf{stab}_1(v) \mid \ell \mapsto v \in h]}^{\gamma_{\mathrm{heap}}}$$

$$\mathsf{heap}_{\mathsf{u}}(h) \triangleq \boxed{◒[\ell \mapsto \mathsf{stab}_1(v) \mid \ell \mapsto v \in h]}^{\gamma_{\mathrm{heap}}}$$

This resource algebra is a variation of a standard technique for constructing a resource algebra for heaps in Iris (from §4.2.2). For the regular points-to $\ell \mapsto_q v$, we use a fragment $\circ$ containing a singleton map that maps ℓ to the *stable resource* $\mathsf{stab}_q(v)$ (from the resource algebra in Fig. 26.4). Analogously, for the unstable points-to $\ell \mapsto_{\mathsf{u}} v$, we use a fragment $\circ$ containing a singleton map that maps ℓ to the *unstable resource* $\mathsf{unst}(v)$ (from the resource algebra in Fig. 26.4). For the full heap $\mathsf{heap}(h)$, we use the authoritative element $\bullet$ containing the entire heap (where every element has been allocated with fraction 1). For the unstable copy $\mathsf{heap}_{\mathsf{u}}(h)$, we use the new unstable authoritative element ◒ of our *PSAuth* resource algebra.

Supporting discardable fractions. Technically, in Rocq, we do not use exactly this resource algebra construction, but a generalization of it. We define a version that supports Iris's discardable fractions.[16] Its construction is more involved than what is discussed here, but not substantially more interesting. For it, we choose the partially stable resource algebra $\mathit{Heap}(\mathit{Loc}, \mathit{Val}) \triangleq \mathit{PSView}((\mathit{Loc} \xrightarrow{\text{fin}} \mathit{Val}), \mathit{Sil}(\mathit{Loc} \xrightarrow{\text{fin}} \mathit{DFrac} \times \mathit{Ag}(\mathit{Val})))$, where *PSView* is a version of Iris's view resource algebra *extended with unstable elements* (generalizing the *PSAuth*-resource algebra from §26.3.1), *Sil* is the silhouette resource algebra (from §26.3.1), and *DFrac* is Iris's resource algebra for discardable fractions [VB21, §9]. The details of this construction can be found in the accompanying Rocq development.[17]

[16] Vindum and Birkedal, "Contextual refinement of the Michael-Scott queue (proof pearl)", 2021 [VB21].

[17] Spies et al., *Destabilizing Iris (Rocq development and appendix)*, 2025 [Spi+25b].

Chapter 27

The Program Logic

Having generalized the resource model of Iris (§26), let us now use it to obtain a program logic. We first focus on the concrete program logic for λ_{dyn} (§27.1), and then we show how one can obtain such program logics via our adaptation of Iris's language-generic *weakest precondition* (§27.2).

27.1 The λ_{dyn} Program Logic

A selection of the rules of the λ_{dyn}-program logic is depicted in Fig. 27.1. Most of the rules are completely standard Hoare triple versions of the rules from Part I (*e.g.*, the structural rules such as hoare-conseq or hoare-bind).

Heap-dependent expression assertions. The main effect of supporting HDEAs are the rules hoare-frame and hoare-eval, which we have already encountered in §25. A second, notable difference is that for loading from a location (hoare-load), the unstable points-to assertion suffices.[1] In addition, several rules—where the expression e does not modify the current state—have a built-in instance of framing such as hoare-val, hoare-load, and hoare-ref.

[1] This rule is a generalization of the regular rule for loading, since we can always get an unstable points-to from a regular points-to with get-pts-unstable in Fig. 26.6.

Let us now discuss the frame modalities in hoare-frame. As we have seen in §25.2, R must be under a frame modality, since otherwise one could frame unstable assertions around an expression that invalidates these assertions. The frame modalities in the postcondition only strengthen hoare-frame and can always be eliminated with frame-elim (in Fig. 25.1). To understand why P must be under a frame modality, let us first look at hoare-pure-step and hoare-fork. These rules look like the standard Hoare rules and *do not* contain any frame modalities. This might be surprising since a concurrent thread could potentially invalidate their preconditions (*e.g.*, an unstable points-to assertion $\ell \mapsto_{\mathsf{u}} v$ for a load with hoare-load). The reason that this cannot happen (and thus no frame modalities are required in these rules) is that Hoare triples implicitly maintain that their pre- and postconditions are always frameable (see the definition of $\{P\}\ e\ \{v.\, Q(v)\}$ in §27.2). The price we have to pay for this is that we need to prove that P in hoare-frame is frameable such that we can use it as the precondition of e in the premise.

Step-indexing, invariants, and resources. Let us now focus on several features that make Iris particularly expressive: *step-indexing* (§3.2), *later credits* (§12), *impredicative invariants* (§3.4), and *custom resources* (§3.6).

We start with step-indexing and later credits. Supporting step-indexing is straightforward. As before (see §4.1), we use the *later modality* $\triangleright P$ to define

$$\textsc{hoare-val}\quad \{P(v)\}\ v\ \{w.\ P(w)\}_{\mathcal{E}}$$

$$\textsc{hoare-conseq}\quad \frac{P \vdash P' \qquad \{P'\}\ e\ \{v.\ Q'(v)\}_{\mathcal{E}} \qquad \forall v.\ Q'(v) \vdash Q(v)}{\{P\}\ e\ \{v.\ Q(v)\}_{\mathcal{E}}}$$

$$\textsc{hoare-exists}\quad \frac{\forall x : X.\ \{P(x)\}\ e\ \{v.\ Q(v)\}_{\mathcal{E}}}{\{\exists x : X.\ P(x)\}\ e\ \{v.\ Q(v)\}_{\mathcal{E}}}$$

$$\textsc{hoare-pure}\quad \frac{P \vdash \phi \qquad \phi \Rightarrow \{P\}\ e\ \{v.\ Q(v)\}_{\mathcal{E}}}{\{P\}\ e\ \{v.\ Q(v)\}_{\mathcal{E}}}$$

$$\textsc{hoare-bind}\quad \frac{\{P\}\ e\ \{v.\ Q(v)\}_{\mathcal{E}} \qquad \forall v.\ \{Q(v)\}\ K[v]\ \{w.\ R(w)\}_{\mathcal{E}}}{\{P\}\ K[e]\ \{w.\ R(w)\}_{\mathcal{E}}}$$

$$\textsc{hoare-frame}\quad \frac{\{P\}\ e\ \{v.\ Q(v)\}_{\mathcal{E}}}{\{\boxplus P * \boxplus R\}\ e\ \{v.\ \boxplus Q(v) * \boxplus R\}_{\mathcal{E}}}$$

$$\textsc{hoare-löb}\quad \frac{\left\{P * \triangleright \{P\}\ e\ \{w.\ Q(w)\}_{\mathcal{E}}\right\}\ e\ \{v.\ Q(v)\}_{\mathcal{E}}}{\{P\}\ e\ \{v.\ Q(v)\}_{\mathcal{E}}}$$

$$\textsc{hoare-pure-step}\quad \frac{e \rightarrow_{\mathsf{pure}} e' \qquad \{P\}\ e'\ \{v.\ Q(v)\}_{\mathcal{E}}}{\{\triangleright P\}\ e\ \{v.\ Q(v)\}_{\mathcal{E}}}$$

$$\textsc{hoare-load}\quad \{P * \ell \mapsto_u v\}\ !\,\ell\ \{w.\ P * v = w\}_{\mathcal{E}}$$

$$\textsc{hoare-store}\quad \{\ell \mapsto_1 v\}\ \ell := w\ \{\_.\ \ell \mapsto_1 w\}_{\mathcal{E}}$$

$$\textsc{hoare-ref}\quad \{P\}\ \mathsf{ref}(v)\ \{w.\ P * \exists \ell.\ w = \ell * \ell \mapsto_1 v\}_{\mathcal{E}}$$

$$\textsc{hoare-fork}\quad \frac{\{P\}\ e\ \{\_.\ \mathsf{True}\}_{\top}}{\{P\}\ \mathsf{fork}\ \{e\}\ \{\_.\ \mathsf{True}\}_{\mathcal{E}}}$$

$$\textsc{hoare-supd}\quad \frac{\{P\}\ e\ \{v.\ Q(v)\}_{\mathcal{E}}}{\{\Rrightarrow_{\mathsf{st}} \boxplus P\}\ e\ \{v.\ Q(v)\}_{\mathcal{E}}}$$

$$\textsc{hoare-inv-open}\quad \frac{\{\triangleright R * P\}\ e\ \{v.\ \triangleright R * \boxplus Q(v)\}_{\mathcal{E}\setminus\mathcal{N}} \qquad R \vdash \boxplus R \qquad \mathsf{atomic}(e) \qquad \mathcal{N} \subseteq \mathcal{E}}{\left\{P * \boxed{R}^{\mathcal{N}}\right\}\ e\ \left\{v.\ Q(v)\right\}_{\mathcal{E}}}$$

$$\textsc{hoare-free}\quad \{\ell \mapsto_1 v\}\ \mathsf{free}(\ell)\ \{\_.\ \mathsf{True}\}_{\mathcal{E}}$$

$$\textsc{hoare-eval}\quad \{P * e \Downarrow \_\}\ e\ \{v.\ P * e \Downarrow v\}_{\mathcal{E}}$$

$$\textsc{hoare-faa}\quad \{\ell \mapsto_1 n_1\}\ \mathsf{FAA}(\ell, n_2)\ \{v.\ v = n_1 * \ell \mapsto_1 (n_1 + n_2)\}_{\mathcal{E}}$$

$$\textsc{hoare-cas-suc}\quad \frac{v_1 = v_2 \qquad v_1, v_2\ \mathit{comparable}}{\{\ell \mapsto_1 v_1\}\ \mathsf{CAS}(\ell, v_2, w)\ \{u.\ u = \mathsf{true} * \ell \mapsto_1 w\}_{\mathcal{E}}}$$

$$\textsc{hoare-cas-fail}\quad \frac{v_1 \neq v_2 \qquad v_1, v_2\ \mathit{comparable}}{\left\{\ell \mapsto_q v_1\right\}\ \mathsf{CAS}(\ell, v_2, w)\ \left\{u.\ u = \mathsf{false} * \ell \mapsto_q v_1\right\}_{\mathcal{E}}}$$

Figure 27.1: A selection of rules for the program logic of λ_{dyn}.

an underlying weakest precondition **wp** $e\ \{v.\ Q(v)\}$ (in §27.2). Thus, one can eliminate a later modality with every program step (*e.g.*, see hoare-pure-step; omitted in the other rules), which supports recursive reasoning such as Löb induction (*e.g.*, with hoare-löb). Moreover, while we omit them from the definition in §27.2 for simplicity (and the rules in Fig. 27.1), Daenerys also supports *later credits* (from Part III). The key to support them is to define a version of the later elimination update $\Rrightarrow_{\mathsf{le}}$ (from §14.2) with the stable update $\Rrightarrow_{\mathsf{st}}$ in place of the regular update $\Rrightarrow$. Later credits are frameable (*i.e.*, $\pounds n \vdash \boxplus \pounds n$), and of course the frame rule for the later elimination update $\Rrightarrow_{\mathsf{le}}$ must be augmented with frame modalities (*i.e.*, resulting in the analogue of supd-frame in Fig. 26.1).[2]

Recall that Iris shares resources between threads using impredicative invariants $\boxed{P}^{\mathcal{N}}$ (§3.4). To open an impredicative invariant, we use the rule hoare-inv-open in Daenerys. It is almost the same as in standard Iris (see hoare-inv-open in §3.4.1). The only user-facing effect that the presence of *unstable resources* in the underlying model of Daenerys has on this rule is that (1) we must prove that the invariant R is frameable (*i.e.*, we prove $R \vdash \boxplus R$) and (2) we may use only stable resources to prove the postcondition Q. The former is important because an invariant storing an unstable assertion without the ownership to stabilize it could be broken when the program invalidates the unstable assertion. The latter is important since the postcondition must not be allowed to take unstable assertions out of the invariant without the ownership that stabilizes them.

[2] For this construction to largely "just work", it is vital that the stable update retains almost the same compositionality as the regular update.

To support resources—and in particular user-defined ghost state (§3.6)—Daenerys provides the rule HOARE-SUPD. It allows one to perform a stable update $\Rrightarrow_{\mathsf{st}}$ in the precondition. (It is the counterpart of HOARE-UPD from §3.6.) Since the regular update entails the stable update (*i.e.*, $\Rrightarrow P \vdash \Rrightarrow_{\mathsf{st}} P$ from Fig. 26.1), this rule means that we can still reuse the standard Iris ghost state constructions from the prior chapters. For example, we can still use monotonically growing natural numbers (from §3.6) and ghost variables (from §4.2.2). To maintain that the precondition is frameable (as discussed above), the update rule HOARE-SUPD requires the resulting P to be under a frame modality.

27.2 The Language-Generic Weakest Precondition

Let us now turn to how we define the Hoare triples in §27.1. In typical Iris fashion, we do so via a weakest precondition $\mathbf{wp}\ e\ \{v.\ Q(v)\}$ (adapted to Daenerys).[3] For Hoare triples, we define:

$$\{P\}\ e\ \{v.\ Q(v)\} \triangleq \Box(\boxplus P \twoheadrightarrow \mathbf{wp}\ e\ \{v.\ \boxplus Q(v)\})$$

That is, as usual, $\{P\}\ e\ \{v.\ Q(v)\}$ holds if we can show that the precondition P implies the weakest precondition of e for post Q. The two frame modalities maintain implicitly that the precondition P and postcondition Q are frameable.

The weakest precondition is then defined in a language-generic fashion over a small-step relation $(e, h) \rightsquigarrow (e', h', es)$ as follows:

$$\begin{aligned}
&\mathbf{wp}\ v\ \{w.\ Q(w)\} \triangleq \forall h.\ \mathrm{SI}(h) \twoheadrightarrow \Rrightarrow_{\mathsf{st}} \mathrm{SI}(h) * Q(v) \\
&\mathbf{wp}\ e\ \{w.\ Q(w)\} \triangleq \forall h.\ \mathrm{SI}(h) \twoheadrightarrow \Rrightarrow_{\mathsf{st}} \mathsf{progress}(e, h) \qquad \text{if } e \notin Val \\
&\quad * \ \forall e', h', es.\ (e, h) \rightsquigarrow (e', h', es) \twoheadrightarrow \triangleright \Rrightarrow_{\mathsf{st}} \\
&\qquad\qquad (\mathrm{SI}(h') * \boxplus \mathbf{wp}\ e'\ \{w.\ Q(w)\} * \mathop{\Large *}_{e'' \in es} \boxplus \mathbf{wp}\ e''\ \{\_.\ \mathsf{True}\})
\end{aligned}$$

This definition is a variation of the concurrent weakest precondition from §4.1. Here, in the value case, we additionally assume the state interpretation $\mathrm{SI}(h)$,[4] return it, and then as usual prove the postcondition Q after an update.[5] In the case where e is not a value, we as usual assume the state interpretation and prove that e can make progress in the current heap. Then, we show that for any successor expression e', heap h', and forked-off threads es, we can re-establish the state interpretation and prove weakest preconditions for e' and the forked-off threads e''.

In this definition, we reap the fruits of our more general model (in §26). We can support HDEAs simply by using the new modalities of Daenerys (highlighted in red): we use stable updates $\Rrightarrow_{\mathsf{st}}$ in the places where Iris would traditionally use its frame preserving update $\Rrightarrow P$ (see §27.2). Moreover, we add the frame modality $\boxplus$ for the successor expression e' and the forked-off threads, because in a concurrent setting, they can be executed concurrently, so we must ensure that, *e.g.*, the verification of e' does not rely on any unstable knowledge that could be invalidated by another thread.

Adequacy. For this version of the weakest precondition, we can then reprove the adequacy theorem of Iris. Instantiated for λ_{dyn},[6] we obtain the result:

Theorem 79 (Adequacy). *If* $\vdash \{\mathsf{True}\}\ e\ \{v.\ \phi(v)\}$, *then e is safe to execute in any heap h and all possible return values v satisfy the pure postcondition $\phi(v)$.*

[3] To focus on the key parts of this definition (and the changes over Iris), we stay at the same level of abstraction here as in §4.1. Concretely, we omit the masks and fancy updates (see §4.5.1) that are needed to support invariants.

[4] Recall from §4.1 that the state interpretation ties the heap h in the weakest precondition to the points-to resources. For λ_{dyn}, we pick $\mathrm{SI}(h) \triangleq \mathsf{heap}(h)$ for the heaps from Fig. 26.6 to tie the heap h to $\ell \mapsto_q v$ and also $\ell \mapsto_{\mathsf{u}} v$.

[5] We have seen a similar pattern for the weakest precondition of Transfinite Iris in §7.3.1. Assuming the state interpretation in the value case gives one access to the current heap in both cases.

[6] In Rocq, we have first established a language generic version of the adequacy theorem (where one gets to choose the state interpretation) and have then instantiated it for λ_{dyn}. As in the prior chapters, we focus here on the concrete instantiation for λ_{dyn}.

Chapter 28
Almost-Pure Assertions

Recall the fragment of *almost-pure assertions* $F, G : hProp$ (from §25.3), containing pure assertions, evaluation, the unstable points-to assertion, and standard logical connectives, including impredicative quantification (*i.e.,* x can also range over *hProp*-assertions):

$$F, G : hProp ::= \phi \mid e \Downarrow v \mid \ell \mapsto_{\mathsf{u}} v \mid F \wedge G \mid F \vee G \mid F \Rightarrow G \mid \exists x.\, F\, x \mid \forall x.\, F\, x \mid \cdots$$

In this chapter, we use it to develop two automatable proof techniques for core aspects of Daenerys: First, we develop a semantic type system $\Gamma \vDash e : \tau$ (§28.1) to streamline stabilizing $e \Downarrow v$ (*i.e.,* to move it into a frame modality $\boxplus$; see §25.2). Then, we develop a correspondence between first-order logic and *hProp*-assertions (§28.2) to automate reasoning about program expressions using SMT solvers (see §25.3). For the correspondence, we will reuse the type system: it connects program functions in λ_{dyn} and first-order logic functions.

Defining almost pure assertions. But before we dive into how almost-pure assertions help us to improve automation, let us first clarify how they are defined. Instead of an inductive characterization as suggested above, we define them extensionally as follows:[1]

$$hProp \triangleq \left\{ \begin{pmatrix} P \in iProp, \\ \phi \in \mathbb{P}(Heap) \end{pmatrix} \middle| \begin{array}{c} P \vdash \circledast P, \textit{ and} \\ \forall h.\, \mathsf{heap}_{\mathsf{u}}(h) * (\ast_{\ell \mapsto v \in h}\, \ell \mapsto_{\mathsf{u}} v) \vdash \phi(h) \Leftrightarrow P \end{array} \right\}$$

[1] The point of the extensional characterization is that it allows us to support impredicative quantification in Rocq (*i.e., hProp* assertions can quantify over *hProp* assertions). Impredicative quantification makes the fragment more expressive. Specifically, it allows one to define least- and greatest fixpoints inside of *hProp*.

An almost-pure assertion $F = (P_F, \phi_F) \in hProp$ is a pair of an Iris proposition P_F and a predicate over heaps ϕ_F. Together, they satisfy two properties: First, the Iris proposition P_F is unstable. Second, the predicate ϕ_F characterizes the proposition P_F when the full heap h is known (*i.e.,* under the assumption $\mathsf{heap}_{\mathsf{u}}(h) * (\ast_{\ell \mapsto v \in h}\, \ell \mapsto_{\mathsf{u}} v)$). We write $\lfloor F \rfloor(h)$ for $\phi_F(h)$, and we treat F as if it was an Iris assertion instead of writing P_F (*e.g.,* in Lemma 80 below).

By construction, almost-pure propositions satisfy the following two lemmas:

Lemma 80. *$F \vdash \circledast F$, and hence, $F \wedge P \dashv\vdash F * P$ and $F \Rightarrow P \dashv\vdash F \mathrel{-\!\!*} P$*

Proof. The first part follows directly from the definition. The remaining two parts then follow from the properties of the unstable modality (UNSTABLE-DUPL in Fig. 26.1). □

Lemma 81. *For any heap h, $\mathsf{heap}_{\mathsf{u}}(h) * (\ast_{\ell \mapsto v \in h}\, \ell \mapsto_{\mathsf{u}} v) \vdash (F \Leftrightarrow \lfloor F \rfloor(h))$.*

Proof. Follows directly from the definition of *hProp*. □

$$\lfloor \phi \rfloor(h) \triangleq \phi \qquad \lfloor e \Downarrow v \rfloor(h) \triangleq (e, h) \rightsquigarrow^*_{\text{det}} (v, h) \qquad \lfloor \ell \mapsto_{\mathsf{u}} v \rfloor(h) \triangleq h(\ell) = v$$

$$\lfloor F \land G \rfloor(h) \triangleq \lfloor F \rfloor(h) \land \lfloor G \rfloor(h) \qquad \lfloor F \lor G \rfloor(h) \triangleq \lfloor F \rfloor(h) \lor \lfloor G \rfloor(h)$$

$$\lfloor F \Rightarrow G \rfloor(h) \triangleq \lfloor F \rfloor(h) \Rightarrow \lfloor G \rfloor(h) \qquad \lfloor \forall x.\, F\, x \rfloor(h) \triangleq \forall x.\, \lfloor F\, x \rfloor(h)$$

$$\lfloor \exists x.\, F\, x \rfloor(h) \triangleq \exists x.\, \lfloor F\, x \rfloor(h)$$

Figure 28.1: The mapping $\lfloor \_ \rfloor(\_) : \mathit{hProp} \rightarrow \mathit{Heap} \rightarrow \mathit{Prop}$ from *hProp*-assertions F to pure assertions $\lfloor F \rfloor(h)$.

The point of the first property (*i.e.,* Lemma 80) is that when we prove an *hProp*-assertion F (*e.g.,* an evaluation $e \Downarrow v$, a relationship between HDEAs, or even a typing $\Gamma \vDash e : \tau$ from §28.1), we can keep all of our ownership. That is, if we can prove $P \vdash F$, then from Lemma 80 it follows that $P \vdash F * P$.

The point of the second property (*i.e.,* Lemma 81) is that it allows us to connect almost-pure assertions inside of Iris to pure assertions at the meta level, which we rely on for connecting to first-order logic in §28.2. The property guarantees that, under the assumption $\mathsf{heap}_{\mathsf{u}}(h) * (\ast_{\ell \mapsto v \in h}\, \ell \mapsto_{\mathsf{u}} v)$, the Iris proposition P_F is equivalent (inside Iris) to the pure assertion $\phi_F(h)$. The assumption $\mathsf{heap}_{\mathsf{u}}(h) * (\ast_{\ell \mapsto v \in h}\, \ell \mapsto_{\mathsf{u}} v)$ fully determines the contents of the heap in the underlying Iris resource.

Almost-pure connectives. As shown above, Lemma 80 and Lemma 81 follow directly from the definition of *hProp*. Implicitly, the proof burden is put onto the individual connectives (*i.e.,* $e \Downarrow v$, $\ell \mapsto_{\mathsf{u}} v$, $F \land G$, *etc.*). The definition of their heap predicate $\lfloor \_ \rfloor(\_)$ is given in Fig. 28.1. To show that they are all almost-pure assertions, we must show that they satisfy the two defining properties of *hProp*. For most connectives, this is straightforward. The two interesting cases are evaluation $e \Downarrow v$ and implication $F \Rightarrow G$.

For evaluation $e \Downarrow v$, the property corresponding to Lemma 80 holds, because $e \Downarrow v$ is defined using only unstable resources (see EVAL-DEF in §26.1). For the property corresponding to Lemma 81, we must show:

$$\mathsf{heap}_{\mathsf{u}}(h) * (\ast_{\ell \mapsto v \in h}\, \ell \mapsto_{\mathsf{u}} v) \vdash (e \Downarrow v \Leftrightarrow \lfloor e \Downarrow v \rfloor(h))$$

Considering that $e \Downarrow v \triangleq \exists h'.\, (e, h') \rightsquigarrow^*_{\text{det}} (v, h') * (\ast_{\ell \mapsto w \in h'}\, \ell \mapsto_{\mathsf{u}} w)$, the forward direction "$\Rightarrow$" follows from $\mathsf{heap}_{\mathsf{u}}(h)$ including all the unstable points-to assertions concealed under the definition of $e \Downarrow v$ (UNSTABLE-LOOKUP in Fig. 26.6). The backward direction "$\Leftarrow$" follows, because we have unstable points-to assertions $\ell \mapsto_{\mathsf{u}} v$ in our assumption for every entry in h.

For implication $F \Rightarrow G$, the propety corresponding to Lemma 80 follows using the implication property of the unstable modality, UNSTABLE-IMPL in Fig. 26.1. The property corresonding to Lemma 81 follows structurally. Here, it is important that we use an equivalence in Lemma 81 and not an implication, because otherwise we could not apply the property on the left side of an implication.

28.1 The Semantic Type System

Having defined almost-pure assertions, let us now use them. Recall (from §25.2) that to stabilize $e \Downarrow v$, we must frame enough ownership P alongside it to ensure that the result of e does not change (see FRAME-EVALS in Fig. 25.1). Formally, the

$$\textsc{type-lam}\ \frac{\Gamma, x : \tau \vDash e : \sigma}{\Gamma \vDash \lambda x.\, e : \tau \to \sigma} \qquad \textsc{type-app}\ \frac{\Gamma \vDash e_1 : \tau \to \sigma \qquad \Gamma \vDash e_2 : \tau}{\Gamma \vDash e_1\, e_2 : \sigma} \qquad \textsc{type-var}\ \frac{x : \tau \in \Gamma}{\Gamma \vDash x : \tau} \qquad \textsc{type-val}\ \frac{v \in \mathcal{V}[\![\tau]\!]}{\Gamma \vDash v : \tau} \qquad \textsc{type-pair}\ \frac{\Gamma \vDash e_1 : \tau_1 \qquad \Gamma \vDash e_2 : \tau_2}{\Gamma \vDash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\textsc{type-fst}\ \frac{\Gamma \vDash e : \tau_1 \times \tau_2}{\Gamma \vDash \pi_1\, e : \tau_1} \qquad \textsc{type-snd}\ \frac{\Gamma \vDash e : \tau_1 \times \tau_2}{\Gamma \vDash \pi_2\, e : \tau_2} \qquad \textsc{type-if}\ \frac{\Gamma \vDash e : \mathsf{bool} \qquad \Gamma \vDash e_1 : \tau \qquad \Gamma \vDash e_2 : \tau}{\Gamma \vDash \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : \tau} \qquad \textsc{type-deref}\ \frac{\Gamma \vDash e : \mathsf{ref}\, \tau}{\Gamma \vDash\, !\, e : \tau}$$

$$\textsc{type-int-add}\ \frac{\Gamma \vDash e_1 : \mathsf{int} \qquad \Gamma \vDash e_2 : \mathsf{int}}{\Gamma \vDash e_1 + e_2 : \mathsf{int}} \qquad \textsc{type-bitvec-xor}\ \frac{\Gamma \vDash e_1 : \mathsf{bv}\, n \qquad \Gamma \vDash e_2 : \mathsf{bv}\, n}{\Gamma \vDash e_1\ \mathsf{xor}\ e_2 : \mathsf{bv}\, n} \qquad \textsc{type-bitvec-eq}\ \frac{\Gamma \vDash e_1 : \mathsf{bv}\, n \qquad \Gamma \vDash e_2 : \mathsf{bv}\, n}{\Gamma \vDash e_1 == e_2 : \mathsf{bool}}$$

Figure 28.2: A selection of typing rules for λ_{dyn}-expressions.

$$\mathcal{V}[\![\mathsf{unit}]\!] \triangleq \{()\} \qquad \mathcal{V}[\![\mathsf{bool}]\!] \triangleq \{\mathsf{true}, \mathsf{false}\} \qquad \mathcal{V}[\![\mathsf{int}]\!] \triangleq \{n \mid n \in \mathbb{Z}\} \qquad \mathcal{V}[\![\mathsf{bv}\, n]\!] \triangleq \{u \mid 0 \leq u < 2^n\}$$

$$\mathcal{V}[\![\mathsf{option}\, \tau]\!] \triangleq \{\mathsf{None}\} \cup \{\mathsf{Some}\, v \mid v \in \mathcal{V}[\![\tau]\!]\} \qquad \mathcal{V}[\![\tau + \sigma]\!] \triangleq \left\{\mathsf{inj}_1\, v \mid v \in \mathcal{V}[\![\tau]\!]\right\} \cup \left\{\mathsf{inj}_2\, w \mid w \in \mathcal{V}[\![\sigma]\!]\right\}$$

$$\mathcal{V}[\![\tau \times \sigma]\!] \triangleq \{(v, w) \mid v \in \mathcal{V}[\![\tau]\!] \wedge w \in \mathcal{V}[\![\sigma]\!]\} \qquad \mathcal{V}[\![\mathsf{ref}\, \tau]\!] \triangleq \{\ell \mid \exists v.\, \ell \mapsto_{\mathsf{u}} v \wedge v \in \mathcal{V}[\![\tau]\!]\}$$

$$\mathcal{V}[\![\mathsf{buf}\, \tau]\!] \triangleq \{b \mid \exists \vec{v}.\, b \mapsto_{\mathsf{u}} \vec{v} \wedge \forall w \in \vec{v}.\, w \in \mathcal{V}[\![\tau]\!]\} \qquad \mathcal{V}[\![\tau \to \sigma]\!] \triangleq \{f \mid \forall v.\, v \in \mathcal{V}[\![\tau]\!] \Rightarrow f\, v \in \mathcal{E}[\![\sigma]\!]\}$$

$$\mathcal{E}[\![\tau]\!] \triangleq \{e \mid \exists v.\, e \Downarrow v \wedge v \in \mathcal{V}[\![\tau]\!]\} \qquad \mathcal{G}[\![\Gamma]\!] \triangleq \{\gamma \mid \forall x : \tau \in \Gamma.\, \gamma(x) \in \mathcal{V}[\![\tau]\!]\}$$

Figure 28.3: Select cases of the logical relation for heap-dependent expressions in the *hProp*-fragment.

side condition that arises is $P \vdash e \Downarrow \_$, where $e \Downarrow \_ \triangleq \exists v.\, e \Downarrow v$. It implicitly means that P constrains enough of the heap to ensure that e safely terminates in some value (considering the definition of $e \Downarrow v$ in EVAL-DEF). To simplify proving it, we will now introduce a semantic type system $\Gamma \vDash e : \tau$ that satisfies the property:

Lemma 82.
$(\emptyset \vDash e : \tau) \vdash e \Downarrow \_$*, meaning closed, well-typed expressions safely evaluate.*

The type system has typing rules for the simply-typed lambda calculus extended with standard data types (*e.g.*, integers, bitvectors, sums, pairs, vectors, buffers, *etc.*). A selection of typing rules is depicted in Fig. 28.2. For example, we can use TYPE-LAM to type λ-functions, TYPE-APP for function application, and TYPE-VAR for variables. Notably, the type system has no rules for expressions that cause side effects (*e.g.*, heap updates or forking threads), for non-deterministic expressions (*e.g.*, allocation), and for recursive functions, because evaluation $e \Downarrow v$ requires termination. We will get back to this last point shortly.

Semantic typing. Traditionally, a type system is a fixed collection of syntactic typing rules. Here, we (again)[2] use a *semantic type system* $\Gamma \vDash e : \tau$ defined via a logical relation, and then prove the typing rules as lemmas about $\Gamma \vDash e : \tau$. The key benefit of the semantic type system, as discussed below, is extensibility.

We define the logical relation—the basis for the typing judgment—in the *hProp*-fragment.[3] The logical relation is depicted in Fig. 28.3. For each type τ, the *value relation* $\mathcal{V}[\![\tau]\!]$ determines which values are of this type. For unit $\mathcal{V}[\![\mathsf{unit}]\!]$, Booleans $\mathcal{V}[\![\mathsf{bool}]\!]$, integers $\mathcal{V}[\![\mathsf{int}]\!]$, bitvectors $\mathcal{V}[\![\mathsf{bv}\, n]\!]$, options

[2] We also use a semantic type system for channels in λ_{CHAN} (§8.3 in Part II); for reorderability (§13.1.2 in Part III); and for automating reasoning about C programs (§20.1 in Part IV).

[3] Unlike in most Iris logical relations, the type interpretations do not have to be persistent in this logical relation. It suffices that they are *hProp*-assertions, because *hProp*-assertions are duplicable (see Lemma 80).

$\mathcal{V}[\![\mathsf{option}\,\tau]\!]$, sums $\mathcal{V}[\![\tau + \sigma]\!]$, and pairs $\mathcal{V}[\![\tau \times \sigma]\!]$, this is straightforward. Where things get more interesting is (1) stateful types like $\mathcal{V}[\![\mathsf{ref}\,\tau]\!]$ and $\mathcal{V}[\![\mathsf{buf}\,\tau]\!]$, because they can use the unstable points-to $\ell \mapsto_{\mathsf{u}} v$ of *hProp* to refer to the current contents of the memory[4] and (2) function types $\mathcal{V}[\![\tau \to \sigma]\!]$, because they can use the implication of *hProp* to say that a value f is semantically a function of type $\tau \to \sigma$ if, applied to an argument v of type τ, it results in a value of type σ. To express "it results in a value of type σ", we define the *expression relation* $\mathcal{E}[\![\tau]\!]$, which uses the evaluation $e \Downarrow v$ of *hProp* to evaluate e.

[4] Note that—unlike in the logical relation in, *e.g.*, §8.3 and §13.1.2, we do not need impredicative invariants here. The reason is that $\ell \mapsto_{\mathsf{u}} v$—unlike $\ell \mapsto v$—is duplicable (Lemma 80), which makes sharing it between different program parts trivial.

With both the value and expression relation in hand, we then define the semantic typing judgment as

$$\Gamma \vDash e : \tau \triangleq \forall \gamma.\, \gamma \in \mathcal{G}[\![\Gamma]\!] \Rightarrow \gamma(e) \in \mathcal{E}[\![\tau]\!].$$

That is, an expression e has type τ in context Γ if for any closing substitution $\gamma \in \mathcal{G}[\![\Gamma]\!]$, the expression after substituting the free variables, $\gamma(e)$, is semantically of type τ.

Extensibility. This typing judgment satisfies the typing rules above if we interpret them as implications in *hProp* (*e.g.*, $(x : \tau \in \Gamma) \Rightarrow (\Gamma \vDash x : \tau)$ for TYPE-VAR). Thus, we get all the benefits of a syntactic type system (*e.g.*, easily automatable rules). The reason why we define it semantically instead of a collection of syntactic rules is *extensibility*. For example, recall that the type system does not have a rule for recursive functions. However, for specific recursive functions such as the combinator `iter` (from §25.1), we can still *prove* that they are well-typed. Concretely, we can prove

$$\vDash \mathtt{iter} : \mathsf{int} \times \mathsf{int} \times \tau \times (\mathsf{int} \to \tau \to \tau) \to \tau$$

simply by unfolding the definition of $\Gamma \vDash e : \tau$ and reasoning about $e \Downarrow v$, because `iter` always terminates (when applied to terminating functions f).

Once we have done so, we can then add `iter` to the type system and—from here on—type check programs involving `iter`. For example, we can *type check* the following `checksum` implementation:

$$\mathtt{checksum}(b) \triangleq \mathtt{iter}(0, \mathsf{length}(b), 0, \lambda i, a.\, a\ \mathtt{xor}\ \mathsf{default}(\mathsf{nthopt}(b, i), 0))$$

as a function of type $\mathsf{buf}\,(\mathsf{bv}\,64) \to \mathsf{bv}\,64$. It iterates over the elements of the input buffer—accessed with $\mathsf{nthopt} : \mathsf{buf}\,\tau \times \mathsf{int} \to \mathsf{option}\,\tau$—and combines them with XOR. All we have to do is apply typing rules (which the Rocq implementation does automatically here).

In fact, we can now trivially derive $b \mapsto_q \vec{u} \vdash \mathtt{checksum}(b) \Downarrow \_$, which is the fact CHECKSUM-EVAL from §25.1:

Corollary 83. *If $\vec{u}$ is a sequence of 64-bit integers, then $b \mapsto_q \vec{u} \vdash \mathtt{checksum}(b) \Downarrow \_$.*

Proof. From $b \mapsto_q \vec{u}$, we have $b \mapsto_{\mathsf{u}} \vec{u}$ (since in general $\ell \mapsto_q v \vdash \ell \mapsto_{\mathsf{u}} v$; see GET-PTS-UNSTABLE in Fig. 26.6). Thus, by definition $b \in \mathcal{V}[\![\mathsf{buf}\,(\mathsf{bv}\,64)]\!]$. From TYPE-VAL, we deduce $\vDash b : \mathsf{buf}\,(\mathsf{bv}\,64)$. Thus, since we can type function application (TYPE-APP), we get $\vDash \mathtt{checksum}(b) : \mathsf{bv}\,64$. From $\vDash \mathtt{checksum}(b) : \mathsf{bv}\,64$, we can then deduce $\mathtt{checksum}(b) \Downarrow \_$ using Lemma 82. □

$$\begin{array}{rlcl}
\text{Sorts} & \mathsf{S}, \mathsf{T} & ::= & \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{int} \mid \mathsf{bv}\,n \mid \cdots \\
\text{Functions} & \mathsf{f}, \mathsf{g} & ::= & () \mid \mathsf{true} \mid \mathsf{false} \mid n \mid +_{\mathsf{int}} \mid \cdot_{\mathsf{int}} \mid -_{\mathsf{int}} \mid \mathsf{neg} \mid \mathsf{xor}_{\mathsf{bv}\,n} \\
 & & \mid & ==_{\mathsf{bv}\,n} \mid \mathsf{if}_{\mathsf{S}} \mid \cdots \\
\text{Predicates} & \mathsf{p}, \mathsf{q} & ::= & \leq_{\mathsf{int}} \mid <_{\mathsf{int}} \mid \cdots \\
\text{Terms} & t, s & ::= & \mathsf{x} \mid \mathsf{f}\,\vec{t} \\
\text{Formulas} & \pi, \chi & ::= & \mathsf{True} \mid \mathsf{False} \mid t \doteq_{\mathsf{S}} s \mid \mathsf{p}\,\vec{t} \mid \pi_1 \dot{\wedge} \pi_2 \mid \pi_1 \dot{\vee} \pi_2 \\
 & & \mid & \pi_1 \dot{\Rightarrow} \pi_2 \mid \dot{\exists}\mathsf{x} : \mathsf{S}.\, \pi \mid \dot{\forall}\mathsf{x} : \mathsf{S}.\, \pi
\end{array}$$

Figure 28.4: The syntax of the first-order logic of Daenerys.

28.2 The First-Order Logic Connection

Let us now connect $e \Downarrow v$ to first-order logic, which enables using an SMT solver to automate proofs. Our focus is on providing sound foundations for an SMT integration; developing an automated conversion between Rocq and SMT, let alone foundationally verifying an SMT solver, is beyond the scope of this dissertation. Instead, we justify why proving a formula π in first-order logic—oblivious to heaps—means that a corresponding *hProp*-formula $\langle \pi \rangle_{\mathsf{F}}$ over HDEAs can be assumed in Daenerys (Theorem 84). The payoff of this result is that, if one trusts an SMT solver to be sound w.r.t. standard first-order logic, then one can use it to verify properties with HDEAs (see §29). While IDF-based verifiers[5] rely on similar correspondences, to our knowledge, we are the first to establish one foundationally for HDEAs as rich as ours (*e.g.,* with functions).

We work with a standard multi-sorted first-order logic, depicted in Fig. 28.4, with (1) *sorts* such as integers int and bitvectors $\mathsf{bv}\,n$, (2) *functions* such as $+_{\mathsf{int}}$ for integer addition, (3) *predicates* such as $\leq_{\mathsf{int}}$ for integer less-or-equal, (4) *terms* consisting of variables x and function applications $\mathsf{f}\,\vec{t}$, and (5) *first-order logic formulas* π over them. The sorts, terms, and predicates in Fig. 28.4 are *interpreted*, meaning their semantics corresponds to the intuitive mathematical semantics (*e.g.,* $+_{\mathsf{int}}$ is integer addition and not subtraction). In addition, the logic can be freely extended with *uninterpreted* sorts, functions, and predicates (indicated by "$\cdots$" in Fig. 28.4), whose semantics is for us to choose.

For example, for the `validate`-example from §25.3, we use an *uninterpreted sort* buffer and two *uninterpreted functions* $\mathsf{validate} : \mathsf{buffer} \times \mathsf{bv}\,64 \longrightarrow \mathsf{bool}$ and $\mathsf{checksum} : \mathsf{buffer} \longrightarrow \mathsf{bv}\,64$ to express the key relationship between $\mathsf{checksum}$ and $\mathsf{validate}$ as the first-order logic formula π_{chk}:

$$\pi_{\mathsf{chk}} \triangleq \begin{array}{l} \left(\begin{array}{l} \dot{\forall}(\mathsf{b} : \mathsf{buffer}), (\mathsf{w} : \mathsf{bv}\,64). \\ \quad \mathsf{validate}(\mathsf{b}, \mathsf{w}) \doteq_{\mathsf{bool}} (\mathsf{checksum}(\mathsf{b})\ \mathsf{xor}_{\mathsf{bv}\,64}\ \mathsf{w} ==_{\mathsf{bv}\,64} 0) \end{array} \right) \dot{\Rightarrow} \\ \dot{\forall}(\mathsf{b} : \mathsf{buffer}), (\mathsf{v} : \mathsf{bv}\,64). \\ \quad \mathsf{checksum}(\mathsf{b}) \doteq_{\mathsf{bv}\,64} \mathsf{v} \dot{\Rightarrow} \mathsf{validate}(\mathsf{b}, \mathsf{v}) \doteq_{\mathsf{bool}} \mathsf{true} \end{array}$$

This is the kind of formula that an SMT solver like Z3[6] can prove. But note that it does *not* refer to the heap: $\mathsf{checksum}$ and $\mathsf{validate}$—to an SMT solver—are simply function symbols. We will now connect it to an *hProp*-assertion about the heap-accessing λ_{dyn}-functions `checksum` and `validate`.

The translation. To relate first-order logic and *hProp*-assertions, we introduce a translation $\langle \_ \rangle$ from the former to the latter. The translation is given

[5] Müller, Schwerhoff, and Summers, "Viper: A verification infrastructure for permission-based reasoning", 2017 [MSS17]; Astrauskas et al., "Leveraging Rust types for modular specification and verification", 2019 [Ast+19]; Wolf et al., "Gobra: Modular specification and verification of Go programs", 2021 [Wol+21]; Eilers and Müller, "Nagini: A static verifier for Python", 2018 [EM18].

[6] Moura and Bjørner, "Z3: An efficient SMT solver", 2008 [MB08].

Sort Translations

$$\langle\mathsf{unit}\rangle_{\mathsf{S}} \triangleq \mathsf{unit} \qquad \langle\mathsf{bool}\rangle_{\mathsf{S}} \triangleq \mathsf{bool} \qquad \langle\mathsf{int}\rangle_{\mathsf{S}} \triangleq \mathsf{int} \qquad \langle\mathsf{bv}\,n\rangle_{\mathsf{S}} \triangleq \mathsf{bv}\,n \qquad \cdots$$

Function Translations

$$\langle()\rangle_{\mathsf{C}} \triangleq () \qquad \langle\mathsf{true}\rangle_{\mathsf{C}} \triangleq \mathsf{true} \qquad \langle\mathsf{false}\rangle_{\mathsf{C}} \triangleq \mathsf{true} \qquad \langle n\rangle_{\mathsf{C}} \triangleq n$$

$$\langle +_{\mathsf{int}}\rangle_{\mathsf{C}} \triangleq \lambda(x,y).\, x+y \qquad \langle \cdot_{\mathsf{int}}\rangle_{\mathsf{C}} \triangleq \lambda(x,y).\, x*y \qquad \langle -_{\mathsf{int}}\rangle_{\mathsf{C}} \triangleq \lambda(x,y).\, x-y$$

$$\langle\mathsf{neg}\rangle_{\mathsf{C}} \triangleq \lambda x.\, \mathord{\sim}x \qquad \langle\mathsf{xor}_{\mathsf{bv}\,n}\rangle_{\mathsf{C}} \triangleq \lambda(x,y).\, x \;\mathtt{xor}\; y$$

$$\langle ==_{\mathsf{bv}\,n}\rangle_{\mathsf{C}} \triangleq \lambda(x,y).\, x == y \qquad \langle\mathsf{if}_{\mathsf{S}}\rangle_{\mathsf{C}} \triangleq \lambda(x,y_1,y_2).\, \mathsf{if}\; x \;\mathsf{then}\; y_1 \;\mathsf{else}\; y_2 \qquad \cdots$$

Predicate Translations

$$\langle \leq_{\mathsf{int}}\rangle_{\mathsf{P}}(v_1,v_2) \triangleq \exists n_1,n_2.\, v_1 = n_1 \wedge v_2 = n_2 \wedge n_1 \leq n_2$$

$$\langle <_{\mathsf{int}}\rangle_{\mathsf{P}}(v_1,v_2) \triangleq \exists n_1,n_2.\, v_1 = n_1 \wedge v_2 = n_2 \wedge n_1 \leq n_2 \qquad \cdots$$

Term Translations

$$\langle \mathsf{x}\rangle^{\gamma}_{\mathsf{T}} = \gamma(\mathsf{x}) \qquad \langle \mathsf{f}\,\vec{t}\rangle^{\gamma}_{\mathsf{T}} = \langle\mathsf{f}\rangle_{\mathsf{C}}\,\langle\vec{t}\rangle^{\gamma}_{\mathsf{T}}$$

Formula Translations

$$\langle\mathsf{True}\rangle^{\gamma}_{\mathsf{F}} \triangleq \mathsf{True} \qquad \langle\mathsf{False}\rangle^{\gamma}_{\mathsf{F}} \triangleq \mathsf{False} \qquad \langle t \doteq_{\mathsf{S}} s\rangle^{\gamma}_{\mathsf{F}} \triangleq \langle t\rangle^{\gamma}_{\mathsf{T}} \equiv \langle s\rangle^{\gamma}_{\mathsf{T}}$$

$$\langle \mathsf{p}\,\vec{t}\rangle^{\gamma}_{\mathsf{F}} \triangleq \langle\mathsf{p}\rangle_{\mathsf{P}}(\langle\vec{t}\rangle^{\gamma}_{\mathsf{T}}) \qquad \langle \pi_1 \dot{\wedge} \pi_2\rangle^{\gamma}_{\mathsf{F}} \triangleq \langle\pi_1\rangle^{\gamma}_{\mathsf{F}} \wedge \langle\pi_2\rangle^{\gamma}_{\mathsf{F}}$$

$$\langle \pi_1 \dot{\vee} \pi_2\rangle^{\gamma}_{\mathsf{F}} \triangleq \langle\pi_1\rangle^{\gamma}_{\mathsf{F}} \vee \langle\pi_2\rangle^{\gamma}_{\mathsf{F}} \qquad \langle \pi_1 \dot{\Rightarrow} \pi_2\rangle^{\gamma}_{\mathsf{F}} \triangleq \langle\pi_1\rangle^{\gamma}_{\mathsf{F}} \Rightarrow \langle\pi_2\rangle^{\gamma}_{\mathsf{F}}$$

$$\langle \dot{\forall}\mathsf{x}:\mathsf{S}.\, \pi\rangle^{\gamma}_{\mathsf{F}} \triangleq \forall v.\, v \in \mathcal{V}[\![\langle\mathsf{S}\rangle_{\mathsf{S}}]\!] \Rightarrow \langle\pi\rangle^{\gamma,\mathsf{x}\mapsto v}_{\mathsf{F}}$$

$$\langle \dot{\exists}\mathsf{x}:\mathsf{S}.\, \pi\rangle^{\gamma}_{\mathsf{F}} \triangleq \exists v.\, v \in \mathcal{V}[\![\langle\mathsf{S}\rangle_{\mathsf{S}}]\!] \wedge \langle\pi\rangle^{\gamma,\mathsf{x}\mapsto v}_{\mathsf{F}}$$

Figure 28.5: Translations of sorts to semantic types $\langle\mathsf{S}\rangle_{\mathsf{S}}$, function constants to λ_{dyn}-values $\langle\mathsf{f}\rangle_{\mathsf{C}}$, predicates to *hProp*-predicates $\langle\mathsf{p}\rangle_{\mathsf{P}}$, terms to λ_{dyn}-expressions $\langle t\rangle^{\gamma}_{\mathsf{T}}$, and formulas to *hProp*-assertions $\langle\pi\rangle^{\gamma}_{\mathsf{F}}$.

in Fig. 28.5.[7] It translates sorts S to types $\langle\mathsf{S}\rangle_{\mathsf{S}}$ (*e.g.*, mapping buffer to the type $\mathsf{buf}\,(\mathsf{bv}\,64)$), functions f to λ_{dyn}-values $\langle\mathsf{f}\rangle_{\mathsf{C}}$ (*e.g.*, mapping checksum to $\mathtt{checksum}$), predicates p to *hProp*-predicates $\langle\mathsf{p}\rangle_{\mathsf{P}}$, terms t to λ_{dyn}-expressions $\langle t\rangle^{\gamma}_{\mathsf{T}}$, and formulas π to *hProp*-assertions $\langle\pi\rangle^{\gamma}_{\mathsf{F}}$.

We discuss the most interesting cases. For term equality $t \doteq_{\mathsf{S}} s$, we use evaluation $e \Downarrow v$ via $e_1 \equiv e_2 \triangleq \exists v.\, e_1 \Downarrow v \wedge e_2 \Downarrow v$. For quantification $\dot{\forall}\mathsf{x}:\mathsf{S}.\,\pi$ and $\dot{\exists}\mathsf{x}:\mathsf{S}.\,\pi$, we use *hProp*-quantification, where we use the logical relation $\mathcal{V}[\![\tau]\!]$ (from §28.1) to constrain the values. For implication $\pi_1 \dot{\Rightarrow} \pi_2$, we map the implication of first-order logic directly to our *hProp*-implication $F \Rightarrow G$.

For example, translating π_{chk}, we obtain $F_{\mathsf{chk}} \triangleq \langle\pi_{\mathsf{chk}}\rangle^{\emptyset}_{\mathsf{F}}$, which is given by:

$$F_{\mathsf{chk}} = \begin{array}{l} \left(\begin{array}{l}\forall b,w.\, b \in \mathcal{V}[\![\mathsf{buf}\,(\mathsf{bv}\,64)]\!] \Rightarrow w \in \mathcal{V}[\![\mathsf{bv}\,64]\!] \Rightarrow \\ \qquad \mathtt{validate}(b,w) \equiv (\mathtt{checksum}(b)\;\mathtt{xor}\;w == 0)\end{array}\right) \Rightarrow \\ \forall b,v.\, b \in \mathcal{V}[\![\mathsf{buf}\,(\mathsf{bv}\,64)]\!] \Rightarrow v \in \mathcal{V}[\![\mathsf{bv}\,64]\!] \Rightarrow \\ \qquad \mathtt{checksum}(b) \equiv v \Rightarrow \mathtt{validate}(b,v) \equiv \mathsf{true} \end{array}$$

Superficially, this assertion looks similar to π_{chk} (which is the point of $\langle\pi\rangle^{\gamma}_{\mathsf{F}}$). However, there is one crucial difference: as an *hProp*-assertion, F_{chk} implicitly *refers to the current heap.* In typical SL fashion, the heap itself is hidden, but it

[7] In this presentation, cases for uninterpreted sorts, functions, and predicates are omitted.

is constrained via resources such as $\ell \mapsto_{\mathsf{u}} v$. For example, $b \in \mathcal{V}[\![\mathrm{buf}\,(\mathrm{bv}\,64)]\!]$ contains an unstable points-to for the buffer b (see Fig. 28.3), and $e_1 \equiv e_2$ implicitly evaluates `checksum` and `validate` on the current heap. In contrast, the formula π_{chk} is a first-order logic formula and does not mention a heap—neither explicitly nor implicitly.

Connecting *hProp*-assertions and first-order logic assertions is useful, because (as we will show below) it gives us access to formulas proven by the SMT solver in, *e.g.*, the rule of consequence.

The correspondence. Informally, we wish to prove that if π holds in first-order logic, then we can get $\langle\pi\rangle^{\emptyset}_{\mathsf{F}}$ in Iris. To make this formal, we must specify what it means for a formula π to hold. To do so, we define a standard Tarski semantics $\vDash \pi$ for first-order logic (see §30.1), where we make sure that the *interpreted* parts of the logic (*e.g.*, bitvectors and integers) have their standard mathematical semantics. With it, we establish the following result:

Theorem 84.
If $\vDash \pi$ *holds, then* $(\langle\pi\rangle^{\emptyset}_{\mathsf{F}} \Rightarrow \mathbf{wp}\; e\; \{v.\, Q(v)\}) \vdash \mathbf{wp}\; e\; \{v.\, Q(v)\}$ *holds in Iris.*

In other words, if π is true in first-order logic, then we can assume $\langle\pi\rangle^{\emptyset}_{\mathsf{F}}$ in Iris when proving a weakest precondition. (The weakest precondition gives us access to the current heap for $\langle\pi\rangle^{\emptyset}_{\mathsf{F}}$.) From this result, we can derive rules such as the consequence rule below (and hoare-assert in §25.3):

$$\textsc{hoare-conseq-fol}\; \frac{\vDash \pi \qquad P * \langle\pi\rangle^{\emptyset}_{\mathsf{F}} \vdash \boxplus Q \qquad \{Q\}\; e\; \{v.\, R(v)\}}{\{P\}\; e\; \{v.\, R(v)\}}$$

If π (*e.g.*, π_{chk}) holds in first-order logic, then we can assume $\langle\pi\rangle^{\emptyset}_{\mathsf{F}}$ (*e.g.*, F_{chk}) and use it to prove any frameable fact Q from it. In our case studies in §29, we use it implicitly to justify solving queries about, *e.g.*, functions manipulating bitvectors and buffers automatically with an SMT solver, which would otherwise involve tedious manual reasoning about these theories.

From almost-pure to actually-pure. We postpone a detailed discussion of the proof of Theorem 84 to §30, because it involves instantiating the Tarski semantics $\vDash \pi$ (for which we first need to *define* the Tarski semantics; see §30.1). For now, we focus on the key feature of *hProp*-assertions that we will use to bridge the gap between first-order logic and Iris: the function $\lfloor \_ \rfloor(h)$ that turns *hProp* assertions F into meta-level propositions $\lfloor F \rfloor(h)$. We use it in the proof of Theorem 84 to transition form first-order logic to Iris via the meta level:

$$\pi \textit{ in first-order logic} \longrightarrow \lfloor\langle\pi\rangle^{\emptyset}_{\mathsf{F}}\rfloor(h) \textit{ at the meta level} \longrightarrow \langle\pi\rangle^{\emptyset}_{\mathsf{F}} \textit{ in Iris}$$

Concretely, we will start from π in first-order logic. We will then obtain $\lfloor\langle\pi\rangle^{\emptyset}_{\mathsf{F}}\rfloor(h)$, which is the first-order formula π first translated into an *hProp*-assertion with $\langle\_\rangle^{\emptyset}_{\mathsf{F}}$ and then turned into a meta-level proposition (for a fixed heap h) with $\lfloor\_\rfloor(h)$. From $\lfloor\langle\pi\rangle^{\emptyset}_{\mathsf{F}}\rfloor(h)$, we will then cross the boundary to Iris and obtain $\langle\pi\rangle^{\emptyset}_{\mathsf{F}}$ in Iris with Lemma 81 above.

We will explore the proof of Theorem 84—and in particular the role of $\lfloor F\rfloor(h)$—in depth in §30. But before we dive into the details of the proof, we first consider several case studies of Daenerys in §29.

Chapter 29

Case Studies

We apply Daenerys to several case studies. First, we take a closer look at a case study that illustrates the benefits of combining Iris and IDF (§29.1). It is a variation of the checksum example from §24.1, where we leverage the combination of Iris's *impredicative invariants* (see §3.4.1) and HDEAs. Then, we consider several case studies in aggregate to highlight different aspects of Daenerys (§29.2).

29.1 The Best of Both Worlds

The example consists of two parts: (1) a concurrent channel library and (2) an exchange of a buffer and its checksum over a channel (from a worker-thread to a client). We first discuss the channel implementation (§29.1.1) and then proceed with how to verify the exchange (§29.1.2).

29.1.1 Channel Library

For the channel library, Daenerys benefits from its Iris roots. As we have discussed in §4.5, due to its elaborate, step-indexed model, Iris supports *impredicative invariants* (§3.4.1). They allow us to prove general and modular specifications for the channel operations, where one can pick an arbitrary Iris predicate Φ to be exchanged over a channel.

Implementation. The implementation of the channels is depicted in Fig. 29.1. Each channel is represented as a reference to an option. The option is either None if there is currently no value being transferred over the channel, or Some(ℓ) where ℓ is a reference storing the value that is currently being transferred.[1]

The three operations for channels are `chan`, `send`, and `recv`.

1. The operation `chan` initializes an new channel by creating the channel reference, initially storing None (*i.e.,* no value is currently being transferred).

2. The operation `send` wraps the value it wants to send x in a reference l, and then tries to store Some(l) in the channel reference c. It does so via an atomic "compare-and-set" operation that tests whether the channel is currently empty (*i.e.,* currently storing None) and, if so, updates it to Some(l). If the update succeeds, `send` returns, and if it fails, `send` loops to wait for a point where the channel is empty again.

[1] We add the additional indirection via a reference wrapping the value, because HeapLang and by extension λ_{dyn} forbid "compare-and-set" operations on references that store larger, composite values such as a pair. See the side condition v_1, v_2 *comparable* in, *e.g.,* HOARE-CAS-SUC in Fig. 27.1

Channel Implementation

```
chan() ≜ ref(None)
recv(c) ≜ let v = ! c in
          match v with
          | None ⇒ recv(c)
          | Some(l) ⇒ if CAS(c, Some(l), None) then !l else recv(c)
          end
send(c, x) ≜ let l = ref(x) in
             if CAS(c, None, Some(l)) then () else send(c, x)
```

Channel Specification

$\{\mathsf{True}\}\ \mathsf{chan}()\ \{c.\ \mathsf{ischan}(c, \Phi)\}$ $\qquad$ $\mathsf{persistent}(\mathsf{ischan}(c, \Phi))$

$\{\mathsf{ischan}(c, \Phi) * \boxplus\Phi(v)\}\ \mathsf{send}(c, v)\ \{\_.\ \mathsf{True}\}$

$\{\mathsf{ischan}(c, \Phi)\}\ \mathsf{recv}(c)\ \{v.\ \boxplus\Phi(v)\}$

Figure 29.1: The channel implementation and specification.

3. The operation `recv` reads the contents v of the channel reference c. If they are None, then no value is currently being transferred and the `recv` operation loops to wait for a value. If there is some value currently being transferred (wrapped in a reference l) then `recv` tries to update c to None with a "compare-and-set" to claim the value. If the update succeeds, then the `recv` can return the contents of l (*i.e.,* the value being transferred). If the update fails, another call to `recv` must have been successful at receiving the value. In this case, `recv` loops to wait for the next value.

Verification. To verify the channel operations, we prove the specifications in Fig. 29.1. That is, when we create a channel with `chan`, we get an abstract predicate $\mathsf{ischan}(c, \Phi)$ that can be shared between threads (*i.e.,* is persistent). We can send a value v satisfying Φ with `send` and receive a value satisfying Φ on the other end with `recv`. When we send a value, the predicate Φ should not depend on any unstable resources, which is ensured by the frame modality $\boxplus$.

To define the abstract predicate $\mathsf{ischan}(c, \Phi)$, we use an Iris invariant. Concretely, we define

$$\mathsf{ischan}(c, \Phi) \triangleq \exists \ell.\ c = \ell * \boxed{I_{\mathrm{chan}}(\ell)}^{\mathcal{N}}$$
$$I_{\mathrm{chan}}(\ell) \triangleq \exists v.\ \ell \mapsto v * (v{=}\mathsf{None} \lor \exists \ell', w.\ v{=}\mathsf{Some}(\ell') * \ell' \mapsto w * \boxplus\Phi\, w)$$

It captures the two possible cases the channel reference ℓ can be in: either (1) no value is currently being transferred and the reference is storing None, or (2) some value w is being transferred, wrapped in some reference ℓ'. In the latter case, the invariant stores that Φ holds for the value being currently transferred. It does so underneath a frame modality to ensure that the ownership being transferred via the channel is stable. (Formally, this modality is needed to make sure that the contents of the invariant are frameable, a side condition of invariants in Daenerys; see §27.1.) With this invariant, verifying the specifications in Fig. 29.1 is straightforward.

29.1.2 Checksum Exchange

We apply the channels to exchange a buffer and its checksum between a worker and a client. The worker and client are implemented as follows:

```
wrk(i, o) ≜ let (p, c) = recv(i) in
            let b = p() in
            let s = c(b) in
            send(o, (b, s));
            wrk(i, o)
client() ≜ let (i, o) = (chan(), chan()) in
           fork {wrk(i, o)} ;
           send(i, (produceA, checksumA));
           let (b, s) = recv(o) in
           assert(s == checksumA(b));
           send(i, (produceB, checksumB));
           let (b, s) = recv(o) in
           assert(s == checksumB(b))
```

Implementation. The client `client` creates an input and an output channel, spawns the worker thread, and then sends the worker two different workloads: First, it sends produceA with checksum function checksumA, receives the result b and s, and ensures that s matches checksumA of b. Then, it repeats the same process with a *different workload and checksum* implementation. The worker wrk (1) receives on the input channel i a workload p (*i.e.,* a function that will produce a buffer) and a checksum function c, (2) produces the buffer b by executing p, (3) computes the checksum s of b, (4) sends both b and s back via the output channel o, and (5) repeats the entire process.

Verification. Let us now discuss how we can verify this implementation given the channel specifications in Fig. 29.1. We start by introducing the two channel predicates, one for the input channel and one for the output channel:

$$\begin{aligned}
\Phi_{\text{inp}}(p, c) &\triangleq \{\mathsf{True}\}\; p()\; \{v.\, \exists b, \vec{u}.\, v = b * b \mapsto \vec{u}\} \\
&\quad * \;\Box(c \in \mathcal{V}[\![\mathsf{buf}\,(\mathsf{bv}\,64) \to \mathsf{bv}\,64]\!]) * \gamma \Mapsto_{1/2} (p, c) \\
\Phi_{\text{outp}}(b, s) &\triangleq \exists \vec{u}, p, c.\, b \mapsto \vec{u} * c(b) \Downarrow s * \gamma \Mapsto_{1/2} (p, c)
\end{aligned}$$

On the input channel (Φ_{inp}), the client sends a pair of *two functions* p and c. For p, it sends a Hoare triple that p will compute a buffer. For c, it sends the fact that c is a well-typed function. In addition, it sends a custom piece of ghost state, a fractional ghost variable $\gamma \Mapsto_{1/2} (p, c)$, to track which workload the worker is currently working on (see §4.2.2). As we will see below, the client keeps one half $\gamma \Mapsto_{1/2} (p, c)$ and sends the other half to the worker (via Φ_{inp}).

On the output channel (Φ_{outp}), the worker sends a pair of buffer b and checksum s. For b, it sends the ownership of the buffer $b \mapsto \vec{u}$. For s, it sends $c(b) \Downarrow s$, meaning s is the result of computing the checksum c on the buffer b. Along with them, it returns the ghost variable $\gamma \Mapsto_{1/2} (p, c)$.

■ $\{\mathsf{True}\}$

let $(i, o) = (\mathsf{chan}(), \mathsf{chan}())$ in

■ $\{\mathsf{ischan}(i, \Phi_{\mathrm{inp}}) * \mathsf{ischan}(o, \Phi_{\mathrm{outp}}) * \gamma \Mapsto_1 (\_, \_)\}$

fork $\{\mathtt{wrk}(i, o)\}$;

■ $\{\mathsf{ischan}(i, \Phi_{\mathrm{inp}}) * \mathsf{ischan}(o, \Phi_{\mathrm{outp}}) * \gamma \Mapsto_1 (\_, \_)\}$

■ $\{\mathsf{ischan}(i, \Phi_{\mathrm{inp}}) * \mathsf{ischan}(o, \Phi_{\mathrm{outp}}) * \gamma \Mapsto_{1/2} (\mathsf{produceA}, \mathsf{checksumA}) * \gamma \Mapsto_{1/2} (\mathsf{produceA}, \mathsf{checksumA})\}$

send$(i, (\mathsf{produceA}, \mathsf{checksumA}))$;

■ $\{\mathsf{ischan}(i, \Phi_{\mathrm{inp}}) * \mathsf{ischan}(o, \Phi_{\mathrm{outp}}) * \gamma \Mapsto_{1/2} (\mathsf{produceA}, \mathsf{checksumA})\}$

let $(b, s) = \mathsf{recv}(o)$ in

■ $\left\{\begin{array}{l}\mathsf{ischan}(i, \Phi_{\mathrm{inp}}) * \mathsf{ischan}(o, \Phi_{\mathrm{outp}}) * \gamma \Mapsto_{1/2} (\mathsf{produceA}, \mathsf{checksumA}) * \\ \boxplus\, (\exists \vec{u}, p, c.\ b \mapsto \vec{u} * c(b) \Downarrow s * \gamma \Mapsto_{1/2} (p, c))\end{array}\right\}$

■ $\left\{\begin{array}{l}\mathsf{ischan}(i, \Phi_{\mathrm{inp}}) * \mathsf{ischan}(o, \Phi_{\mathrm{outp}}) * \gamma \Mapsto_{1/2} (\mathsf{produceA}, \mathsf{checksumA}) * \\ (\exists \vec{u}.\ b \mapsto \vec{u} * \mathsf{checksumA}(b) \Downarrow s * \gamma \Mapsto_{1/2} (\mathsf{produceA}, \mathsf{checksumA}))\end{array}\right\}$

assert$(s = \mathsf{checksumA}(b))$;

■ $\{\mathsf{ischan}(i, \Phi_{\mathrm{inp}}) * \mathsf{ischan}(o, \Phi_{\mathrm{outp}}) * \gamma \Mapsto_1 (\_, \_)\}$

■ $\{\mathsf{ischan}(i, \Phi_{\mathrm{inp}}) * \mathsf{ischan}(o, \Phi_{\mathrm{outp}}) * \gamma \Mapsto_{1/2} (\mathsf{produceB}, \mathsf{checksumB}) * \gamma \Mapsto_{1/2} (\mathsf{produceB}, \mathsf{checksumB})\}$

send$(i, (\mathsf{produceB}, \mathsf{checksumB}))$;

■ $\{\mathsf{ischan}(i, \Phi_{\mathrm{inp}}) * \mathsf{ischan}(o, \Phi_{\mathrm{outp}}) * \gamma \Mapsto_{1/2} (\mathsf{produceB}, \mathsf{checksumB})\}$

let $(b, s) = \mathsf{recv}(o)$ in

■ $\left\{\begin{array}{l}\mathsf{ischan}(i, \Phi_{\mathrm{inp}}) * \mathsf{ischan}(o, \Phi_{\mathrm{outp}}) * \gamma \Mapsto_{1/2} (\mathsf{produceB}, \mathsf{checksumB}) * \\ \boxplus\, (\exists \vec{u}', p, c.\ b \mapsto \vec{u}' * c(b) \Downarrow s * \gamma \Mapsto_{1/2} (p, c))\end{array}\right\}$

■ $\left\{\begin{array}{l}\mathsf{ischan}(i, \Phi_{\mathrm{inp}}) * \mathsf{ischan}(o, \Phi_{\mathrm{outp}}) * \gamma \Mapsto_{1/2} (\mathsf{produceB}, \mathsf{checksumB}) * \\ (\exists \vec{u}'.\ b \mapsto \vec{u}' * \mathsf{checksumB}(b) \Downarrow s * \gamma \Mapsto_{1/2} (\mathsf{produceB}, \mathsf{checksumB}))\end{array}\right\}$

assert$(s = \mathsf{checksumB}(b))$

■ $\{\mathsf{True}\}$

Figure 29.2: Proof outline of the client. Separation logic states in ■ orange.

Given these two predicates, the specifications that we have proven are:

$$\{\mathsf{True}\}\ \mathtt{client}()\ \{\_.\ \mathsf{True}\}$$

$$\{\mathsf{ischan}(i, \Phi_{\mathrm{inp}}) * \mathsf{ischan}(o, \Phi_{\mathrm{outp}})\}\ \mathtt{wrk}(i, o)\ \{\_.\ \mathsf{True}\}$$

The client is safe to execute (*i.e.*, pre- and postcondition True), which means the asserts inside must succeed. The worker thread is safe to execute given an input channel adhering to Φ_{inp} and an output channel adhering to Φ_{outp}.

Proof sketch. To illustrate the basic structure of the proof, we give a proof outline of the client in Fig. 29.2. Initially, we allocate the two channels (using the specification of chan in Fig. 29.1). We additionally allocate a new ghost variable γ initially storing an arbitrary pair of values. Next, we fork-off the worker thread. To do so, we give it a copy of the representation predicate of the input/output channel $\mathsf{ischan}(i, \Phi_{\mathrm{inp}}) * \mathsf{ischan}(o, \Phi_{\mathrm{outp}})$ and keep another copy to ourselves.

Group	Case Study	Iris [Jun+18b]	ViperCore [Dar+25]	Viper [MSS17]	Daenerys
#1	Channel Library	●	○	○	●
	Checksum Exchange	○	○	●	●
#2	Popcount 32-bit Integer	◐	●	●	●
	Popcount Buffer *à la* Redis [Red25b]	◐	●	●	●
	Priority Bit Map *à la* RefinedC [Ref25]	●	●	●	●
#3	Iterative Linked-List	○	○	●	●
#4	Polymorphic Hashmap	○	○	●	●
#5	Iris Concurrent Logical Relation [Tim+24b]	●	○	○	●
	Barrier [Jun+16], Reader-Writer Lock, Spinlock	●	○	○	●
Foundational Model		●	●	○	●

Figure 29.3: Evaluation of Daenerys. We compare with the other approaches Iris, ViperCore, and Viper-based verifiers and mark whether they support the case study. We write ● for *yes*, ○ for *no*, and ◐ for case studies that *could conceivably be done* but require significant manual effort in Iris.

Next, we start with the first checksum-and-produce combination. We update the ghost variable and split it into two halves: $\gamma \Mapsto_{1/2} (\texttt{produceA}, \texttt{checksumA}) \ast \gamma \Mapsto_{1/2} (\texttt{produceA}, \texttt{checksumA})$. Moreover, we prove the following specifications for `produceA` and `checksumA`:

$$\{\mathsf{True}\}\ \texttt{produceA}()\ \{v.\ \exists b, \vec{u}.\ v = b \ast b \mapsto \vec{u}\}$$

$$\vDash \texttt{checksumA} : \mathsf{buf}\,(\mathsf{bv}\,64) \to \mathsf{bv}\,64$$

Together, we can then send $\Phi_{\mathrm{inp}}(\texttt{produceA}, \texttt{checksumA})$ to the worker thread via the input channel. (The assertion $\Phi_{\mathrm{inp}}(\texttt{produceA}, \texttt{checksumA})$ is frameable, meaning $\Phi_{\mathrm{inp}}(\texttt{produceA}, \texttt{checksumA}) \vdash \boxplus\Phi_{\mathrm{inp}}(\texttt{produceA}, \texttt{checksumA})$, so we can cross the frame modality in the specification of send.)

Next, we receive on the output channel the resulting combination of buffer b and checksum s, meaning $\Phi_{\mathrm{outp}}(b, s)$. We have two halves of the ghost variable γ: $\gamma \Mapsto_{1/2} (\texttt{produceA}, \texttt{checksumA})$ and $\gamma \Mapsto_{1/2} (p, c)$ at this point. Just like for a fractional points-to, this means the two must agree (so $p = \texttt{produceA}$ and $c = \texttt{checksumA}$). Thus, the client has just received the knowledge of $\texttt{checksumA}(b) \Downarrow s$. It can use it to justify that the assert succeeds.

Finally, we can combine the ghost variable again into one, and analogously proceed with the second checksum-and-produce combination.

29.2 Aggregate Evaluation

We apply Daenerys to several case studies, depicted in Fig. 29.3. We discuss them below in five groups. To give an impression how Daenerys compares to other approaches, Fig. 29.3 provides a comparison with Iris,[2] tools based on Viper[3] as expressive representatives of SL and IDF, and with ViperCore,[4] a recent foundational formalization of IDF and a subset of Viper.

#1 The best of both worlds. With the example discussed in §29.1, Group 1 illustrates how the marriage of Iris and IDF goes beyond what either approach typically provides in isolation. Similar to the example in §25.2, HDEAs allow us here to avoid proving functional correctness of (two) checksum implementations. Instead, we can simply send the assertion $c(b) \Downarrow s$ from the worker to the client. We get the best of both worlds. The verification of the channels is

[2] Jung et al., "Iris from the ground up: A modular foundation for higher-order concurrent separation logic", 2018 [Jun+18b].

[3] Müller, Schwerhoff, and Summers, "Viper: A verification infrastructure for permission-based reasoning", 2017 [MSS17]; Wolf et al., "Gobra: Modular specification and verification of Go programs", 2021 [Wol+21]; Eilers and Müller, "Nagini: A static verifier for Python", 2018 [EM18]; Astrauskas et al., "Leveraging Rust types for modular specification and verification", 2019 [Ast+19].

[4] Dardinier et al., "Formal foundations for translational separation logic verifiers", 2025 [Dar+25].

$$
\begin{aligned}
\pi_{\text{pc32}} \triangleq\, & (\dot\forall \mathsf{x}.\ \mathsf{aux1}(\mathsf{x}) \doteq_{\text{bv}\,32} \mathsf{x} -_{\text{bv}\,32} ((\mathsf{x} \gg_{\text{bv}\,32} 1) \mathbin{\&_{\text{bv}\,32}} \mathtt{0x55555555})) \mathrel{\dot\Rightarrow} \\
& (\dot\forall \mathsf{y}.\ \mathsf{aux2}(\mathsf{y}) \doteq_{\text{bv}\,32} (\mathsf{y} \mathbin{\&_{\text{bv}\,32}} \mathtt{0x33333333}) +_{\text{bv}\,32} ((\mathsf{y} \gg_{\text{bv}\,32} 2) \mathbin{\&_{\text{bv}\,32}} \mathtt{0x33333333})) \mathrel{\dot\Rightarrow} \\
& (\dot\forall \mathsf{z}.\ \mathsf{aux3}(\mathsf{z}) \doteq_{\text{bv}\,32} ((\mathsf{z} +_{\text{bv}\,32} (\mathsf{z} \gg_{\text{bv}\,32} 4)) \mathbin{\&_{\text{bv}\,32}} \mathtt{0x0F0F0F0F}) \mathrel{\dot\Rightarrow} \\
& (\dot\forall \mathsf{x}.\ \mathsf{aux}(\mathsf{x}) \doteq_{\text{bv}\,32} \mathsf{aux3}(\mathsf{aux2}(\mathsf{aux1}(\mathsf{x})))) \mathrel{\dot\Rightarrow} \\
& (\dot\forall \mathsf{x}.\ \mathsf{pc32}(\mathsf{x}) \doteq_{\text{bv}\,32} (\mathsf{aux}(\mathsf{x}) \mathbin{\&_{\text{bv}\,32}} \mathtt{0x01010101}) \gg_{\text{bv}\,32} 24) \mathrel{\dot\Rightarrow} \\
& (\dot\forall \mathsf{x}.\ \mathsf{ones}(\mathsf{x}) \doteq (\mathsf{x} \gg_{\text{bv}\,32} 31) \mathbin{\&_{\text{bv}\,32}} \mathtt{0x1} +_{\text{bv}\,32} \cdots +_{\text{bv}\,32} (\mathsf{x} \gg_{\text{bv}\,32} 0) \mathbin{\&_{\text{bv}\,32}} \mathtt{0x1}) \mathrel{\dot\Rightarrow} \\
& \quad \dot\forall \mathsf{x}.\ \mathsf{True} \mathrel{\dot\Rightarrow} \dot\forall \mathsf{r}.\ \mathsf{pc32}(\mathsf{x}) \doteq_{\text{bv}\,32} \mathsf{r} \mathrel{\dot\Rightarrow} \mathsf{r} \doteq_{\text{bv}\,32} \mathsf{ones}(\mathsf{x})
\end{aligned}
$$

Figure 29.4: Query π_{pc32} for the Popcount Single example from Group 2.

beyond the scope of Viper (due to the higher-order abstract predicate), and reasoning about checksum as an HDEA is beyond Iris.

#2 Leveraging SMT solvers. Group 2 contains case studies that illustrate the benefit of the connection to first-order logic (§28.2). Specifically, we consider case studies where SMT solvers provide automation that otherwise, in a regular Iris proof, would require tedious manual reasoning about, *e.g.*, bitvectors. A poster child example in this category is the function[5]

$$
\begin{aligned}
\mathsf{pc32}(x) \triangleq\ & \mathsf{let}\ y = x - (x \gg 1 \mathbin{\&} \mathtt{0x55555555})\ \mathsf{in} \\
& \quad \mathsf{let}\ z = (y \mathbin{\&} \mathtt{0x33333333}) + ((y \gg 2) \mathbin{\&} \mathtt{0x33333333})\ \mathsf{in} \\
& \quad (((z + (z \gg 4)) \mathbin{\&} \mathtt{0x0F0F0F0F}) * \mathtt{0x01010101}) \gg 24
\end{aligned}
$$

[5] To match the formulation of traditional first-order logic, the first-order logic discussed in §28.2 does not have let-bindings in its terms. Thus, we encode the let-bindings in pc32 via auxiliary functions in this example (see Fig. 29.4).

It counts the number of *ones* in the 32-bit bitvector x via an intricate combination of shifting, masking, addition, and multiplication. However, the SMT solver Z3 can show in an instant that pc32 behaves the same as

$$\mathsf{ones}(x) \triangleq (x \gg 31) \mathbin{\&} \mathtt{0x1} + \cdots + (x \gg 0) \mathbin{\&} \mathtt{0x1}.$$

Thus, (1) we encode the desired relationship between the two functions into a first-order logic formula π_{pc32} (depicted in Fig. 29.4), (2) we use our first-order connection (Theorem 84) to show:

$$\vDash \pi_{\text{pc32}} \quad \textit{implies} \quad \{0 \le u < 2^{32}\}\ \mathsf{pc32}(u)\ \{v.\, v \equiv \mathsf{ones}(u)\},$$

and then (3) we ask Z3 to prove the first-order logic formula π_{pc32}.

This case study ("Popcount 32-bit Integer" in Fig. 29.3) is a toy version of "Popcount Buffer", which is inspired by a popcount implementation from Redis[6] and works on a buffer of 32-bit integers, processing *seven integers at a time* in a loop using a scaled up version of the expression in pc32.

[6] Redis, *Redis popcount implementation for potentially large buffers*, 2025 [Red25b].

Furthermore, we verify a version of a bit map implementation previously verified in RefinedC.[7] Since Rocq provides little to no automation for bitvectors, the RefinedC-version requires around 300 lines of manual reasoning about bitvector arithmetic. In contrast, in Daenerys, we prove *not a single lemma* about bitvector arithmetic manually thanks to its connection to SMT solvers. (This connection requires a significant amount of boilerplate code at the moment that would be straightforward to automatically generate.) Note that Viper-based tools can verify all case studies in this category, but in contrast to ours, their encoding of assertions into first-order logic is not foundational.

[7] Sammler et al., "RefinedC: Automating the foundational verification of C code with refined ownership types", 2021 [Sam+21].

#3 Incremental verification. In Group 3, we incrementally verify a linked-list with increasingly stronger specifications—one of the strengths of IDF. In each step, we use an additional function operating on the linked-list to expose additional information about the data structure. For example, for the `set`-function, we prove:

$$\{\mathsf{list}(l)\}\ \mathtt{set}(l,i,x)\ \{\_, \_.\ \mathsf{list}(l)\}$$

$$\{\mathsf{list}(l)\}\ \mathsf{set}(l,i,x)\ \{\_, \mathbf{old}.\ \mathsf{list}(l) * \mathtt{len}(l) \Downarrow \mathbf{old}\,\{\mathtt{len}(l)\}\}$$

$$\begin{array}{l} \{\mathsf{list}(l)\} \\ \quad \mathsf{set}(l,i,x) \\ \left\{ \_, \mathbf{old}. \begin{array}{l} \mathsf{list}(l) * \mathtt{len}(l) \Downarrow \mathbf{old}\,\{\mathtt{len}(l)\} * (0 \le_{\mathsf{hp}} i <_{\mathsf{hp}} \mathtt{len}(l) \Rightarrow \mathsf{nth}(l,i) \Downarrow x) \\ * \ (\forall (j : \mathbb{Z}).\ 0 \le_{\mathsf{hp}} j <_{\mathsf{hp}} \mathtt{len}(l) \wedge i \neq_{\mathsf{hp}} j \Rightarrow \mathsf{nth}(l,j) \Downarrow \mathbf{old}\,\{\mathsf{nth}(l,j)\}) \end{array} \right\} \end{array}$$

where $e_1 \sim_{\mathsf{hp}} e_2 \triangleq \exists n_1, n_2. e_1 \Downarrow n_1 \wedge e_2 \Downarrow n_2 \wedge n_1 \sim n_2$ for $\sim \in \{<, \le\}$. The first specification expresses memory safety. The second one establishes that the length of the list, `len`, does not change. It uses an "old expression" $\mathbf{old}\,\{e\}$, which refers to the value of e in the precondition (defined below). The third specification additionally uses an `nth`-function to say that the i-th value becomes x and that all others are unchanged. Each step only involves strengthening the postcondition (thereby leaving proofs of clients intact) and only requires adding to the original proofs. This form of incremental verification is also supported in Viper-based tools (although functions such as `len` are not formalized in ViperCore).

In separation logics like Iris, in contrast, this form of incremental verification is not supported, because when choosing a representation predicate such as list, one simultaneously decides on an abstraction of the data structure (*e.g.,* length, contents, the underlying heap fragment, *etc.*). If one later wants to track a richer abstraction of the data structure, one has to define a new predicate and adapt the verification. (While one could start with a rich abstraction from the beginning but not expose all information to clients yet, this means already reasoning about the rich abstraction, *e.g.,* the list contents, when proving simpler properties like memory safety.) With HDEAs, the predicate stays the same and each function on the data structure exposes an additional abstraction.

The specifications in this example are slightly different in that they use so-called "old expressions" $\mathbf{old}\,\{e\}$, which refer to the value of e in the precondition. To encode them in Daenerys, we define a new triple $\{P\}\ e\ \{v, \mathbf{old}.\ Q(v, \mathbf{old})\}$. In the postcondition, it gets access to $\mathbf{old} : \mathit{Expr} \to \mathit{Val}$, a function from expressions to their resulting values if executed in the heap of the precondition. To define it, we use classical reasoning. Specifically, using the axiom of choice and excluded middle, one can obtain a evaluation function $\mathsf{eval} : \mathit{Heap} \to \mathit{Expr} \to \mathit{Val}$ such that if $(e, h) \rightsquigarrow^*_{\mathsf{det}} (v, h)$, then $\mathsf{eval}\,h\,e = v$. With it, we define:

$$\begin{array}{l} \{P\}\ e\ \{v, \mathbf{old}.\ Q(v, \mathbf{old})\} \triangleq \\ \qquad \Box(\forall h.\ \boxplus P \mathrel{-\!\!*} (\mathsf{heap}_{\mathsf{u}}(h) * (\mathop{\ast}_{\ell \mapsto v \in h} \ell \mapsto_{\mathsf{u}} v)) \mathrel{-\!\!*} \mathbf{wp}\ e\ \{v.\ \boxplus Q(v, \mathsf{eval}\,h)\}) \end{array}$$

Compared to the regular Hoare-triples $\{P\}\ e\ \{v.\ Q(v)\}$ (from §27.2), this version additionally assumes knowledge about the current heap h in the precondition. In the postcondition, we can then use eval partially instantiated with the precondition heap h to be able to refer to values of expressions in the precondition.

As shown in the Rocq development, for postconditions that do not care about old expressions, this triple is equivalent to the regular triple from §27.2.

#4 Polymorphic hashmap. Group 4 verifies a polymorphic hashmap implementation to illustrate how we can use HDEAs to *relate different program expressions* without abstracting them to mathematical functions. This hashmap takes a user-provided equality function eq and a hash function hash and relies on the following property of these functions:

$$\forall x, y.\ \mathsf{eq}(x, y) \equiv \mathsf{true} \Rightarrow \mathsf{hash}(x) \equiv \mathsf{hash}(y) \qquad (\textsc{eq-hash-rel})$$

What is interesting about eq-hash-rel is that we can state the relationship directly on the code of eq and hash using an almost-pure assertion. In a traditional Iris proof, one would first model eq and hash as mathematical functions *eq* and *hash* in order to state eq-hash-rel as a pure property. Furthermore, we can use an SMT solver to prove eq-hash-rel for concrete instantiations.

#5 Iris examples. In Group 5, we ported existing Iris proofs to Daenerys to show that Daenerys retains the expressiveness of Iris. We ported the rich logical relation of Timany et al.[8] and several fine-grained concurrency examples, including the challenging Barrier example of Jung et al.[9] In all cases, the effort was at most a few hours, and the delta over the original proofs is negligible (mostly adding frame modalities and introducing them in proofs). Viper-based tools do not support the expressive Iris features needed for these examples.

[8] Timany et al., “A logical approach to type soundness”, 2024 [Tim+24b].

[9] Jung et al., “Higher-order ghost state”, 2016 [Jun+16].

Chapter 30

Connecting Iris with First-Order Logic (Appendix)

In this chapter, we discuss the proof of the main theorem connecting the HDEAs of Daenerys to first-order logic (from §28.2):

Theorem 84.
If $\vDash \pi$ holds, then $(\langle\pi\rangle^{\emptyset}_{\mathsf{F}} \Rightarrow \mathbf{wp}\ e\ \{v.\ Q(v)\}) \vdash \mathbf{wp}\ e\ \{v.\ Q(v)\}$ holds in Iris.

We proceed in two steps: First, we make the validity judgment $\vDash \pi$ precise by defining the Tarski-semantics of multi-sorted first-order logic with interpreted theories such as bitvectors and integers (in §30.1). Then, we instantiate this semantics with a first-order logic model to obtain a proof of Theorem 84 (in §30.2).

30.1 Many-Sorted First-Order Logic

We define the validity judgment $\vDash \pi$ in two layers. First, we discuss our generic definition of first-order logic and its Tarski-semantics (§30.1.1). It is parametric over a signature and semantics for, *e.g.*, function symbols. Then, in the second layer, we instantiate the generic definition to ensure that the interpreted theories such as bitvectors and integers have their intended meaning (§30.1.2).

30.1.1 Generic First-Order Logic

For the generic first-order logic, we define first-order terms t and formulas π in Fig. 30.1, parametrically over a signature $\Omega = (\mathcal{S}, \mathcal{C}, \mathcal{P}, \vdash_{\text{func}}, \vdash_{\text{pred}})$, where $\mathcal{S}$ is a collection of first-order sort symbols; $\mathcal{C}$ a collection of first-order function symbols; $\mathcal{P}$ a collection of first-order predicate symbols; $\vdash_{\text{func}}$ assigns argument sorts and a result sort to every function symbol $\mathsf{f} \in \mathcal{C}$; and $\vdash_{\text{func}}$ assigns argument sorts to every predicate symbol $\mathsf{p} \in \mathcal{P}$. We write $\vdash_{\text{func}} \mathsf{f} : \vec{\mathsf{S}} \longrightarrow \mathsf{T} \in \mathcal{C}$ to mean $\mathsf{f} \in \mathcal{C}$ and $\vdash_{\text{func}} \mathsf{f} : \vec{\mathsf{S}} \longrightarrow \mathsf{T}$. We write $\vdash_{\text{pred}} \mathsf{p} : \vec{\mathsf{S}} \longrightarrow \mathsf{Prop} \in \mathcal{P}$ to mean $\mathsf{p} \in \mathcal{P}$ and $\vdash_{\text{pred}} \mathsf{p} : \vec{\mathsf{S}} \longrightarrow \mathsf{Prop}$.

Definition 85. *We say a signature $\Omega_1 = (\mathcal{S}_1, \mathcal{C}_1, \mathcal{P}_1, \vdash_{\text{func}1}, \vdash_{\text{pred}_1})$ extends a signature $\Omega_2 = (\mathcal{S}_2, \mathcal{C}_2, \mathcal{P}_2, \vdash_{\text{func}2}, \vdash_{\text{pred}_2})$, written $\Omega_1 \sqsupseteq_{\text{sig}} \Omega_2$, iff. (1) $\mathcal{S}_1 \supseteq \mathcal{S}_2$, (2) $\mathcal{C}_1 \supseteq \mathcal{C}_2$, (3) $\mathcal{P}_1 \supseteq \mathcal{P}_2$, and (4) the typing judgments agree:*

1. *$\vdash_{\text{pred}_2} \mathsf{p} : \vec{\mathsf{S}} \longrightarrow \mathsf{Prop}$ implies $\vdash_{\text{pred}_1} \mathsf{p} : \vec{\mathsf{S}} \longrightarrow \mathsf{Prop}$ for every $\mathsf{p} \in \mathcal{P}_2$ and $\vec{\mathsf{S}} \in \mathcal{S}_2$.*
2. *$\vdash_{\text{func}2} \mathsf{f} : \vec{\mathsf{S}} \longrightarrow \mathsf{T}$ implies $\vdash_{\text{func}1} \mathsf{f} : \vec{\mathsf{S}} \longrightarrow \mathsf{T}$ for every $\mathsf{f} \in \mathcal{C}_2$, $\vec{\mathsf{S}} \in \mathcal{S}_2$, and $\mathsf{T} \in \mathcal{S}_2$.*

Tarski semantics. We define the semantics of first-order terms and formulas, a standard Tarski-semantics, in Fig. 30.2. To define the semantics, we assume

Terms and Formulas

$$\begin{array}{llcll}
\text{Terms} & t, s & ::= & \mathsf{x} \mid \mathsf{f}\,\vec{t} & (\mathsf{f} \in \mathcal{C}) \\
\text{Formulas} & \pi, \chi & ::= & \mathsf{True} \mid \mathsf{False} \mid t \doteq_{\mathsf{S}} s \mid \mathsf{p}\,\vec{t} \mid \pi_1 \mathbin{\dot\wedge} \pi_2 \mid \pi_1 \mathbin{\dot\vee} \pi_2 \mid \pi_1 \mathbin{\dot\Rightarrow} \pi_2 \mid \dot\exists \mathsf{x} : \mathsf{S}.\, \pi \mid \dot\forall \mathsf{x} : \mathsf{S}.\, \pi & (\mathsf{p} \in \mathcal{P}, \mathsf{S} \in \mathcal{S})
\end{array}$$

Well-Sortedness

$$\textsc{type-var}\ \frac{\mathsf{x} : \mathsf{S} \in \Sigma}{\Sigma \vdash_{\mathrm{term}} \mathsf{x} : \mathsf{S}} \qquad \textsc{type-func}\ \frac{\vdash_{\mathrm{func}} \mathsf{f} : \vec{\mathsf{S}} \to \mathsf{T} \qquad \Sigma \vdash_{\mathrm{term}} \vec{t} : \vec{\mathsf{S}}}{\Sigma \vdash_{\mathrm{term}} \mathsf{f}\,\vec{t} : \mathsf{T}} \qquad \textsc{type-true}\ \frac{}{\Sigma \vdash_{\mathrm{form}} \mathsf{True}} \qquad \textsc{type-false}\ \frac{}{\Sigma \vdash_{\mathrm{form}} \mathsf{False}}$$

$$\textsc{type-and}\ \frac{\Sigma \vdash_{\mathrm{form}} \pi_1 \qquad \Sigma \vdash_{\mathrm{form}} \pi_2}{\Sigma \vdash_{\mathrm{form}} \pi_1 \mathbin{\dot\wedge} \pi_2} \qquad \textsc{type-or}\ \frac{\Sigma \vdash_{\mathrm{form}} \pi_1 \qquad \Sigma \vdash_{\mathrm{form}} \pi_2}{\Sigma \vdash_{\mathrm{form}} \pi_1 \mathbin{\dot\vee} \pi_2} \qquad \textsc{type-impl}\ \frac{\Sigma \vdash_{\mathrm{form}} \pi_1 \qquad \Sigma \vdash_{\mathrm{form}} \pi_2}{\Sigma \vdash_{\mathrm{form}} \pi_1 \mathbin{\dot\Rightarrow} \pi_2} \qquad \textsc{type-all}\ \frac{\Sigma, \mathsf{x} : \mathsf{S} \vdash_{\mathrm{form}} \pi}{\Sigma \vdash_{\mathrm{form}} \dot\forall \mathsf{x} : \mathsf{S}.\, \pi}$$

$$\textsc{type-exists}\ \frac{\Sigma, \mathsf{x} : \mathsf{S} \vdash_{\mathrm{form}} \pi}{\Sigma \vdash_{\mathrm{form}} \dot\exists \mathsf{x} : \mathsf{S}.\, \pi} \qquad \textsc{type-eq}\ \frac{\Sigma \vdash_{\mathrm{term}} t : \mathsf{S} \qquad \Sigma \vdash_{\mathrm{term}} s : \mathsf{S}}{\Sigma \vdash_{\mathrm{form}} t \doteq_{\mathsf{S}} s} \qquad \textsc{type-pred}\ \frac{\vdash_{\mathrm{pred}} \mathsf{p} : \vec{\mathsf{S}} \to \mathsf{Prop} \qquad \Sigma \vdash_{\mathrm{term}} \vec{t} : \vec{\mathsf{S}}}{\Sigma \vdash_{\mathrm{form}} \mathsf{p}\,\vec{t}}$$

Figure 30.1: Terms and formulas of first-order logic.

$$\begin{array}{rcl@{\qquad}rcl}
[\![\mathsf{x}]\!]^{M}_{\mathrm{term}}(\epsilon) & \triangleq & \epsilon(\mathsf{x}) & [\![\mathsf{f}\,\vec{t}]\!]^{M}_{\mathrm{term}}(\epsilon) & \triangleq & [\![\mathsf{f}]\!]_{\mathrm{func}}([\![\vec{t}]\!]^{M}_{\mathrm{term}}(\epsilon)) \\
[\![\mathsf{True}]\!]^{M}_{\mathrm{form}}(\epsilon) & \triangleq & \mathit{True} & [\![\mathsf{False}]\!]^{M}_{\mathrm{form}}(\epsilon) & \triangleq & \mathit{False} \\
[\![\pi_1 \mathbin{\dot\wedge} \pi_2]\!]^{M}_{\mathrm{form}}(\epsilon) & \triangleq & [\![\pi_1]\!]^{M}_{\mathrm{form}}(\epsilon) \wedge [\![\pi_2]\!]^{M}_{\mathrm{form}}(\epsilon) & [\![\pi_1 \mathbin{\dot\vee} \pi_2]\!]^{M}_{\mathrm{form}}(\epsilon) & \triangleq & [\![\pi_1]\!]^{M}_{\mathrm{form}}(\epsilon) \vee [\![\pi_2]\!]^{M}_{\mathrm{form}}(\epsilon) \\
[\![\pi_1 \mathbin{\dot\Rightarrow} \pi_2]\!]^{M}_{\mathrm{form}}(\epsilon) & \triangleq & [\![\pi_1]\!]^{M}_{\mathrm{form}}(\epsilon) \Rightarrow [\![\pi_2]\!]^{M}_{\mathrm{form}}(\epsilon) & [\![\mathsf{p}\,\vec{t}]\!]^{M}_{\mathrm{form}}(\epsilon) & \triangleq & [\![\mathsf{p}]\!]_{\mathrm{pred}}([\![\vec{t}]\!]^{M}_{\mathrm{term}}(\epsilon)) \\
[\![\dot\forall \mathsf{x} : \mathsf{S}.\, \pi]\!]^{M}_{\mathrm{form}}(\epsilon) & \triangleq & \forall d : D_{\mathsf{S}}.\, [\![\pi]\!]^{M}_{\mathrm{form}}(\epsilon, \mathsf{x} \mapsto d) & [\![\dot\exists \mathsf{x} : \mathsf{S}.\, \pi]\!]^{M}_{\mathrm{form}}(\epsilon) & \triangleq & \exists d : D_{\mathsf{S}}.\, [\![\pi]\!]^{M}_{\mathrm{form}}(\epsilon, \mathsf{x} \mapsto d) \\
[\![t \doteq_{\mathsf{S}} s]\!]^{M}_{\mathrm{form}}(\epsilon) & \triangleq & [\![t]\!]^{M}_{\mathrm{term}}(\epsilon) \triangleq [\![s]\!]^{M}_{\mathrm{term}}(\epsilon) & & &
\end{array}$$

Figure 30.2: The Tarski semantics for a given model $M = (D_{\_}, [\![\_]\!]_{\mathrm{func}}, [\![\_]\!]_{\mathrm{pred}})$

a model $M = (D_{\_}, [\![\_]\!]_{\mathrm{func}}, [\![\_]\!]_{\mathrm{pred}})$ where D maps each $\mathsf{S} \in \mathcal{S}$ to an inhabited type D_{S}, the *domain* for S; $[\![\_]\!]_{\mathrm{func}}$ maps each function $\vdash_{\mathrm{func}} \mathsf{f} : \vec{\mathsf{S}} \to \mathsf{T} \in \mathcal{C}$ to a meta-level function $[\![\mathsf{f}]\!]_{\mathrm{func}} : \vec{D_{\mathsf{S}}} \to D_{\mathsf{T}}$; and $[\![\_]\!]_{\mathrm{pred}}$ maps each predicate $\vdash_{\mathrm{pred}} \mathsf{p} : \vec{\mathsf{S}} \to \mathsf{Prop} \in \mathcal{P}$ to a meta-level predicate $[\![\mathsf{p}]\!]_{\mathrm{pred}} : \vec{D_{\mathsf{S}}} \to \mathit{Prop}$. We write $M : \Omega$ to indicate that the model M interprets the signature Ω.

Definition 86. *We say a formula π holds in a model $M = (D_{\_}, [\![\_]\!]_{\mathrm{func}}, [\![\_]\!]_{\mathrm{pred}})$, written $M \vDash \pi$, iff. $[\![\pi]\!]^{M}_{\mathrm{form}}(\emptyset)$ holds, where $[\![\pi]\!]^{M}_{\mathrm{form}}(\epsilon)$ is defined in Fig. 30.2.*

Note the distinction between the meta-level logic and the object language: The terms and formulas in Fig. 30.1 are just syntax. We give them meaning via the Tarski-semantics in Fig. 30.2, which interprets every symbol using the corresponding connective from the ambient meta logic.

To ensure in §30.2 that a model assigns the right semantics to the interpreted parts of the logic, we define the notion of model extension $M^1 \sqsupseteq_{\mathrm{model}} M^2$. Model extension $M^1 \sqsupseteq_{\mathrm{model}} M^2$ ensures that, for the part described by the signature of M^2, the domains are in bijection, the function interpretations are the same (up to the bijection), and the predicates are equivalent (up to the bijection):

Sort Interpretations

$$D_{\mathsf{unit}} \triangleq \{()\} \qquad D_{\mathsf{bool}} \triangleq \mathbb{B} \qquad D_{\mathsf{int}} \triangleq \mathbb{Z} \qquad D_{\mathsf{bv}\,n} \triangleq \{m \in \mathbb{Z} \mid 0 \leq m < 2^n\} \qquad \cdots$$

Function Interpretations

$$[\![()]\!]_{\text{func}}() \triangleq () \qquad [\![\mathsf{true}]\!]_{\text{func}}() \triangleq \mathsf{true} \qquad [\![\mathsf{false}]\!]_{\text{func}}() \triangleq \mathsf{true} \qquad [\![n]\!]_{\text{func}}() \triangleq n \qquad [\![+_{\mathsf{int}}]\!]_{\text{func}}(n, m) \triangleq n + m$$

$$[\![\cdot_{\mathsf{int}}]\!]_{\text{func}}(n, m) \triangleq n \cdot m \qquad [\![-_{\mathsf{int}}]\!]_{\text{func}}(n, m) \triangleq n - m \qquad [\![\mathsf{neg}]\!]_{\text{func}}(a) \triangleq \textit{if } a \textit{ then } \mathsf{false} \textit{ else } \mathsf{true}$$

$$[\![\mathsf{xor}_{\mathsf{bv}\,n}]\!]_{\text{func}}(u_1, u_2) \triangleq u_1 \oplus u_2 \qquad [\![==_{\mathsf{bv}\,n}]\!]_{\text{func}}(u_1, u_2) \triangleq \textit{if } u_1 = u_2 \textit{ then } \mathsf{true} \textit{ else } \mathsf{false}$$

$$[\![\mathsf{if}_{\mathsf{S}}]\!]_{\text{func}}(a, d_1, d_2) \triangleq \textit{if } a \textit{ then } d_1 \textit{ else } d_2 \qquad \cdots$$

Predicate Interpretations

$$[\![\leq_{\mathsf{int}}]\!]_{\text{pred}}(n, m) \triangleq n \leq m \qquad [\![<_{\mathsf{int}}]\!]_{\text{pred}}(n, m) \triangleq n < m \qquad \cdots$$

Figure 30.3: The base model M_{base}.

Definition 87. *Let* $\Omega_1 \sqsupseteq_{\text{sig}} \Omega_2$ *where* $\Omega_1 = (\mathcal{S}_1, \mathcal{C}_1, \mathcal{P}_1, \vdash_{\text{func}\,1}, \vdash_{\text{pred}\,1})$ *and* $\Omega_2 = (\mathcal{S}_2, \mathcal{C}_2, \mathcal{P}_2, \vdash_{\text{func}2}, \vdash_{\text{pred}_2})$. *We say a model* $M^1 = (D^1_{\_}, [\![\_]\!]^1_{\text{func}}, [\![\_]\!]^1_{\text{pred}})$ *for signature* Ω_1 *extends a model* $M^2 = (D^2_{\_}, [\![\_]\!]^2_{\text{func}}, [\![\_]\!]^2_{\text{pred}})$ *for signature* Ω_2, *written* $M^1 \sqsupseteq_{\text{model}} M^2$, *iff.*

1. *for every* $\mathsf{S} \in \mathcal{S}_2$, *there is a function* $i_{\mathsf{S}} : D^2_{\mathsf{S}} \to D^1_{\mathsf{S}}$,
2. *for every* $\mathsf{S} \in \mathcal{S}_2$, *there is a function* $r_{\mathsf{S}} : D^1_{\mathsf{S}} \to D^2_{\mathsf{S}}$,
3. *for every* $\mathsf{S} \in \mathcal{S}_2$, i_{S} *and* r_{S} *form a bijection, i.e.,* $\forall (a : D^2_{\mathsf{S}}), (b : D^1_{\mathsf{S}}).\ i_{\mathsf{S}}(a) = b$ *iff.* $a = r_{\mathsf{S}}(b)$,
4. *for every* $\vdash_{\text{func}} \mathsf{f} : \vec{\mathsf{S}} \to \mathsf{T} \in \mathcal{C}_2$, *we have* $\forall \vec{a} : \vec{D}_{\mathsf{S}}.\ r_{\mathsf{T}}([\![\mathsf{f}]\!]^1_{\text{func}}(i_{\vec{\mathsf{S}}}(\vec{a}))) = [\![\mathsf{f}]\!]^2_{\text{func}}(\vec{a})$,
5. *for every* $\vdash_{\text{pred}_2} \mathsf{p} : \vec{\mathsf{S}} \to \mathit{Prop} \in \mathcal{P}_2$, *we have* $\forall \vec{a} : \vec{D}_{\mathsf{S}}.\ [\![\mathsf{p}]\!]^1_{\text{pred}}(i_{\vec{\mathsf{S}}}(\vec{a}))$ *iff.* $[\![\mathsf{p}]\!]^2_{\text{pred}}(\vec{a})$.

30.1.2 Interpreted Theories

To be sound, we have to make sure that we agree with the SMT solver on the interpreted theories of the logic (*i.e.*, unit, Booleans, bitvectors, and integers). To this end, we define the *base signature* Ω_{base} (and *base model* M_{base} below):

$$\begin{array}{llll}
\text{Sorts} & \mathsf{S}, \mathsf{T} & ::= & \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{int} \mid \mathsf{bv}\,n \mid \cdots \\
\text{Functions} & \mathsf{f}, \mathsf{g} & ::= & () \mid \mathsf{true} \mid \mathsf{false} \mid n \mid +_{\mathsf{int}} \mid \cdot_{\mathsf{int}} \mid -_{\mathsf{int}} \mid \mathsf{neg} \mid \mathsf{xor}_{\mathsf{bv}\,n} \\
 & & \mid & ==_{\mathsf{bv}\,n} \mid \mathsf{if}_{\mathsf{S}} \mid \cdots \\
\text{Predicates} & \mathsf{p}, \mathsf{q} & ::= & \leq_{\mathsf{int}} \mid <_{\mathsf{int}} \mid \cdots
\end{array}$$

The signature introduces several standard base sorts such as integers int and bit vectors bv n, along with canonical functions and predicates on them (*e.g.*, addition on integers, or comparison on integers, *etc.*). (Here, "$\cdots$" stands for additional entries in the base signature such as additional first-order functions.) All functions and predicates have the obvious types. We then define a base model M_{base}, a standard interpretation for these sorts, functions, and predicates, depicted in Fig. 30.3. Notably, the base model *does not interpret or constrain*

$$\lfloor \phi \rfloor(h) \triangleq \phi \qquad \lfloor e \Downarrow v \rfloor(h) \triangleq (e, h) \rightsquigarrow^*_{\mathrm{det}} (v, h) \qquad \lfloor \ell \mapsto_{\mathrm{u}} v \rfloor(h) \triangleq h(\ell) = v$$

$$\lfloor F \land G \rfloor(h) \triangleq \lfloor F \rfloor(h) \land \lfloor G \rfloor(h) \qquad \lfloor F \lor G \rfloor(h) \triangleq \lfloor F \rfloor(h) \lor \lfloor G \rfloor(h)$$

$$\lfloor F \Rightarrow G \rfloor(h) \triangleq \lfloor F \rfloor(h) \Rightarrow \lfloor G \rfloor(h) \qquad \lfloor \forall x.\, F\, x \rfloor(h) \triangleq \forall x.\, \lfloor F\, x \rfloor(h)$$

$$\lfloor \exists x.\, F\, x \rfloor(h) \triangleq \exists x.\, \lfloor F\, x \rfloor(h)$$

Figure 30.4: The mapping $\lfloor \_ \rfloor(\_) : hProp \to Heap \to Prop$ from *hProp*-assertions F to pure assertions $\lfloor F \rfloor(h)$.

the uninterpreted parts of the logic (*i.e.,* uninterpreted sorts, uninterpreted functions, and uninterpreted predicates).

With the base model M_{base} in hand, we can then define the validity judgment $\vDash \pi$ as π holds true *in all models extending the base model*:

Definition 88. *Let Ω_{smt} be a signature such that $\Omega_{\mathrm{smt}} \sqsupseteq_{\mathrm{sig}} \Omega_{\mathrm{base}}$. We define*

$$\vDash \pi \triangleq \forall (M : \Omega_{\mathrm{smt}}).\, M \sqsupseteq_{\mathrm{model}} M_{\mathrm{base}} \Rightarrow M \vDash \pi$$

30.2 From First-Order Logic to Iris, Step by Step

Let us now show the main correspondence:

Theorem 84.
If $\vDash \pi$ holds, then $(\langle \pi \rangle^{\emptyset}_{\mathrm{F}} \Rightarrow \mathbf{wp}\ e\ \{v.\, Q(v)\}) \vdash \mathbf{wp}\ e\ \{v.\, Q(v)\}$ holds in Iris.

Considering the definition of validity $\vDash \pi$ (see Definition 88), the proof consists of four parts: (1) we define a first-order logic model M_{dyn} to interpret the first-order logic (§30.2.1); (2) we prove it extends the base model M_{base} (§30.2.2); (3) we use the resulting assumption $M_{\mathrm{dyn}} \vDash \pi$ to relate π and $\langle \pi \rangle^{\emptyset}_{\mathrm{F}}$ (§30.2.3); and (4) we obtain the Iris entailment $(\langle \pi \rangle^{\emptyset}_{\mathrm{F}} \Rightarrow \mathbf{wp}\ e\ \{v.\, Q(v)\}) \vdash \mathbf{wp}\ e\ \{v.\, Q(v)\}$ (§30.2.4). An essential part in all of the steps will be the translation $\lfloor F \rfloor(h)$ from §28 to bridge between almost-pure assertions (*e.g.,* $\langle \pi \rangle^{\emptyset}_{\mathrm{F}} : hProp$) and actually-pure, meta-level assertions used in validity definition (*e.g.,* $M \vDash \pi : Prop$). For convenience, we have restated it in Fig. 30.4.

30.2.1 A Model for Heap-Dependent Expression Assertions

We construct the model M^h_{dyn} for an arbitrary, but fixed heap h. To indicate the dependency on h, we write M^h_{dyn}. To define, M^h_{dyn} we must choose domains $D_{\_}$, predicate interpretations $[\![\_]\!]_{\mathrm{pred}}$, and function interpretations $[\![\_]\!]_{\mathrm{func}}$. We pick them as follows:

$$D_{\mathsf{S}} \triangleq \{v \mid \lfloor v \in \mathcal{V}[\![\langle \mathsf{S} \rangle_{\mathsf{S}}]\!] \rfloor(h)\} \qquad [\![\mathsf{p}]\!]_{\mathrm{pred}}(\vec{v}) \triangleq \lfloor \langle \mathsf{p} \rangle_{\mathsf{P}}(\vec{v}) \rfloor(h)$$

$$[\![\mathsf{f}]\!]_{\mathrm{func}}(\vec{v}) \triangleq \mathrm{eval}(\langle \mathsf{f} \rangle_{\mathsf{C}}, h)(\vec{v})$$

That is, we instantiate the domains with sets of values v, where—using the mapping $\lfloor \_ \rfloor(h)$ that turns *hProp*-assertions into pure assertions—we know the value v is in the logical relation at type $\langle \mathsf{S} \rangle_{\mathsf{S}}$.[1] For the predicates p, we use the *hProp*-predicates obtained from the translation $\langle \_ \rangle$ and turn them into meta-level assertions using $\lfloor \_ \rfloor(h)$. For functions, we have to turn λ_{dyn}-functions into actual meta-level functions from (well-typed) values to (well-typed) values.

[1] Since first-order domains must be non-empty, we only do so for sorts where the corresponding value relation $\mathcal{V}[\![\langle \mathsf{S} \rangle_{\mathsf{S}}]\!]$ is non-empty.

To this end, we prove Lemma 89. It turns a well-typed λ_{dyn}-function f into a meta-level function $\mathrm{eval}(f, h)$ (for a fixed heap h): We use it (generalized to arbitrary function arities) to define $[\![\mathsf{f}]\!]_{\mathrm{func}}$ by picking $f \triangleq \langle \mathsf{f} \rangle_{\mathsf{C}}$.

Lemma 89. *Let* $\vdash f \in \mathcal{V}[\![\tau_1 \to \tau_2]\!]$. *For any* h*, there exists a function* $\mathrm{eval}(f, h) :$ *Val* $\to$ *Val such that for all values* v *where* $\lfloor v \in \mathcal{V}[\![\tau_1]\!] \rfloor(h)$*, we have* $(f\ v, h) \rightsquigarrow^*_{det}$ $(\mathrm{eval}(f, h)(v), h)$ *and* $\lfloor \mathrm{eval}(f, h)(v) \in \mathcal{V}[\![\tau_2]\!] \rfloor(h)$.

Proof Sketch. Using Lemma 81, we first obtain the pure, meta-level assertion $\lfloor f \in \mathcal{V}[\![\tau_1 \to \tau_2]\!] \rfloor(h)$. Unfolding the definition of $\lfloor \_ \rfloor(h)$ (see Fig. 30.4), we get

$$\forall v.\ \lfloor v \in \mathcal{V}[\![\tau_1]\!] \rfloor(h) \Rightarrow \exists w.\ (f\ v, h) \rightsquigarrow^*_{\mathrm{det}} (w, h) \wedge \lfloor w \in \mathcal{V}[\![\tau_2]\!] \rfloor(h)$$

The value w is the desired result of f. We extract it using the axiom of choice. □

30.2.2 Proving Model Extension

Having defined the model, we must then show that it is an extension of the base model M_{base}:

Lemma 90. $M^h_{\mathrm{dyn}} \sqsupseteq_{\mathrm{model}} M_{\mathrm{base}}$ *for any heap* h.

Proof Sketch. Recall that the signature of M_{base} is Ω_{base}. For the sorts S in Ω_{base}, it is straightforward to construct a bijection between the domains D_{S} of the base model M_{base} and well-typed values in our semantic types $\mathcal{V}[\![\langle \mathsf{S} \rangle_{\mathsf{S}}]\!]$. For example, the type of mathematical integers $\mathbb{Z}$ is clearly in bijection with the type of well-typed λ_{dyn}-integers $D_{\mathsf{int}} = \{v \mid \lfloor v \in \mathcal{V}[\![\mathsf{int}]\!] \rfloor(h)\}$ (since, as defined in Fig. 28.3, $\mathcal{V}[\![\mathsf{int}]\!] = \{n \mid n \in \mathbb{Z}\}$). The remaining equivalences of $M^h_{\mathrm{dyn}} \sqsupseteq_{\mathrm{model}} M_{\mathrm{base}}$ then follow by case analysis on our interpreted functions and predicates. Here, we must make sure that the implementation of, *e.g.*, multiplication $\langle \cdot_{\mathsf{int}} \rangle_{\mathsf{C}} \triangleq \lambda xy.\, x * y$ (a *value* in λ_{dyn}; see Fig. 28.5) matches (up to the bijection) the semantics of multiplication on integers, *i.e.*, $[\![\cdot_{\mathsf{int}}]\!]_{\mathrm{func}}(n, m) \triangleq n \cdot m$ (in ordinary mathematics; see Fig. 30.3). □

30.2.3 Connecting to the Translation

Using the first-order logic model M^h_{dyn}, we now show that valid formulas π imply their translation $\langle \pi \rangle_{\mathsf{F}}$ at the meta level if we turn them from an *almost-pure* assertion into an *actually-pure* assertion with $\lfloor \_ \rfloor(h)$:

Lemma 91. *If* $\vDash \pi$*, then* $\lfloor \langle \pi \rangle^{\emptyset}_{\mathsf{F}} \rfloor(h)$ *for any heap* h.

Proof. Fix a heap h. We instantiate $\vDash \pi$ with M^h_{dyn}, which extends the base model M_{base} (Lemma 90). Thus, we obtain $[\![\pi]\!]^{M^h_{\mathrm{dyn}}}_{\mathrm{form}}(\emptyset)$. With Lemma 93 (below) it follows that $\lfloor \langle \pi \rangle^{\emptyset}_{\mathsf{F}} \rfloor(h)$ holds. □

The crux of this lemma is that we want to go from $[\![\pi]\!]^{M^h_{\mathrm{dyn}}}_{\mathrm{form}}(\emptyset)$ to the *hProp*-translation of π. To state the desired lemma formally, we need to generalize over non-empty variable mappings. To this end, we relate a first-order variable mapping ϵ (mapping variables to elements of D_{S} in M^h_{dyn}) to a variable substitution γ (mapping variables to values) with:

$$\mathrm{agree}(\Sigma, \gamma, \epsilon) \triangleq \forall \mathsf{x} : \mathsf{S} \in \Sigma.\ \exists v.\ \gamma(\mathsf{x}) = v \wedge \epsilon(\mathsf{x}) = v$$

We then first show that the semantics of terms in the model $[\![\pi]\!]^{M^h_{\mathsf{dyn}}}_{\mathsf{form}}(\emptyset)$ corresponds to the operational semantics in λ_{dyn}:

Lemma 92. *Let $\Sigma \vdash_{\mathsf{term}} t : \mathsf{S}$. If $\mathsf{agree}(\Sigma, \gamma, \epsilon)$, then for every heap h,*

$$[\![t]\!]^{M^h_{\mathsf{dyn}}}_{\mathsf{term}}(\epsilon) = v \quad \textit{iff.} \quad (\langle t \rangle^{\gamma}_{\mathsf{T}}, h) \rightsquigarrow^*_{det} (v, h)$$

Proof. By induction on $\Sigma \vdash_{\mathsf{term}} t : \mathsf{S}$. □

Then we use this result to show the equivalence:

Lemma 93. *Let $\Sigma \vdash_{\mathsf{form}} \pi$. If $\mathsf{agree}(\Sigma, \gamma, \epsilon)$, then for every heap h,*

$$[\![\pi]\!]^{M^h_{\mathsf{dyn}}}_{\mathsf{form}}(\epsilon) \quad \textit{iff.} \quad \lfloor \langle \pi \rangle^{\gamma}_{\mathsf{F}} \rfloor(h)$$

Proof. By induction on $\Sigma \vdash_{\mathsf{form}} \pi$, using Lemma 92. □

30.2.4 Importing the Fact into Iris

Recall the theorem we wish to prove:

If $\vDash \pi$ holds, then $(\langle \pi \rangle^{\emptyset}_{\mathsf{F}} \Rightarrow \mathbf{wp}\ e\ \{v.\, Q(v)\}) \vdash \mathbf{wp}\ e\ \{v.\, Q(v)\}$ holds in Iris.

With Lemma 91, we have now a way to deduce $\lfloor \langle \pi \rangle^{\emptyset}_{\mathsf{F}} \rfloor(h)$ from $\vDash \pi$ for any heap h. But while $\lfloor \langle \pi \rangle^{\emptyset}_{\mathsf{F}} \rfloor(h)$ already uses the translation "$\langle \pi \rangle^{\emptyset}_{\mathsf{F}}$", it is still a meta-level assertion. Thus, let us now transition from the meta level to Iris.

The key to doing so is the equivalence from Lemma 81 (*i.e.,* relating almost-pure and actually-pure assertions). It—*almost*—says that if $\lfloor \langle \pi \rangle^{\emptyset}_{\mathsf{F}} \rfloor(h)$ holds at the meta level, then $\langle \pi \rangle^{\emptyset}_{\mathsf{F}}$ holds in Iris. The only thing missing is the precondition that it imposes, namely $\mathsf{heap_u}(h) * (\mathop{\Huge *}_{\ell \mapsto v \in h} \ell \mapsto_{\mathsf{u}} v)$, requiring unstable resources for the entire current heap. We call this precondition the *ambient heap* below:

$$\mathsf{ambient}(h) \triangleq \mathsf{heap_u}(h) * (\mathop{*}_{\ell \mapsto v \in h} \ell \mapsto_{\mathsf{u}} v)$$

Fortunately, whenever we prove a weakest precondition, we can always "get a copy" of the ambient heap (from the state interpretation SI). Concretely, one can prove the following property for the weakest precondition of λ_{dyn}:

get-ambient-heap
$$(\forall h.\, \mathsf{ambient}(h) \mathrel{-\!\!*} \mathbf{wp}\ e\ \{v.\, Q(v)\}) \vdash \mathbf{wp}\ e\ \{v.\, Q(v)\}$$

Thus, we can prove the key theorem:

Theorem 84.
If $\vDash \pi$ holds, then $(\langle \pi \rangle^{\emptyset}_{\mathsf{F}} \Rightarrow \mathbf{wp}\ e\ \{v.\, Q(v)\}) \vdash \mathbf{wp}\ e\ \{v.\, Q(v)\}$ holds in Iris.

Proof. Let $\vDash \pi$. With get-ambient-heap, we can assume the ambient heap. We are left to prove:

$$(\langle \pi \rangle^{\emptyset}_{\mathsf{F}} \Rightarrow \mathbf{wp}\ e\ \{v.\, Q(v)\}) * \mathsf{ambient}(h) \vdash \mathbf{wp}\ e\ \{v.\, Q(v)\}$$

for all heaps h. Using the ambient heap, by Lemma 81, it then suffices to prove

$$(\langle \pi \rangle^{\emptyset}_{\mathsf{F}} \Rightarrow \mathbf{wp}\ e\ \{v.\, Q(v)\}) * (\langle \pi \rangle^{\emptyset}_{\mathsf{F}} \Leftrightarrow \lfloor \langle \pi \rangle^{\emptyset}_{\mathsf{F}} \rfloor(h)) \vdash \mathbf{wp}\ e\ \{v.\, Q(v)\}$$

By Lemma 91 and $\vDash \pi$, we know $\lfloor \langle \pi \rangle^{\emptyset}_{\mathsf{F}} \rfloor(h)$ holds true for any heap h. Thus, we can use the equivalence $\langle \pi \rangle^{\emptyset}_{\mathsf{F}} \Leftrightarrow \lfloor \langle \pi \rangle^{\emptyset}_{\mathsf{F}} \rfloor(h)$ to establish $\langle \pi \rangle^{\emptyset}_{\mathsf{F}}$ and hence $\mathbf{wp}\ e\ \{v.\, Q(v)\}$. This concludes the proof. □

Chapter 31

Related Work

Implicit dynamic frames. There is a long line of work on building automated verifiers based on implicit dynamic frames[1] culminating in the work around Viper.[2] This work aims to build practical verification tools with a focus on automation. Compared to them, Daenerys is less automated (*e.g.,* we manually query the SMT solver) but is—by nature of being connected to Iris—more expressive (*e.g.,* supporting higher-order functions and impredicative invariants). See §29.2 for a more detailed comparison.

We discuss the two most closely related pieces of work that *do* focus on the meta theory of IDF. Parkinson and Summers[3] show how to encode SL in IDF. They define an umbrella logic with a total-heap semantics—similar to a standard model for IDF—and then show that it has the intended meaning for SL assertions (stopping short of an SL program logic). We consider the opposite direction: we integrate IDF into a very expressive SL, Iris. In doing so, we define an encoding of IDF based on SL resources, which could be understood as a partial-heap semantics for IDF.

Dardinier et al.[4] provide a foundational approach for showing the soundness of verifiers based on intermediate verification languages. They extend a variant of resource algebras with stable and unstable projections and instantiate it with a subset of Viper called ViperCore. However, their notion of resource algebras is less expressive than our Iris-based resource algebras (*e.g.,* no step-indexing, and no persistency) and their work does not address advanced features (*e.g.,* frame preserving updates, heap-dependent functions, predicates, or impredicative invariants). Thus, they do not cover most of the examples described in §29 (see Fig. 29.3).

Separation logic with unstable resources. There is a long line of work on separation logics with unstable resources focused on fine-grained concurrency verification.[5] None of them use unstable assertions about heaps or program expressions (*i.e.,* no HDEAs). Instead, they use unstable resources to unstably assert the current *logical state* s of a concurrent program. At a technical level, an interesting difference is that they take the transition system $s \longrightarrow s'$ as the primitive and then derive their notion of "stable resources" w.r.t. it. In contrast, for us, stability is a primitive notion of the resource algebras (see §26) and then we derive our updates $a \rightsquigarrow_{\mathsf{st}} b$ from it.

Charguéraud and Pottier[6] develop a read-only modality $\mathsf{RO}(P)$ that temporarily gives read-only access to the memory described by P. Like our unstable points-to $\ell \mapsto_{\mathsf{u}} v$, their read-only modality is freely duplicable, *i.e.,* $\mathsf{RO}(P) \vdash \mathsf{RO}(P) * \mathsf{RO}(P)$. However, read-only access and read-write access

[1] Smans, Jacobs, and Piessens, "Implicit Dynamic Frames: Combining dynamic frames and separation logic", 2009 [SJP09]; Leino and Müller, "A basis for verifying multi-threaded programs", 2009 [LM09]; Wise et al., "Gradual verification of recursive heap data structures", 2020 [Wis+20].

[2] Müller, Schwerhoff, and Summers, "Viper: A verification infrastructure for permission-based reasoning", 2017 [MSS17]; Astrauskas et al., "Leveraging Rust types for modular specification and verification", 2019 [Ast+19]; Wolf et al., "Gobra: Modular specification and verification of Go programs", 2021 [Wol+21]; Eilers and Müller, "Nagini: A static verifier for Python", 2018 [EM18].

[3] Parkinson and Summers, "The relationship between separation logic and implicit dynamic frames", 2011 [PS11].

[4] Dardinier et al., "Formal foundations for translational separation logic verifiers", 2025 [Dar+25].

[5] Dinsdale-Young et al., "Concurrent abstract predicates", 2010 [Din+10]; Svendsen, Birkedal, and Parkinson, "Modular reasoning about separation of concurrent data structures", 2013 [SBP13]; Svendsen and Birkedal, "Impredicative concurrent abstract predicates", 2014 [SB14]; Nanevski et al., "Communicating state transition systems for fine-grained concurrent resources", 2014 [Nan+14]; Raad, Villard, and Gardner, "CoLoSL: Concurrent local subjective logic", 2015 [RVG15]; Dinsdale-Young et al., "Views: Compositional reasoning for concurrent programs", 2013 [Din+13]; Rocha Pinto, Dinsdale-Young, and Gardner, "TaDA: A logic for time and data abstraction", 2014 [RDG14]; D'Osualdo et al., "TaDA Live: Compositional reasoning for termination of fine-grained concurrent programs", 2021 [DOs+21].

[6] Charguéraud and Pottier, "Temporary read-only permissions for separation logic", 2017 [CP17].

are temporally disjoint: in their work, one can either own $\mathrm{RO}(\ell \mapsto v)$ or $\ell \mapsto v$, but not both at the same time. Thus, their approach cannot support HDEAs: For HDEAs, it is crucial that $\ell \mapsto_{\mathrm{u}} v$ holds at the same time as $\ell \mapsto v$ (*e.g.*, for $\ell \mapsto v_{\mathrm{vec}} \vdash \ell \mapsto v_{\mathrm{vec}} * e_{\mathrm{add}} \Downarrow 42$ in §25.1).

Building on the work of Charguéraud and Pottier, Gospel[7] is a separation logic-based specification language for OCaml that allows leaving the mathematical model of an abstract predicate implicit and thus enables a form of incremental verification. However, Gospel still requires one to fix a final mathematical abstraction up-front, providing a different kind of incrementality than HDEAs. In particular, as demonstrated in case study #3 in §29.2, HDEAs do *not* require fixing a mathematical abstraction up-front. They allow one to incrementally add abstractions by considering additional functions (*e.g.*, `len`, `nth`, *etc.*) that expose different information about the data structure.

[7] Charguéraud et al., "GOSPEL - Providing OCaml with a formal specification language", 2019 [Cha+19b].

Iris. Daenerys alters Iris at a fundamental level by introducing unstable resources into its resource model and altering the frame rule. However, as mentioned in §29, for many existing Iris proofs with only stable resources, the presence of instability should not introduce a noteworthy proof overhead (*e.g.*, we have backported several existing Iris proofs by changing only a few lines).

Vindum, Georges, and Birkedal[8] develop a nextgen update modality $\dot{\leftrightarrow}^{t} P$ in Iris that—like our update $\Rrightarrow_{\mathtt{st}}$—does not preserve (all) frames. Their modality—unlike Iris's $\Rrightarrow P$ and our $\Rrightarrow_{\mathtt{st}} P$—is not centered around the concept of (stable) frame preservation. Instead, it applies a user-specified function t to the current resource (without being concerned with frame preservation). Also unlike us, they do not modify the notion of resources in Iris and do not consider HDEAs.

[8] Vindum, Georges, and Birkedal, "The Nextgen modality: A modality for non-frame-preserving updates in separation logic", 2025 [VGB25].

One goal of Daenerys is to improve automation for Iris via HDEAs, laying the foundation for integrating SMT solvers. This is orthogonal to other lines of work on automation for Iris,[9] which focus on different directions for automation (*e.g.*, automating resource manipulation in concurrent programs). BFF[10] and Katamaran[11] provide specialized bitvector solvers. While these solvers provide end-to-end foundational proofs, they are not as powerful as the bitvector support of state-of-the-art SMT solvers. For example, they cannot handle the Popcount 32-bit integer case study in group #2 from §29.2.

[9] Sammler et al., "RefinedC: Automating the foundational verification of C code with refined ownership types", 2021 [Sam+21]; Zhu et al., "BFF: Foundational and automated verification of bitfield-manipulating programs", 2022 [Zhu+22]; Mulder, Krebbers, and Geuvers, "Diaframe: Automated verification of fine-grained concurrent programs in Iris", 2022 [MKG22]; Keuchel et al., "Verified symbolic execution with Kripke specification monads (and no meta-programming)", 2022 [Keu+22].

[10] Zhu et al., "BFF: Foundational and automated verification of bitfield-manipulating programs", 2022 [Zhu+22].

[11] Keuchel et al., "Verified symbolic execution with Kripke specification monads (and no meta-programming)", 2022 [Keu+22].

SMT solvers and proof assistants. There are several approaches that aim to integrate the automation of SMT solvers into foundational proof assistants.[12] These approaches address a problem that is orthogonal to the results of this part of the dissertation: They focus on reflecting proof artifacts of an SMT solver into a proof assistant, while Daenerys shows how one can leverage (potentially reflected) results from an SMT solver to reason about heap-accessing programs.

[12] Blanchette, Böhme, and Paulson, "Extending Sledgehammer with SMT solvers", 2013 [BBP13]; Ekici et al., "SMTCoq: A plug-in for integrating SMT solvers into Coq", 2017 [Eki+17].

Chapter 32

Conclusion and Future Work

The goal of this dissertation was (1) to make Iris more expressive by *generalizing step-indexing* and (2) to reduce its verification overhead by *increasing automation.* To this end, the dissertation has presented four projects, two for step-indexing and two for automation. We review the contributions of each project and outline directions for future work.

32.1 Generalizing Step-Indexing

Transfinite Iris. In **Transfinite Iris** (Part II), we changed the step-indexing technique underlying Iris to support proving liveness properties. Concretely, by using ordinals as step-indices instead of natural numbers in the model of Iris (see §9), we obtained *the existential property*, a high-level reasoning principle that can be used to hoist existential quantifiers out of the logic. Leveraging the existential property, we have then developed a liveness logic on top of the new model for proving two liveness properties—termination-preserving refinements (§7.1) and termination (§7.2)—and applied it to several case studies. The price of the transfinite model is the loss of two commuting rules of Iris (§9.2). To explore the consequences of their absence, we have adapted the safety logic of Iris to Transfinite Iris (discussed briefly in §9.2) and used it to recover several examples originally verified in Iris.

A natural next step for Transfinite Iris would be the extension to the concurrent setting. That is, while the safety logic of Transfinite Iris handles concurrency, the liveness logic presented in this dissertation only supports proving liveness properties of *sequential* programs. To generalize the liveness logic to concurrency, the main missing aspect is *fairness.* More specifically, there is currently nothing at the separation logic level that prevents one from generalizing the definition of the Transfinite Iris weakest precondition (see Fig. 7.6) to multiple threads. However, liveness reasoning in a concurrent setting typically requires the assumption of fairness of the underlying scheduler (*e.g.*, to prove termination of a program with multiple threads). Transfinite Iris currently has no notion of fairness. To integrate fairness, a particularly interesting direction for future work would be to combine Transfinite Iris with Gäher et al.'s Simuliris,[1] a framework for verifying compiler transformations (in concurrent languages) by proving fair termination-preserving refinements in Iris. To support termination-preservation, Gäher et al. eschew step-indexing and work in a fragment of Iris without the later modality, impredicative invariants, and guarded fixpoints. In contrast, with Transfinite Iris as a foundation, one would potentially unlock the full range of step-indexed features and open the door to

[1] Gäher et al., "Simuliris: A separation logic framework for verifying concurrent program optimizations", 2022 [Gäh+22].

verifying compiler transformations that involve advanced language features such as higher-order state.

A different direction would be the use of Transfinite Iris for reasoning about *angelic non-determinism.* That is, since the original publication of Transfinite Iris, the base logic of Transfinite Iris has been used by Guéneau et al.[2] to build a program logic for reasoning about the interoperability between idealized versions of OCaml and C. In their work, Guéneau et al. do not consider liveness properties. However, they still make use of the existential property of Transfinite Iris, because their underlying operational semantics uses both angelic and demonic non-determinism. The use of angelic non-determinism results in existential quantifiers in the definition of their weakest precondition (similar to the weakest precondition in Fig. 7.6), and the existential property allows them to hoist these existential quantifiers out of the weakest precondition. An interesting direction for future work would be to use Transfinite Iris to build more general program logics for languages with angelic non-determinism. For example, it would be particularly interesting to integrate Transfinite Iris with DimSum,[3] a framework based on Iris for proving refinements between multi-language programs, which crucially relies on angelic non-determinism.

[2] Guéneau et al., "Melocoton: A program logic for verified interoperability between OCaml and C", 2023 [Gué+23].

[3] Sammler et al., "DimSum: A decentralized approach to multi-language semantics and verification", 2023 [Sam+23].

Lastly, since the original development of Transfinite Iris was published, it has been an ongoing process to share more of the Rocq implementation between regular Iris and Transfinite Iris. Completing this process would make Transfinite Iris easily available for the broader Iris community.

Later Credits. With **Later Credits** (Part III), we introduced a new, amortized approach to step-indexing. Concretely, we introduced (in §12) a resource that can be used to justify later eliminations after the corresponding physical step has happened, and this resource is subject to all the traditional separation logic reasoning principles (*e.g.,* framing, sharing via invariants, *etc.*). As we have discussed in §15, the approach works both in regular Iris and in Transfinite Iris. It can be used to simplify existing proofs (*e.g.,* by using later credits for helping when proving logical atomicity; see §13.2) and to develop new proofs (*e.g.,* by using later credits for proving reordering refinements; see §13.1).

Since the original development of later credits, later credits have been integrated into the main Iris distribution, and they are nowadays commonly used in the Iris community. In particular, Gäher et al.[4] use them to model parts of the Rust type system, and Chang et al.[5] use them to verify a high-performance transaction library using multi-version concurrency control.

Later credits have also found applications beyond Iris: a variant of them has been integrated into the automated verifier Verus.[6] In Verus, they are required whenever one wants to open an invariant to ensure soundness of the invariant mechanism. That is, while Verus in its current form only allows simple invariants (akin to the timeless invariants in §3.4), the Verus developers are preparing for more powerful, higher-order invariants similar to Iris's impredicative invariants (see §3.4.1). With these invariants and without later credits, Verus would then admit paradoxes similar to the paradoxes that exist for Iris's invariants[7] (*e.g.,* it is unsound to open a non-timeless invariant as part of a ghost update and get direct access to the contents without a guarding later). In Iris, these kinds of paradoxes are prevented via the later modality. In Verus, however, there is no later modality, because it is based on the Rust type

[4] Gäher et al., "RefinedRust: A type system for high-assurance verification of Rust programs", 2024 [Gäh+24].

[5] Chang et al., "Verifying vMVCC, a high-performance transaction library using multi-version concurrency control", 2023 [Cha+23].

[6] Lattuada et al., "Verus: Verifying Rust programs using linear ghost types", 2023 [Lat+23].

[7] See §8.2 in Jung et al., "Iris from the ground up: A modular foundation for higher-order concurrent separation logic", 2018 [Jun+18b].

system. Instead, inspired by later credits, Lorch et al.[8] have added a form of "open-invariant credits" to guard against unsoundness of Verus in the future. To open an invariant, one has to spend a credit, and to obtain a credit, one has to take a physical step in the Rust program.

[8] Lorch et al., *Require open-invariant credits*, 2024 [Lor+24].

32.2 Increasing Automation

Quiver. With **Quiver** (Part IV), we have introduced the approach of abductive deductive verification (§18) and applied it to specification inference of C programs, using the abduction engine Argon (§19) and the separation logic based type system Thorium (§20). In particular, we applied Quiver to several examples, including a vector implementation and code from popular open source libraries.

In the future, it would be interesting to apply abductive deductive verification, and in particular Argon, to other languages and flavors of separation logic.

Moreover, it would be interesting to investigate *loop invariant inference.* That is, finding loop invariants in separation logic is a non-trivial task: it requires finding pure invariants and, additionally, invariants about resources. For *restricted fragments* of separation logic, loop invariant inference techniques have been developed.[9] But for rich separation logics like the one targeted by Quiver, no loop invariant inference algorithms are known. Thus, like Quiver, deductive verification tools for expressive separation logics (*e.g.,* VeriFast,[10] CN,[11] Viper,[12] and RefinedC[13]) require user-provided loop invariants. It would be interesting to investigate how to integrate (a) existing loop invariant inference algorithms for separation logic when the invariant falls into a supported fragment and (b) existing *non-separation logic* loop invariant inference techniques by requiring loop invariant sketches (for the resources) but leaving holes for the pure invariants.

[9] Magill et al., "Inferring invariants in separation logic for imperative list-processing programs", 2006 [Mag+06]; Calcagno et al., "Compositional shape analysis by means of bi-abduction", 2009 [Cal+09]; He et al., "Automated specification discovery via user-defined predicates", 2013 [He+13]; Holík et al., "Low-level bi-abduction", 2022 [Hol+22].

[10] Jacobs et al., "VeriFast: A powerful, sound, predictable, fast verifier for C and Java", 2011 [Jac+11].

[11] Pulte et al., "CN: Verifying systems C code with separation-logic refinement types", 2023 [Pul+23].

[12] Müller, Schwerhoff, and Summers, "Viper: A verification infrastructure for permission-based reasoning", 2017 [MSS17].

[13] Sammler et al., "RefinedC: Automating the foundational verification of C code with refined ownership types", 2021 [Sam+21].

Daenerys. With **Daenerys** (Part V), we have brought heap-dependent expression assertions to Iris in the form of our evaluation assertion $e \Downarrow v$. To do so, we made a fundamental change to the model of Iris (§26): we added unstable resources such as $\ell \mapsto_{\mathrm{u}} v$, and we introduced the frame modality $\boxplus P$ to govern framing. On top of the new model, we then developed a program logic (§27) and connected it to first-order logic via our heap-dependent expression assertions (§28). Leveraging the connection to first-order logic, we then used an SMT solver to automate parts of our proofs (*e.g.,* bitvector reasoning in the population count) in our case studies (§29).

Since we currently manually query the SMT solver in Daenerys, one natural direction for future work would be to develop a verification tool on top of Daenerys (*e.g.,* with a frontend, automation for reasoning about separation logic resources, and an automated translation to SMT). A related, but slightly different direction, would be to integrate our heap-dependent expression assertions into existing Iris verification tools like the C verification tool RefinedC.[14] Specifically for RefinedC, one would have to build its separation logic type system on top of Daenerys, which could have interesting effects on the typing rules (*e.g.,* proof obligations arising from the frame modality in the frame rule). Moreover, since the semantics of C can be nontrivial at times (and heavily uses mutation), it may be worthwhile to consider a *different* language for the assertion level than

[14] Sammler et al., "RefinedC: Automating the foundational verification of C code with refined ownership types", 2021 [Sam+21].

for the implementation (*e.g.,* a functional one that connects well with first-order logic). Traditionally, the assertion-level language and the implementation-level language are the same for implicit dynamic frames based approaches. However, even for different languages, heap-dependent expression assertions could prove useful, since they provide a convenient way to refer to program memory at the assertion level—including via functions operating on it.

Moreover, another interesting direction for future work would be to develop first-order verification conditions on top of Iris—especially considering that as part of Daenerys, we have already developed a formal semantics for multi-sorted first-order logic in Rocq. That is, many automated verifiers, including Viper,[15] Verus,[16] and Dafny[17] reduce verification to checking a first-order formula for satisfiability in an SMT solver, a so-called verification condition. Typically, the translation from the original program to the resulting verification condition is trusted. It would be interesting to prove soundness of a verification condition generation approach—either for (a subset of) an existing approach such as Viper or for a new verification condition generation strategy on top of Iris. To connect first-order verification conditions (which are checked by an SMT solver) to a foundational framework based on Iris in Rocq, the multi-sorted first-order logic of Daenerys could be a useful starting point—it could be used to bridge the gap between SMT solvers and Rocq.

[15] Müller, Schwerhoff, and Summers, "Viper: A verification infrastructure for permission-based reasoning", 2017 [MSS17].

[16] Lattuada et al., "Verus: Verifying Rust programs using linear ghost types", 2023 [Lat+23].

[17] Leino, "Dafny: An automatic program verifier for functional correctness", 2010 [Lei10].

Lastly, it would also be interesting to explore further use of unstable resources in Iris. That is, the main unstable resource that we use in this dissertation is the heap. However, Iris is famous for supporting a wide variety of resources for ghost state and developing constructions such as the heap by composing a collection of reusable combinators. In particular, in Daenerys, we have already defined several reusable combinators on unstable resources (in §26.3.1), including the $Sil(M)$ resource algebra, which extends a regular unital resource algebra M with unstable elements. It would be interesting to explore whether there are further use cases of unstable resources beyond heaps.

Bibliography

[ADG16] Aws Albarghouthi, Isil Dillig, and Arie Gurfinkel. “Maximal specification synthesis”. In: *POPL*. ACM, 2016, pp. 789–801. DOI: 10.1145/2837614.2837628.

[ADR09] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. “State-dependent representation independence”. In: *POPL*. ACM, 2009, pp. 340–353. DOI: 10.1145/1480881.1480925.

[Ahm+10] Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. “Semantic foundations for typed assembly languages”. In: *ACM Trans. Program. Lang. Syst.* 32.3 (2010), 7:1–7:67. DOI: 10.1145/1709093.1709094.

[Ahm04] Amal Ahmed. “Semantics of types for mutable state”. PhD thesis. Princeton University, 2004.

[AM01] Andrew W. Appel and David A. McAllester. “An indexed model of recursive types for foundational proof-carrying code”. In: *ACM Trans. Program. Lang. Syst.* 23.5 (2001), pp. 657–683. DOI: 10.1145/504709.504712.

[App+07] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. “A very modal model of a modern, major, general type system”. In: *POPL*. ACM, 2007, pp. 109–122. DOI: 10.1145/1190216.1190235.

[App12] Andrew W. Appel. “Verified Software Toolchain”. In: *NASA Formal Methods*. Vol. 7226. LNCS. Springer, 2012, p. 2. DOI: 10.1007/978-3-642-28891-3_2.

[AR89] Pierre America and Jan J. M. M. Rutten. “Solving reflexive domain equations in a category of complete metric spaces”. In: *J. Comput. Syst. Sci.* 39.3 (1989), pp. 343–375. DOI: 10.1016/0022-0000(89)90027-5.

[Ast+19] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. “Leveraging Rust types for modular specification and verification”. In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019), 147:1–147:30. DOI: 10.1145/3360573.

[Atk11] Robert Atkey. “Amortised resource analysis with separation logic”. In: *Log. Methods Comput. Sci.* 7.2 (2011). DOI: 10.2168/LMCS-7(2:17)2011.

[Bar+22] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. “cvc5: A versatile and industrial-strength SMT solver”. In: *TACAS (1)*. Vol. 13243. LNCS. Springer, 2022, pp. 415–442. DOI: 10.1007/978-3-030-99524-9_24.

[BB07] Nick Benton and Peter Buchlovsky. “Semantics of an effect analysis for exceptions”. In: *TLDI*. ACM, 2007, pp. 15–26. DOI: 10.1145/1190315.1190320.

[BBM14] Ales Bizjak, Lars Birkedal, and Marino Miculan. “A model of countable nondeterminism in guarded type theory”. In: *RTA-TLCA*. Vol. 8560. LNCS. Springer, 2014, pp. 108–123. DOI: 10.1007/978-3-319-08918-8_8.

[BBP13] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. “Extending Sledgehammer with SMT solvers”. In: *J. Autom. Reason.* 51.1 (2013), pp. 109–128. DOI: 10.1007/S10817-013-9278-5.

[BBS13] Lars Birkedal, Ales Bizjak, and Jan Schwinghammer. “Step-indexed relational reasoning for countable nondeterminism”. In: *Log. Methods Comput. Sci.* 9.4 (2013). DOI: 10.2168/LMCS-9(4:4)2013.

[Ben+06] Nick Benton, Andrew Kennedy, Martin Hofmann, and Lennart Beringer. “Reading, writing and relations”. In: *APLAS*. Vol. 4279. LNCS. Springer, 2006, pp. 114–130. DOI: 10.1007/11924661_7.

[Ben+07] Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. “Relational semantics for effect-based program transformations with dynamic allocation”. In: *PPDP*. ACM, 2007, pp. 87–96. DOI: 10.1145/1273920.1273932.

[Ben+09] Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. “Relational semantics for effect-based program transformations: Higher-order store”. In: *PPDP*. ACM, 2009, pp. 301–312. DOI: 10.1145/1599410.1599447.

[BGK17] James Brotherston, Nikos Gorogiannis, and Max I. Kanovich. “Biabduction (and related problems) in array separation logic”. In: *CADE*. Vol. 10395. LNCS. Springer, 2017, pp. 472–490. DOI: 10.1007/978-3-319-63046-5_29.

[BGM21] Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. “Diamonds are not forever: Liveness in reactive programming with guarded recursion”. In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–28. DOI: 10.1145/3434283.

[Bir+11] Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. “Step-indexed Kripke models over recursive worlds”. In: *POPL*. ACM, 2011, pp. 119–132. DOI: 10.1145/1926385.1926401.

[Bir+12] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. “First steps in synthetic guarded domain theory: Step-indexing in the topos of trees”. In: *Log. Methods Comput. Sci.* 8.4 (2012). DOI: 10.2168/LMCS-8(4:1)2012.

[Bir+16] Lars Birkedal, Guilhem Jaber, Filip Sieczkowski, and Jacob Thamsborg. “A Kripke logical relation for effect-based program transformations”. In: *Inf. Comput.* 249 (2016), pp. 160–189. DOI: 10.1016/J.IC.2016.04.003.

[Bir+21] Lars Birkedal, Thomas Dinsdale-Young, Armaël Guéneau, Guilhem Jaber, Kasper Svendsen, and Nikos Tzevelekos. “Theorems for free from separation logic specifications”. In: *Proc. ACM Program. Lang.* 5.ICFP (2021), pp. 1–29. DOI: 10.1145/3473586.

[Blo+17] Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. “The VerCors tool set: Verification of parallel and concurrent software”. In: *IFM*. Vol. 10510. LNCS. Springer, 2017, pp. 102–110. DOI: 10.1007/978-3-319-66845-1_7.

[Bor+05] Richard Bornat, Cristiano Calcagno, Peter W. O’Hearn, and Matthew J. Parkinson. “Permission accounting in separation logic”. In: *POPL*. ACM, 2005, pp. 259–270. DOI: 10.1145/1040305.1040327.

[Boy03] John Boyland. “Checking interference with fractional permissions”. In: *SAS*. Vol. 2694. LNCS. Springer, 2003, pp. 55–72. DOI: 10.1007/3-540-44898-5_4.

[Bro07] Stephen Brookes. “A semantics for concurrent separation logic”. In: *Theor. Comput. Sci.* 375.1-3 (2007), pp. 227–270. DOI: 10.1016/J.TCS.2006.12.034.

[Cal+09] Cristiano Calcagno, Dino Distefano, Peter W. O’Hearn, and Hongseok Yang. “Compositional shape analysis by means of bi-abduction”. In: *POPL*. ACM, 2009, pp. 289–300. DOI: 10.1145/1480881.1480917.

[Cal+11] Cristiano Calcagno, Dino Distefano, Peter W. O’Hearn, and Hongseok Yang. “Compositional shape analysis by means of bi-abduction”. In: *J. ACM* 58.6 (2011), 26:1–26:66. DOI: 10.1145/2049697.2049700.

[Cal+19] Cristiano Calcagno, Dino Distefano, Peter O’Hearn, and Hongseok Yang. *Go Huge or Go Home: POPL’19 Most Influential Paper Retrospective.* 2019. URL: https://blog.sigplan.org/2020/03/03/go-huge-or-go-home-popl19-most-influential-paper-retrospective/ (visited on 2024).

[Cao+18] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. “VST-Floyd: A separation logic tool to verify correctness of C programs”. In: *J. Autom. Reason.* 61.1-4 (2018), pp. 367–422. DOI: 10.1007/S10817-018-9457-5.

[Car+22] Quentin Carbonneaux, Noam Zilberstein, Christoph Klee, Peter W. O'Hearn, and Francesco Zappa Nardelli. "Applying formal verification to microkernel IPC at Meta". In: *CPP*. ACM, 2022, pp. 116–129. DOI: 10.1145/3497775.3503681.

[Cha+15] Soham Chakraborty, Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. "Aspect-oriented linearizability proofs". In: *Log. Methods Comput. Sci.* 11.1 (2015). DOI: 10.2168/LMCS-11(1:20)2015.

[Cha+19a] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. "Verifying concurrent, crash-safe systems with Perennial". In: *SOSP*. ACM, 2019, pp. 243–258. DOI: 10.1145/3341301.3359632.

[Cha+19b] Arthur Charguéraud, Jean-Christophe Filliâtre, Cláudio Lourenço, and Mário Pereira. "GOSPEL - Providing OCaml with a formal specification language". In: *FM*. Vol. 11800. LNCS. Springer, 2019, pp. 484–501. DOI: 10.1007/978-3-030-30942-8_29.

[Cha+21] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. "GoJournal: A verified, concurrent, crash-safe journaling system". In: *OSDI*. USENIX Association, 2021, pp. 423–439. URL: https://www.usenix.org/conference/osdi21/presentation/chajed.

[Cha+23] Yun-Sheng Chang, Ralf Jung, Upamanyu Sharma, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. "Verifying vMVCC, a high-performance transaction library using multi-version concurrency control". In: *OSDI*. USENIX Association, 2023, pp. 871–886. URL: https://www.usenix.org/conference/osdi23/presentation/chang.

[Cha10] Arthur Charguéraud. "Program verification through characteristic formulae". In: *ICFP*. ACM, 2010, pp. 321–332. DOI: 10.1145/1863543.1863590.

[Cha11] Arthur Charguéraud. "Characteristic formulae for the verification of imperative programs". In: *ICFP*. ACM, 2011, pp. 418–430. DOI: 10.1145/2034773.2034828.

[Chl11] Adam Chlipala. "Mostly-automated verification of low-level programs in computational separation logic". In: *PLDI*. ACM, 2011, pp. 234–245. DOI: 10.1145/1993498.1993526.

[CLQ19] Christopher Curry, Quang Loc Le, and Shengchao Qin. "Bi-abductive inference for shape and ordering properties". In: *ICECCS*. IEEE, 2019, pp. 220–225. DOI: 10.1109/ICECCS.2019.00031.

[CP17] Arthur Charguéraud and François Pottier. "Temporary read-only permissions for separation logic". In: *ESOP*. Vol. 10201. LNCS. Springer, 2017, pp. 260–286. DOI: 10.1007/978-3-662-54434-1_10.

[Cyr25] Cyrus IMAPD. *Cyrus IMAPD memory wrapper operations*. https://github.com/cyrusimap/cyrus-imapd/blob/0552750789f23d205b50f582f73358d73cc15706/lib/xmalloc.c. 2025.

[DAB11] Derek Dreyer, Amal Ahmed, and Lars Birkedal. "Logical step-indexed logical relations". In: *Log. Methods Comput. Sci.* 7.2 (2011). DOI: 10.2168/LMCS-7(2:16)2011.

[Dan+20] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. "RustBelt meets relaxed memory". In: *Proc. ACM Program. Lang.* 4.POPL (2020), 34:1–34:29. DOI: 10.1145/3371102.

[Dar+25] Thibault Dardinier, Michael Sammler, Gaurav Parthasarathy, Alexander J. Summers, and Peter Müller. "Formal foundations for translational separation logic verifiers". In: *Proc. ACM Program. Lang.* 9.POPL (Jan. 2025). DOI: 10.1145/3704856. URL: https://doi.org/10.1145/3704856.

[Del00] David Delahaye. "A tactic language for the system Coq". In: *LPAR*. Vol. 1955. LNCS. Springer, 2000, pp. 85–95. DOI: 10.1007/3-540-44404-1_7.

[DH10] Robert Dockins and Aquinas Hobor. "A theory of termination via indirection". In: *Modelling, Controlling and Reasoning About State*. Vol. 10351. Dagstuhl Seminar Proceedings. 2010. URL: http://drops.dagstuhl.de/opus/volltexte/2010/2805/.

[DH12] Robert Dockins and Aquinas Hobor. "Time bounds for general function pointers". In: *MFPS*. Vol. 286. Electronic Notes in Theoretical Computer Science. Elsevier, 2012, pp. 139–155. DOI: 10.1016/J.ENTCS.2012.08.010.

[Din+10] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. "Concurrent abstract predicates". In: *ECOOP*. Vol. 6183. LNCS. Springer, 2010, pp. 504–528. DOI: 10.1007/978-3-642-14107-2_24.

[Din+13] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. "Views: Compositional reasoning for concurrent programs". In: *POPL*. ACM, 2013, pp. 287–300. DOI: 10.1145/2429069.2429104.

[Doh+18] Jérôme Dohrau, Alexander J. Summers, Caterina Urban, Severin Münger, and Peter Müller. "Permission inference for array programs". In: *CAV (2)*. Vol. 10982. LNCS. Springer, 2018, pp. 55–74. DOI: 10.1007/978-3-319-96142-2_7.

[Doh22] Jérôme Dohrau. "Automatic inference of permission specifications". PhD thesis. ETH Zurich, Zürich, Switzerland, 2022. DOI: 10.3929/ethz-b-000588977.

[DOs+21] Emanuele D'Osualdo, Julian Sutherland, Azadeh Farzan, and Philippa Gardner. "TaDA Live: Compositional reasoning for termination of fine-grained concurrent programs". In: *ACM Trans. Program. Lang. Syst.* 43.4 (2021), 16:1–16:134. DOI: 10.1145/3477082.

[Dre+10] Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. "A relational modal logic for higher-order stateful ADTs". In: *POPL*. ACM, 2010, pp. 185–198. DOI: 10.1145/1706299.1706323.

[Dre+25] Derek Dreyer, Simon Spies, Lennard Gäher, Ralf Jung, Jan-Oliver Kaiser, Hoang-Hai Dang, David Swasey, Jan Menz, Niklas Mück, and Benjamin Peters. *Semantics of type systems (lecture notes)*. Available at https://plv.mpi-sws.org/semantics-course/. 2025.

[Eki+17] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W. Barrett. "SMTCoq: A plug-in for integrating SMT solvers into Coq". In: *CAV (2)*. Vol. 10427. LNCS. Springer, 2017, pp. 126–133. DOI: 10.1007/978-3-319-63390-9_7.

[Elm+10] Tayfun Elmas, Shaz Qadeer, Ali Sezgin, Omer Subasi, and Serdar Tasiran. "Simplifying linearizability proofs with reduction and abstraction". In: *TACAS*. Vol. 6015. LNCS. Springer, 2010, pp. 296–311. DOI: 10.1007/978-3-642-12002-2_25.

[EM18] Marco Eilers and Peter Müller. "Nagini: A static verifier for Python". In: *CAV (1)*. Vol. 10981. LNCS. Springer, 2018, pp. 596–603. DOI: 10.1007/978-3-319-96145-3_33.

[FKB18] Dan Frumin, Robbert Krebbers, and Lars Birkedal. "ReLoC: A mechanised relational logic for fine-grained concurrency". In: *LICS*. ACM, 2018, pp. 442–451. DOI: 10.1145/3209108.3209174.

[FKB21a] Dan Frumin, Robbert Krebbers, and Lars Birkedal. "Compositional non-interference for fine-grained concurrent programs". In: *SP*. IEEE, 2021, pp. 1416–1433. DOI: 10.1109/SP40001.2021.00003.

[FKB21b] Dan Frumin, Robbert Krebbers, and Lars Birkedal. "ReLoC Reloaded: A mechanized relational logic for fine-grained concurrency and logical atomicity". In: *Log. Methods Comput. Sci.* 17.3 (2021). DOI: 10.46298/LMCS-17(3:9)2021.

[Flo67] Robert W Floyd. "Assigning meanings to programs". In: *American Mathematical Society*. Mathematical Aspects of Computer Science (1967). Ed. by JT Schwartz.

[FM12] Pietro Ferrara and Peter Müller. "Automatic inference of access permissions". In: *VMCAI*. Vol. 7148. LNCS. Springer, 2012, pp. 202–218. DOI: 10.1007/978-3-642-27940-9_14.

[Fro+21] Aymeric Fromherz, Aseem Rastogi, Nikhil Swamy, Sydney Gibson, Guido Martínez, Denis Merigoux, and Tahina Ramananandro. "Steel: Proof-oriented programming in a dependently typed concurrent separation logic". In: *Proc. ACM Program. Lang.* 5.ICFP (2021), pp. 1–30. DOI: 10.1145/3473590.

[Gäh+22] Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. "Simuliris: A separation logic framework for verifying concurrent program optimizations". In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–31. DOI: 10.1145/3498689.

[Gäh+24] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. "RefinedRust: A type system for high-assurance verification of Rust programs". In: *Proc. ACM Program. Lang.* 8.PLDI (June 2024). DOI: 10.1145/3656422.

[Gia+20] Paolo G. Giarrusso, Léo Stefanesco, Amin Timany, Lars Birkedal, and Robbert Krebbers. "Scala step-by-step: Soundness for DOT with step-indexed logical relations in Iris". In: *Proc. ACM Program. Lang.* 4.ICFP (2020), 114:1–114:29. DOI: 10.1145/3408996.

[Git25] Git. *Git memory wrapper operations.* https://github.com/git/git/blob/2e8e77cbac8ac17f94eee2087187fa1718e38b14/wrapper.c. 2025.

[GM04] Pietro Di Gianantonio and Marino Miculan. "Unifying recursive and co-recursive definitions in sheaf categories". In: *FoSSaCS.* Vol. 2987. LNCS. Springer, 2004, pp. 136–150. DOI: 10.1007/978-3-540-24727-2_11.

[Gre+24] Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. "Asynchronous probabilistic couplings in higher-order separation logic". In: *Proc. ACM Program. Lang.* 8.POPL (2024), pp. 753–784. DOI: 10.1145/3632868.

[Gué+23] Armaël Guéneau, Johannes Hostert, Simon Spies, Michael Sammler, Lars Birkedal, and Derek Dreyer. "Melocoton: A program logic for verified interoperability between OCaml and C". In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (2023), pp. 716–744. DOI: 10.1145/3622823.

[Ham+24] Angus Hammond, Zongyuan Liu, Thibaut Pérami, Peter Sewell, Lars Birkedal, and Jean Pichon-Pharabod. "An axiomatic basis for computer programming on the relaxed Arm-A architecture: The AxSL logic". In: *Proc. ACM Program. Lang.* 8.POPL (2024), pp. 604–637. DOI: 10.1145/3632863.

[HBK22] Jonas Kastberg Hinrichsen, Jesper Bengtson, and Robbert Krebbers. "Actris 2.0: Asynchronous session-type based reasoning in separation logic". In: *Log. Methods Comput. Sci.* 18.2 (2022). DOI: 10.46298/LMCS-18(2:16)2022.

[He+13] Guanhua He, Shengchao Qin, Wei-Ngan Chin, and Florin Craciun. "Automated specification discovery via user-defined predicates". In: *ICFEM.* Vol. 8144. LNCS. Springer, 2013, pp. 397–414. DOI: 10.1007/978-3-642-41202-8_26.

[Hes06] Gerhard Hessenberg. *Grundbegriffe der Mengenlehre.* Vol. 1. Vandenhoeck & Ruprecht, 1906.

[Hoa69] C. A. R. Hoare. "An axiomatic basis for computer programming". In: *Commun. ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259.

[Hol+22] Lukás Holík, Petr Peringer, Adam Rogalewicz, Veronika Soková, Tomás Vojnar, and Florian Zuleger. "Low-level bi-abduction". In: *ECOOP.* Vol. 222. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 19:1–19:30. DOI: 10.4230/LIPICS.ECOOP.2022.19.

[HW90] Maurice Herlihy and Jeannette M. Wing. "Linearizability: A correctness condition for concurrent objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (1990), pp. 463–492. DOI: 10.1145/78969.78972.

[Inf24] Infer. *Infer v1.2.0.* https://fbinfer.com. June 2024.

[Iri24] Iris Team. *The Iris 4.3 Reference.* 2024. URL: https://plv.mpi-sws.org/iris/appendix-4.3.pdf.

[Jac+11] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. "VeriFast: A powerful, sound, predictable, fast verifier for C and Java". In: *NASA Formal Methods.* Vol. 6617. LNCS. Springer, 2011, pp. 41–55. DOI: 10.1007/978-3-642-20398-5_4.

[JBK13] Jonas Braband Jensen, Nick Benton, and Andrew Kennedy. "High-level separation logic for low-level code". In: *POPL.* ACM, 2013, pp. 301–314. DOI: 10.1145/2429069.2429105.

[JHK23] Jules Jacobs, Jonas Kastberg Hinrichsen, and Robbert Krebbers. "Dependent session protocols in separation logic from first principles (functional pearl)". In: *Proc. ACM Program. Lang.* 7.ICFP (2023), pp. 768–795. DOI: 10.1145/3607856.

[Jun+15] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. "Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning". In: *POPL*. ACM, 2015, pp. 637–650. DOI: 10.1145/2676726.2676980.

[Jun+16] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. "Higher-order ghost state". In: *ICFP*. ACM, 2016, pp. 256–269. DOI: 10.1145/2951913.2951943.

[Jun+18a] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "RustBelt: Securing the foundations of the Rust programming language". In: *Proc. ACM Program. Lang.* 2.POPL (2018), 66:1–66:34. DOI: 10.1145/3158154.

[Jun+18b] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. "Iris from the ground up: A modular foundation for higher-order concurrent separation logic". In: *J. Funct. Program.* 28 (2018), e20. DOI: 10.1017/S0956796818000151.

[Jun+20] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. "The future is ours: Prophecy variables in separation logic". In: *Proc. ACM Program. Lang.* 4.POPL (2020), 45:1–45:32. DOI: 10.1145/3371113.

[Jun19] Ralf Jung. *Logical atomicity in Iris: The good, the bad, and the ugly*. Presented at the Iris Workshop (https://iris-project.org/workshop-2019/). 2019. URL: https://people.mpi-sws.org/~jung/iris/logatom-talk-2019.pdf.

[Jun20] Ralf Jung. "Understanding and evolving the Rust programming language". PhD thesis. Saarland University, 2020.

[Kai+17] Jan-Oliver Kaiser, Hoang-Hai Dang, Derek Dreyer, Ori Lahav, and Viktor Vafeiadis. "Strong logic for weak memory: Reasoning about release-acquire consistency in Iris". In: *ECOOP*. Vol. 74. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 17:1–17:29. DOI: 10.4230/LIPICS.ECOOP.2017.17.

[Kan+17] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. "A promising semantics for relaxed-memory concurrency". In: *POPL*. ACM, 2017, pp. 175–189. DOI: 10.1145/3009837.3009850.

[Keu+22] Steven Keuchel, Sander Huyghebaert, Georgy Lukyanov, and Dominique Devriese. "Verified symbolic execution with Kripke specification monads (and no meta-programming)". In: *Proc. ACM Program. Lang.* 6.ICFP (2022), pp. 194–224. DOI: 10.1145/3547628.

[KGP16] Artem Khyzha, Alexey Gotsman, and Matthew J. Parkinson. "A generic logic for proving linearizability". In: *FM*. Vol. 9995. LNCS. 2016, pp. 426–443. DOI: 10.1007/978-3-319-48989-6_26.

[Kre+17] Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. "The essence of higher-order concurrent separation logic". In: *ESOP*. Vol. 10201. LNCS. Springer, 2017, pp. 696–723. DOI: 10.1007/978-3-662-54434-1_26.

[Kre+18] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. "MoSeL: A general, extensible modal framework for interactive proofs in separation logic". In: *Proc. ACM Program. Lang.* 2.ICFP (2018), 77:1–77:30. DOI: 10.1145/3236772.

[KS19] Dominik Kirst and Gert Smolka. "Categoricity results and large model constructions for second-order ZF in dependent type theory". In: *J. Autom. Reason.* 63.2 (2019), pp. 415–438. DOI: 10.1007/S10817-018-9480-6.

[KSB17] Morten Krogh-Jespersen, Kasper Svendsen, and Lars Birkedal. "A relational model of types-and-effects in higher-order concurrent separation logic". In: *POPL*. ACM, 2017, pp. 218–231. DOI: 10.1145/3009837.3009877.

[KTB17] Robbert Krebbers, Amin Timany, and Lars Birkedal. "Interactive proofs in higher-order concurrent separation logic". In: *POPL*. ACM, 2017, pp. 205–217. DOI: 10.1145/3009837.3009855.

[Lat+23] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. "Verus: Verifying Rust programs using linear ghost types". In: *Proc. ACM Program. Lang.* 7.OOPSLA1 (2023), pp. 286–315. DOI: 10.1145/3586037.

[Le+14] Quang Loc Le, Cristian Gherghina, Shengchao Qin, and Wei-Ngan Chin. "Shape analysis via second-order bi-abduction". In: *CAV*. Vol. 8559. LNCS. Springer, 2014, pp. 52–68. DOI: 10.1007/978-3-319-08867-9_4.

[Le+22] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. "Finding real bugs in big programs with incorrectness logic". In: *Proc. ACM Program. Lang.* 6.OOPSLA1 (2022), pp. 1–27. DOI: 10.1145/3527325.

[Leh+23] Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. "Flux: Liquid types for Rust". In: *Proc. ACM Program. Lang.* 7.PLDI (2023), pp. 1533–1557. DOI: 10.1145/3591283.

[Lei10] K. Rustan M. Leino. "Dafny: An automatic program verifier for functional correctness". In: *LPAR (Dakar)*. Vol. 6355. LNCS. Springer, 2010, pp. 348–370. DOI: 10.1007/978-3-642-17511-4_20.

[Lep+22] Rodolphe Lepigre, Michael Sammler, Kayvan Memarian, Robbert Krebbers, Derek Dreyer, and Peter Sewell. "VIP: Verifying real-world C idioms with integer-pointer casts". In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–32. DOI: 10.1145/3498681.

[Lev65] Vladimir Iosifovich Levenshtein. "Binary codes capable of correcting deletions, insertions, and reversals". In: *Soviet Physics Doklady* 10 (1965), pp. 707–710.

[LF13] Hongjin Liang and Xinyu Feng. "Modular verification of linearizability with non-fixed linearization points". In: *PLDI*. ACM, 2013, pp. 459–470. DOI: 10.1145/2491956.2462189.

[LF16] Hongjin Liang and Xinyu Feng. "A program logic for concurrent objects under fair scheduling". In: *POPL*. ACM, 2016, pp. 385–399. DOI: 10.1145/2837614.2837635.

[LF18] Hongjin Liang and Xinyu Feng. "Progress of concurrent objects with partial methods". In: *Proc. ACM Program. Lang.* 2.POPL (2018), 20:1–20:31. DOI: 10.1145/3158108.

[LM09] K. Rustan M. Leino and Peter Müller. "A basis for verifying multi-threaded programs". In: *ESOP*. Vol. 5502. LNCS. Springer, 2009, pp. 378–393. DOI: 10.1007/978-3-642-00590-9_27.

[Lor+24] Jay Lorch, Travis Hance, Chris Hawblitzel, Andrea Lattuada, and Upamanyu Sharma. *Require open-invariant credits*. Verus pull request 1042. 2024. URL: https://github.com/verus-lang/verus/pull/1042.

[Mag+06] Stephen Magill, Aleksandar Nanevski, Edmund Clarke, and Peter Lee. "Inferring invariants in separation logic for imperative list-processing programs". In: *SPACE* 1.1 (2006), pp. 5–7.

[Mak+21] Petar Maksimovic, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. "Gillian, Part II: Real-world verification for JavaScript and C". In: *CAV (2)*. Vol. 12760. LNCS. Springer, 2021, pp. 827–850. DOI: 10.1007/978-3-030-81688-9_38.

[Mat+22] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. "RustHornBelt: A semantic foundation for functional verification of Rust programs with unsafe code". In: *PLDI*. ACM, 2022, pp. 841–856. DOI: 10.1145/3519939.3523704.

[MB08] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. "Z3: An efficient SMT solver". In: *TACAS*. Vol. 4963. LNCS. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[mem25] memcached. *memcached*. https://www.memcached.org/. 2025.

[MJ16] Mahmoud Mohsen and Bart Jacobs. "One step towards automatic inference of formal specifications using automated VeriFast". In: *FMICS-AVoCS*. Vol. 9933. LNCS. Springer, 2016, pp. 56–64. DOI: 10.1007/978-3-319-45943-1_4.

[MJP19] Glen Mével, Jacques-Henri Jourdan, and François Pottier. "Time credits and time receipts in Iris". In: *ESOP*. Vol. 11423. LNCS. Springer, 2019, pp. 3–29. DOI: 10.1007/978-3-030-17184-1_1.

[MJP20] Glen Mével, Jacques-Henri Jourdan, and François Pottier. "Cosmo: A concurrent separation logic for multicore OCaml". In: *Proc. ACM Program. Lang.* 4.ICFP (2020), 96:1–96:29. DOI: 10.1145/3408978.

[MKG22] Ike Mulder, Robbert Krebbers, and Herman Geuvers. "Diaframe: Automated verification of fine-grained concurrent programs in Iris". In: *PLDI*. ACM, 2022, pp. 809–824. DOI: 10.1145/3519939.3523432.

[MSS17] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. "Viper: A verification infrastructure for permission-based reasoning". In: *Dependable Software Systems Engineering*. Vol. 50. NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, 2017, pp. 104–125. DOI: 10.3233/978-1-61499-810-5-104.

[Nan+14] Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. "Communicating state transition systems for fine-grained concurrent resources". In: *ESOP*. Vol. 8410. LNCS. Springer, 2014, pp. 290–310. DOI: 10.1007/978-3-642-54833-8_16.

[Nan+19] Aleksandar Nanevski, Anindya Banerjee, Germán Andrés Delbianco, and Ignacio Fábregas. "Specifying concurrent programs in separation logic: Morphisms and simulations". In: *Proc. ACM Program. Lang.* 3.OOPSLA (2019), 161:1–161:30. DOI: 10.1145/3360587.

[OHe07] Peter W. O'Hearn. "Resources, concurrency, and local reasoning". In: *Theor. Comput. Sci.* 375.1-3 (2007), pp. 271–307. DOI: 10.1016/J.TCS.2006.12.035.

[OHe20] Peter W. O'Hearn. "Incorrectness Logic". In: *Proc. ACM Program. Lang.* 4.POPL (2020), 10:1–10:32. DOI: 10.1145/3371078.

[OP99] Peter W. O'Hearn and David J. Pym. "The logic of bunched implications". In: *Bull. Symb. Log.* 5.2 (1999), pp. 215–244. DOI: 10.2307/421090.

[Ope25] OpenSSL. *OpenSSL*. https://www.openssl.org. 2025.

[ORY01] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. "Local reasoning about programs that alter data structures". In: *CSL*. Vol. 2142. LNCS. Springer, 2001, pp. 1–19. DOI: 10.1007/3-540-44802-0_1.

[PB08] Matthew J. Parkinson and Gavin M. Bierman. "Separation logic, abstraction and inheritance". In: *POPL*. ACM, 2008, pp. 75–86. DOI: 10.1145/1328438.1328451.

[Pot+24] François Pottier, Armaël Guéneau, Jacques-Henri Jourdan, and Glen Mével. "Thunks and debits in separation logic with time credits". In: *Proc. ACM Program. Lang.* 8.POPL (2024), pp. 1482–1508. DOI: 10.1145/3632892.

[PP11] Alexandre Pilkiewicz and François Pottier. "The essence of monotonic state". In: *TLDI*. ACM, 2011, pp. 73–86. DOI: 10.1145/1929553.1929565.

[PS11] Matthew J. Parkinson and Alexander J. Summers. "The relationship between separation logic and implicit dynamic frames". In: *ESOP*. Vol. 6602. LNCS. Springer, 2011, pp. 439–458. DOI: 10.1007/978-3-642-19718-5_23.

[Pul+23] Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. "CN: Verifying systems C code with separation-logic refinement types". In: *Proc. ACM Program. Lang.* 7.POPL (2023), pp. 1–32. DOI: 10.1145/3571194.

[PWZ14] Ruzica Piskac, Thomas Wies, and Damien Zufferey. "GRASShopper - complete heap verification with mixed specifications". In: *TACAS*. Vol. 8413. LNCS. Springer, 2014, pp. 124–139. DOI: 10.1007/978-3-642-54862-8_9.

[Qin+11] Shengchao Qin, Chenguang Luo, Wei-Ngan Chin, and Guanhua He. "Automatically refining partial specifications for program verification". In: *FM*. Vol. 6664. LNCS. Springer, 2011, pp. 369–385. DOI: 10.1007/978-3-642-21437-0_28.

[Raa+20] Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter W. O'Hearn, and Jules Villard. "Local reasoning about the presence of bugs: Incorrectness Separation Logic". In: *CAV (2)*. Vol. 12225. LNCS. Springer, 2020, pp. 225–252. DOI: 10.1007/978-3-030-53291-8_14.

[RDG14] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. “TaDA: A logic for time and data abstraction”. In: *ECOOP*. Vol. 8586. LNCS. Springer, 2014, pp. 207–231. DOI: 10.1007/978-3-662-44202-9_9.

[Red25a] Redis. *Redis memory wrapper operations*. https://github.com/redis/redis/blob/3fac869f02657d94dc89fab23acb8ef188889c96/src/zmalloc.c. 2025.

[Red25b] Redis. *Redis popcount implementation for potentially large buffers*. https://github.com/redis/redis/blob/3fac869f02657d94dc89fab23acb8ef188889c96/src/bitops.c#L40. 2025.

[Ref25] RefinedC Developers. *RefinedC verification of a priority bitmap*. https://gitlab.mpi-sws.org/iris/refinedc/-/blob/7945a29d1647970709a9b5ad2ffc53c757e130cc/examples/scheduler/include/fdsched/priority.h. 2025.

[Rey02] John C. Reynolds. “Separation Logic: A logic for shared mutable data structures”. In: *LICS*. IEEE Computer Society, 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817.

[RKJ08] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. “Liquid types”. In: *PLDI*. ACM, 2008, pp. 159–169. DOI: 10.1145/1375581.1375602.

[RKJ10] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. “Low-level liquid types”. In: *POPL*. ACM, 2010, pp. 131–144. DOI: 10.1145/1706299.1706316.

[Roc+16] Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner, and Julian Sutherland. “Modular termination verification for non-blocking concurrency”. In: *ESOP*. Vol. 9632. LNCS. Springer, 2016, pp. 176–201. DOI: 10.1007/978-3-662-49498-1_8.

[RVG15] Azalea Raad, Jules Villard, and Philippa Gardner. “CoLoSL: Concurrent local subjective logic”. In: *ESOP*. Vol. 9032. LNCS. Springer, 2015, pp. 710–735. DOI: 10.1007/978-3-662-46669-8_29.

[S+21] Sumanth Prabhu S, Grigory Fedyukovich, Kumar Madhukar, and Deepak D’Souza. “Specification synthesis with constrained Horn clauses”. In: *PLDI*. ACM, 2021, pp. 1203–1217. DOI: 10.1145/3453483.3454104.

[Sam+21] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. “RefinedC: Automating the foundational verification of C code with refined ownership types”. In: *PLDI*. ACM, 2021, pp. 158–174. DOI: 10.1145/3453483.3454036.

[Sam+23] Michael Sammler, Simon Spies, Youngju Song, Emanuele D’Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. “DimSum: A decentralized approach to multi-language semantics and verification”. In: *Proc. ACM Program. Lang.* 7.POPL (2023), pp. 775–805. DOI: 10.1145/3571220.

[San+20] José Fragoso Santos, Petar Maksimovic, Sacha-Élie Ayoun, and Philippa Gardner. “Gillian, Part I: A multi-language platform for symbolic execution”. In: *PLDI*. ACM, 2020, pp. 927–942. DOI: 10.1145/3385412.3386014.

[SB14] Kasper Svendsen and Lars Birkedal. “Impredicative concurrent abstract predicates”. In: *ESOP*. Vol. 8410. LNCS. Springer, 2014, pp. 149–168. DOI: 10.1007/978-3-642-54833-8_9.

[SBP13] Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson. “Modular reasoning about separation of concurrent data structures”. In: *ESOP*. Vol. 7792. LNCS. Springer, 2013, pp. 169–188. DOI: 10.1007/978-3-642-37036-6_11.

[SJP09] Jan Smans, Bart Jacobs, and Frank Piessens. “Implicit Dynamic Frames: Combining dynamic frames and separation logic”. In: *ECOOP*. Vol. 5653. LNCS. Springer, 2009, pp. 148–172. DOI: 10.1007/978-3-642-03013-0_8.

[SK13] Mohamed Nassim Seghir and Daniel Kroening. “Counterexample-guided precondition inference”. In: *ESOP*. Vol. 7792. LNCS. Springer, 2013, pp. 451–471. DOI: 10.1007/978-3-642-37036-6_25.

[SKD21] Simon Spies, Neel Krishnaswami, and Derek Dreyer. “Transfinite step-indexing for termination”. In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–29. DOI: 10.1145/3434294.

[SNB15] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. “Specifying and verifying concurrent algorithms with histories and subjectivity”. In: *ESOP*. Vol. 9032. LNCS. Springer, 2015, pp. 333–358. DOI: 10.1007/978-3-662-46669-8_14.

[SO08] Matthieu Sozeau and Nicolas Oury. “First-class type classes”. In: *TPHOLs*. Vol. 5170. LNCS. Springer, 2008, pp. 278–293. DOI: 10.1007/978-3-540-71067-7_23.

[Spi+21a] Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. *Transfinite Iris appendix and Rocq development*. 2021. URL: https://iris-project.org/transfinite-iris/.

[Spi+21b] Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. “Transfinite Iris: Resolving an existential dilemma of step-indexed separation logic”. In: *PLDI*. ACM, 2021, pp. 80–95. DOI: 10.1145/3453483.3454031.

[Spi+22a] Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. “Later Credits: Resourceful reasoning for the later modality”. In: *Proc. ACM Program. Lang.* 6.ICFP (2022), pp. 283–311. DOI: 10.1145/3547631.

[Spi+22b] Simon Spies, Lennard Gäher, Joseph Tassarotti, Ralf Jung, Robbert Rebbers, Lars Birkedal, and Derek Dreyer. *Later Credits Rocq development and technical documentation*. Latest development at https://plv.mpi-sws.org/later-credits/. 2022. DOI: 10.5281/zenodo.6702804.

[Spi+24a] Simon Spies, Lennard Gäher, Michael Sammler, and Derek Dreyer. “Quiver: Guided abductive inference of separation logic specifications in Coq”. In: *Proc. ACM Program. Lang.* 8.PLDI (June 2024). DOI: 10.1145/3656413.

[Spi+24b] Simon Spies, Lennard Gäher, Michael Sammler, and Derek Dreyer. *Quiver: Guided abductive inference of separation logic specifications in Coq (Rocq development and appendix)*. Project webpage with appendix: https://plv.mpi-sws.org/quiver/. June 2024. DOI: 10.5281/zenodo.10940320.

[Spi+25a] Simon Spies, Niklas Mück, Haoyi Zeng, Michael Sammler, Andrea Lattuada, Peter Müller, and Derek Dreyer. “Destabilizing Iris”. In: *Proc. ACM Program. Lang.* 9.PLDI (June 2025). DOI: 10.1145/3729284.

[Spi+25b] Simon Spies, Niklas Mück, Haoyi Zeng, Michael Sammler, Andrea Lattuada, Peter Müller, and Derek Dreyer. *Destabilizing Iris (Rocq development and appendix)*. Project webpage with appendix: https://plv.mpi-sws.org/iris-daenerys/. June 2025. DOI: 10.5281/zenodo.15041580.

[Spi24] Simon Spies. *Making adequacy of Iris satisfying*. Presented at the 4th Iris Workshop (https://iris-project.org/workshop-2024/). 2024. URL: https://plv.mpi-sws.org/iris-satisfiability/talks/iris-workshop-2024.pdf.

[SSB16] Kasper Svendsen, Filip Sieczkowski, and Lars Birkedal. “Transfinite step-indexing: Decoupling concrete and logical steps”. In: *ESOP*. Vol. 9632. LNCS. Springer, 2016, pp. 727–751. DOI: 10.1007/978-3-662-49498-1_28.

[Sve+18] Kasper Svendsen, Jean Pichon-Pharabod, Marko Doko, Ori Lahav, and Viktor Vafeiadis. “A separation logic for a promising semantics”. In: *ESOP*. Vol. 10801. LNCS. Springer, 2018, pp. 357–384. DOI: 10.1007/978-3-319-89884-1_13.

[Swa+20] Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. “SteelCore: An extensible concurrent separation logic for effectful dependently typed programs”. In: *Proc. ACM Program. Lang.* 4.ICFP (2020), 121:1–121:30. DOI: 10.1145/3409003.

[TDB13] Aaron Turon, Derek Dreyer, and Lars Birkedal. “Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency”. In: *ICFP*. ACM, 2013, pp. 377–390. DOI: 10.1145/2500365.2500600.

[TH19] Joseph Tassarotti and Robert Harper. “A separation logic for concurrent randomized programs”. In: *Proc. ACM Program. Lang.* 3.POPL (2019), 64:1–64:30. DOI: 10.1145/3290377.

[Tim+18] Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. "A logical relation for monadic encapsulation of state: Proving contextual equivalences in the presence of runST". In: *Proc. ACM Program. Lang.* 2.POPL (2018), 64:1–64:28. DOI: 10.1145/3158152.

[Tim+24a] Amin Timany, Simon Oddershede Gregersen, Léo Stefanesco, Jonas Kastberg Hinrichsen, Léon Gondelman, Abel Nieto, and Lars Birkedal. "Trillium: Higher-order concurrent and distributed separation logic for intensional refinement". In: *Proc. ACM Program. Lang.* 8.POPL (2024), pp. 241–272. DOI: 10.1145/3632851.

[Tim+24b] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. "A logical approach to type soundness". In: *J. ACM* (July 2024). ISSN: 0004-5411. DOI: 10.1145/3676954. URL: https://doi.org/10.1145/3676954.

[TJH17] Joseph Tassarotti, Ralf Jung, and Robert Harper. "A higher-order logic for concurrent termination-preserving refinement". In: *ESOP*. Vol. 10201. LNCS. 2017, pp. 909–936. DOI: 10.1007/978-3-662-54434-1_34.

[Tri+13] Minh-Thai Trinh, Quang Loc Le, Cristina David, and Wei-Ngan Chin. "Bi-abduction with pure properties for specification inference". In: *APLAS*. Vol. 8301. LNCS. Springer, 2013, pp. 107–123. DOI: 10.1007/978-3-319-03542-0_8.

[Tur+13] Aaron Joseph Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. "Logical relations for fine-grained concurrency". In: *POPL*. ACM, 2013, pp. 343–356. DOI: 10.1145/2429069.2429111.

[Vaz+14] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. "Refinement types for Haskell". In: *ICFP*. ACM, 2014, pp. 269–282. DOI: 10.1145/2628136.2628161.

[VB21] Simon Friis Vindum and Lars Birkedal. "Contextual refinement of the Michael-Scott queue (proof pearl)". In: *CPP*. ACM, 2021, pp. 76–90. DOI: 10.1145/3437992.3439930.

[VGB25] Simon Friis Vindum, Aïna Linn Georges, and Lars Birkedal. "The Nextgen modality: A modality for non-frame-preserving updates in separation logic". In: *CPP*. New York, NY, USA: ACM, 2025, pp. 83–97. ISBN: 9798400713477. DOI: 10.1145/3703595.3705876. URL: https://doi.org/10.1145/3703595.3705876.

[Vog+11] Frédéric Vogels, Bart Jacobs, Frank Piessens, and Jan Smans. "Annotation inference for separation logic based verifiers". In: *FMOODS/FORTE*. Vol. 6722. LNCS. Springer, 2011, pp. 319–333. DOI: 10.1007/978-3-642-21461-5_21.

[Wer97] Benjamin Werner. "Sets in types, types in sets". In: *TACS*. Vol. 1281. LNCS. Springer, 1997, pp. 530–346. DOI: 10.1007/BFB0014566.

[Wis+20] Jenna Wise, Johannes Bader, Cameron Wong, Jonathan Aldrich, Éric Tanter, and Joshua Sunshine. "Gradual verification of recursive heap data structures". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 228:1–228:28. DOI: 10.1145/3428296.

[Wol+21] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João Carlos Pereira, and Peter Müller. "Gobra: Modular specification and verification of Go programs". In: *CAV (1)*. Vol. 12759. LNCS. Springer, 2021, pp. 367–379. DOI: 10.1007/978-3-030-81685-8_17.

[YHB08] Nobuko Yoshida, Kohei Honda, and Martin Berger. "Logical reasoning for higher-order functions with local state". In: *Log. Methods Comput. Sci.* 4.4 (2008). DOI: 10.2168/LMCS-4(4:2)2008.

[Zhu+22] Fengmin Zhu, Michael Sammler, Rodolphe Lepigre, Derek Dreyer, and Deepak Garg. "BFF: Foundational and automated verification of bitfield-manipulating programs". In: *Proc. ACM Program. Lang.* 6.OOPSLA2 (2022), pp. 1613–1638. DOI: 10.1145/3563345.